Use your browser's search function to easily search all the notes and exercises below...

Setup

Exercises

Objective

Welcome to your first exercise! We're delighted to have you along for the course.

Before we can start creating Rails apps, we need to get Ruby and Rails installed on your computer. Depending on your operating system, you may already have a version of Rails installed. But for this course you'll need to be running Rails **7.0.4**.

So just follow the steps below for your favorite operating system—Mac, Windows, or Linux— and we'll be on our way!

Mac OS X

The best way we've found to install Ruby and Rails on a Mac is using rbenv. It's a command-line tool that makes it easy to install and manage Ruby versions. We'll be installing Ruby 3.1.4 and Rails 7.0.4.

If you already have Ruby 3 installed, you can jump straight to installing Rails.

Note that Mac OS X ships with a version of Ruby. However, it's best not to mess around with the system-installed Ruby as it's intended to be used by the operating system and apps installed by Apple. So we'll use rbenv to install a separate user-level Ruby environment, rather than changing the system-installed Ruby.

1. Start by finding the Terminal application (it's under the *Applications -> Utilities* directory) and dragging it onto your dock. You'll end up using Terminal a lot as a Rails developer, so it's good to have it handy. Then open a new Terminal session. You should see a new window with a cursor and a prompt that looks something like this:

   ```
   enoch:~ mike$
   ```

   If this is the first time you've seen this side of a Mac, it may seem rather intimidating. Don't let it throw you. It's simply a way to interact with your computer by entering commands. The default prompt includes the computer name (*enoch* in my case), the current working directory (tilde represents your home directory), the current user name (mike), and a trailing $ which is the prompt for input.

   We'll use the command prompt throughout the course to run commands and perform tasks on our Rails app. In fact, here comes our first command...

2. To install rbenv, first we need to install Homebrew which makes it easy to install and compile software packages from source.

   To install Homebrew, copy the following command and paste it after your Terminal prompt:

   ```
   /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
   ```

   If you're asked about installing Apple's Command Line Tools for Xcode, go ahead and answer "yes".

3. Now we can use Homebrew to install rbenv. To do that, run the following command in your Terminal session:

   ```
   brew install rbenv
   ```

4. Next, set up rbenv in your Terminal's shell by running the following command and following the printed instructions:

   ```
   rbenv init
   ```

5. Then close your Terminal session and open a fresh new Terminal session so that the changes take effect.

6. Now that we have rbenv installed and the Terminal's shell set up properly, we're ready to install Ruby 3.1.4. To do that, type

   ```
   rbenv install 3.1.4
   ```

   This will download, compile, and install Ruby into a directory managed by rbenv. (If you're curious, it's under the `~/.rbenv` directory.)

   Installing Ruby may take a while, so feel free to grab a refreshing beverage or a tasty snack. Mmm...

7. When installation is complete, set Ruby 3.1.4 as the global Ruby version by typing:

   ```
   rbenv global 3.1.4
   ```

   This sets Ruby 3.1.4 as the *default* version to be used whenever you open any new Terminal sessions.

8. Then verify that Ruby 3.1.4 is the current Ruby version by typing:

   ```
   ruby -v
   ```

   You should see something like the following, though your exact patch number, date, and revision may be slightly different:

   ```
   ruby 3.1.4p223 (2023-03-30 revision 957bb7cb81) [arm64-darwin22]
   ```

9. With Ruby installed, now we're ready to install Rails! Rails is distributed via RubyGems: the standard Ruby package manager. When you installed Ruby, the RubyGems system came along for the ride. With RubyGems already installed, it's easy to install Rails and its dependencies.

   Install Rails 7.0.4 by typing

   ```
   gem install rails -v 7.0.4 --no-document
   ```

   Then sit back and relax as RubyGems downloads all the Rails-related gems. After a minute or so, you should end up with a couple dozen gems installed.

10. Installing Rails gives us a `rails` command, but to use it we first need to inform rbenv about that command. To do that, type

    ```
    rbenv rehash
    ```

11. Finally, verify that Rails 7.0.4 was successfully installed by typing

    ```
    rails -v
    ```

    You should see:

    ```
    Rails 7.0.4
    ```

12. Finally, you'll need to install Git (a distributed version control system) if you don't already have a version installed. You can check if you have Git installed by

    ```
    git --version
    ```

    We won't use Git until later in the course, but Rails will auto-generate some handy Git-related files if we have Git installed from the get-go.

Excellent! Now everything you need is installed.

Choose A Code Editor

Throughout the course we'll be creating a Rails application by writing Ruby code in one or more files. You'll need a code editor to create and edit these files. The editor doesn't need to have a lot of features. In fact, a basic code editor that has Ruby syntax highlighting and a file/directory browser works best.

We'll make the decision easy for you. Unless you already have a code editor that you're very comfortable using, we recommend using Visual Studio Code as we do in the videos. VS Code runs equally well on Mac, Windows, or Linux and it's completely free! We also recommend installing the Simple Ruby ERB extension which provides both Ruby and ERB syntax highlighting.

You're now ready to start creating Rails apps!

Windows 11

The best way we've found to install Ruby and Rails on Windows is using RubyInstaller. It's a self-contained Windows-based installer that includes a Ruby language execution environment and the Windows DevKit.

1. Navigate to the RubyInstaller downloads page and click the **Ruby+Devkit 3.1.4-1 (x64)** executable installer. Save the file to your *Desktop*, for example, and then run the file once it has finished downloading. You may need to use *Windows Explorer* to navigate to where you saved the executable file and double-click it to start the installation process.

2. After accepting the license agreement, the installer will ask you to select components to install. **Make sure the checkmarks labeled "Ruby RI and HTML documentation" and "MSYS2 development toolchain" are checked.** MSYS2 is required to install Ruby gems with C extensions, and some of the gems used by Rails do indeed have C extensions. Then click "Next" and Ruby will be installed in the `C:\Ruby31-x64` directory as promised.

   Wait! Do not click "Finish" on the last installer screen until you've made sure the checkmark labeled "Run 'ridk install' to set up MSYS2 and development toolchain" is checked. With that checkbox checked, when you click "Finish", a new command window will open asking you which components of MSYS2 you want to install. Choose "1" which is the base MSYS2 installation. It will proceed to install a bunch of stuff, and then return back to the prompt asking which components you want to install. You should see the message "MSYS2 seems to be properly installed". Go ahead and close that command window.

3. Next, open a new command prompt by selecting the *Start* menu, typing `cmd` into the search box, and pressing `Enter`. You should see a new window with a blinking cursor and a command prompt that looks something like this:

   ```
   C:\Users\mike>
   ```

   If this is the first time you've seen this command prompt it may seem rather intimidating, but don't let it throw you. It's simply a way to interact with your computer by entering commands. We'll use the command prompt throughout the course to run Ruby and use some related command-line tools. In fact, here comes our first command...

4. Verify that Ruby 3.1.4 was successfully installed by typing the following at the command prompt:

   ```
   ruby -v
   ```

   You should see something like this, though your exact patch number, date, and revision may be slightly different:

   ```
   ruby 3.1.4p223 (2023-03-30 revision 957bb7cb81) [x64-mingw-ucrt]
   ```

   If instead you see "Command not found", then you either need to open a *new* command prompt and try again, or check that your `PATH` environment variable includes the `C:\Ruby31-x64\bin` directory.

5. Next, install Rails 7.0.4 by typing

   ```
   gem install rails -v 7.0.4 --no-document
   ```

   Then sit back and relax as all the Rails-related gems are downloaded. After a few minutes, you should end up with 40 or so gems installed.

6. And verify that Rails 7.0.4 was successfully installed by typing

   ```
   rails -v
   ```

   You should see:

   ```
   Rails 7.0.4
   ```

7. Finally, you'll need to install Git (a distributed version control system) if you don't already have a version installed. You can check if you have Git installed by

   ```
   git --version
   ```

   We won't use Git until later in the course, but Rails will auto-generate some handy Git-related files if we have Git installed from the get-go.

Excellent! Now everything you need is installed.

Choose A Code Editor

Throughout the course we'll be creating a Rails application by writing Ruby code in one or more files. You'll need a code editor to create and edit these files. The editor doesn't need to have a lot of features. In fact, a basic code editor that has Ruby syntax highlighting and a file/directory browser works best.

We'll make the decision easy for you. Unless you already have a code editor that you're very comfortable using, we recommend using Visual Studio Code as we do in the videos. VS Code runs equally well on Mac, Windows, or Linux and it's completely free! We also recommend installing the Simple Ruby ERB extension which provides both Ruby and ERB syntax highlighting.

You're now ready to start creating Rails apps!

Linux

Folks who run Linux tend to already be comfortable with the command line, installing and building software, and configuring things just the way they like. We're not so bold as to tell you *exactly* how to tweak your Linux install. The important thing is that you get Ruby 3.1.4, Rails 7.0.4, and SQLite 3 installed to match the versions we'll use throughout this course.

Unless you have a strong preference, we recommend installing Ruby on Linux using rbenv. It's a command-line tool that makes it easy to install and manage Ruby versions.

1. Start by updating `apt-get` and installing the dependencies required for rbenv and Ruby:

   ```
   sudo apt-get update

   sudo apt-get install git-core curl zlib1g-dev build-essential libssl-dev libreadline-dev libyaml-dev libsqlite3-dev sqlite3 libxml2-dev l
   ```

2. Then install rbenv:

   ```
   sudo apt install rbenv
   ```

   If you prefer a manual approach, follow the steps in the Basic GitHub Checkout section of the official rbenv installation documentation.

3. Once you have rbenv installed, it's then easy to get Ruby and and Rails installed by picking up with step 6 in the Mac OS X instructions above.

If you need additional help with other shells, more examples can be found in the Basic GitHub Checkout section of the official rbenv installation documentation.

Choose A Code Editor

Throughout the course we'll be creating a Rails application by writing Ruby code in one or more files. You'll need a code editor to create and edit these files. The editor doesn't need to have a lot of features. In fact, a basic code editor that has Ruby syntax highlighting and a file/directory browser works best.

We'll make the decision easy for you. Unless you already have a code editor that you're very comfortable using, we recommend using Visual Studio Code as we do in the videos. VS Code runs equally well on Mac, Windows, or Linux and it's completely free! We also recommend installing the Simple Ruby ERB extension which provides both Ruby and ERB syntax highlighting.

You're now ready to start creating Rails apps!

Create the App

Exercises

Objective

Now that we have Rails installed, it's time to start creating a Rails app!

In this exercise we'll create a new Rails app called `flix`. It's a super-short exercise just to get everything up and running. We'll incrementally add features to the `flix` app throughout subsequent exercises.

1. Generate the Skeleton App

All Rails apps have a common application directory structure and a set of supporting files. Creating all the directories and files by hand would be really tedious, not to mention being prone to error. Instead, with a single command we can generate a working Rails app. We often refer to this as a *skeleton app* because it's a bare-bones app that we then customize by putting some meat on the bones. Bone appétit!

1. Start by creating a directory named `rails-studio` to hold the app you'll create while taking this course. Then change into that directory. The commands to do that are the same regardless of which operating system you're using, but the directory structure is slightly different.

   If you're running **Mac OS X or Linux**, create the `rails-studio` directory in your home directory (represented by the tilde character). To create the directory and change into it, type the following two commands:

   ```
   mkdir ~/rails-studio
   cd ~/rails-studio
   ```

   If you're running **Windows**, create the `rails-studio` directory in the top-level `C:\` directory (represented by the backslash character) since Windows doesn't really have the concept of a user's home directory. To create the directory and change into it, type the following two commands:

   ```
   cd c:\
   mkdir \rails-studio
   cd \rails-studio
   ```

2. Next, use the `rails` command to generate a new Rails app called `flix`:

   ```
   rails _7.0.4_ new flix
   ```

   > **Note:** The `_7.0.4_` part makes sure that the correct version of Rails is used to generate the app, in the event that you have multiple versions of Rails installed.

Running this command will generate all the standard Rails directories and supporting files inside of a new `flix` directory. All the required gems listed in the `Gemfile` will be automatically installed, as well.

3. Go ahead and change into the new `flix` directory:

```
cd flix
```

4. Then open that directory in your code editor. For example, if you're using Visual Studio Code then you can either open the directory using the *File -> Open...* menu item or from the command line by typing:

```
code .
```

Have a quick look around at what got generated, just to get your bearings. There's a lot of stuff in there, but don't let it throw you. We'll visit each directory as needed throughout the course.

2. Run the App

The Rails application generator is kind enough to generate a *working* application, which means we can run it without having to first fuss around with configuration.

1. Inside the `flix` application directory, start your new Rails app by typing:

```
rails server
```

Or, if you're the type of person who loves to show off your power moves around the office, you can use the shortcut:

```
rails s
```

This command starts a web server and you should see output that looks something like this:

```
=> Booting Puma
=> Rails 7.0.4 application starting in development
=> Run `bin/rails server --help` for more startup options
Puma starting in single mode...
* Puma version: 5.6.4 (ruby 3.0.4-p208) ("Birdie's Version")
*  Min threads: 5
*  Max threads: 5
*  Environment: development
*          PID: 32727
* Listening on http://127.0.0.1:3000
* Listening on http://[::1]:3000
Use Ctrl-C to stop
```

Now you have a web server listening on port 3000 of your local machine.

2. Next, open your favorite web browser and browse to the URL http://localhost:3000. You should see the default Rails welcome page. Yay!

As a skeleton app this app doesn't do much (yet), but it's always good to fire up a newly-generated app and make sure everything works before customizing it.

3. Now would be a really good time to check the information on the default Rails page to verify that you're running the correct versions of Ruby and Rails. :-)

4. Finally, leave the app running in your current Terminal or command prompt window, and open a new session. That way you can type the commands in future exercises without having to stop and restart the server. Don't forget to change into the `flix` application directory in your new session.

Solution

All the exercise solutions for the movie-review app you're writing in the exercises, as well as the code for the events app we're writing in the videos, is available for download in the code bundle file (right-click to save).

When you unzip the file, you'll end up with a directory named `pragstudio-rails7-code`. Inside that directory you'll find the following directories:

- `eventz` contains the events app we build in the videos. The code is organized into directories matching the course modules. For example, the `create-app` directory contains a snapshot of how the `eventz` code looks at the end of the "Create the App" video.
- `flix` contains the application you build in the exercises. The code is organized into directories matching the course modules. For example, after completing the exercise for the "Create the App" module, your code should roughly match the code in the `create-app` directory.
- `prepared-files` contains files we've prepared for you to copy into the project as needed later on in the course to save tedious typing.

The full solution for this exercise is in the `create-app` directory of the code bundle.

Bonus Round

Some folks just aren't satisfied to gloss over all those generated files and directories. We know, it's just too tempting to go digging around in there. So if you're feeling curious, go ahead and get familiar with what got generated. Here's a rundown of what's in each directory:

- **app** is where you'll put the bulk of your application code. This is where you'll spend most of your time. Rails puts a lot of emphasis on keeping code organized, so the `app` directory has a number of subdirectories:
    - **models**, **views**, and **controllers** are appropriately-named directories where you'll organize your (wait for it) model, view, and controller code.
    - **assets** contains sub-directories where you'll store your application's static assets such as images and stylesheets.
    - **channels** is where you put Ruby classes for handling real-time features using Action Cable.
    - **helpers** is where you put Ruby modules that define utility methods for views.

    - **javascript** is where you put JavaScript modules that get compiled by Webpack.
    - **jobs** is where you put Ruby classes for running background jobs.
    - **mailers** is where you put Ruby classes for generating and sending e-mails.
- **bin** isn't something you'll mess with. It contains the `rails` script file and other scripts.
- **config** is where you go to tweak the configuration settings of your application. Initially you won't have to touch most of the files in here because generated Rails apps are configured with "sensible defaults". Later in this course we'll make stops at the following key places:
    - **database.yml** configures the database used by each environment.
    - **routes.rb** maps incoming requests (URLs) to application code.
    - **environments** is a directory that contains three files that define specific settings for each environment: development, test, and production.
    - **initializers** is a directory where you put Ruby code that needs to be run when the application starts.
- **db** contains everything related to the database, including migration files and, in the case of using the default SQLite3 database, the database file itself.
- **lib** is where you put any reusable "library" code that's not a model, view, or controller. It's more common these days to package this sort of code as a gem. But it has one important subdirectory:
    - **tasks** is where you would put any custom Rake tasks for your application, with each file having a `.rake` extension.
- **log** is where Rails automatically creates log files so you have a record of what happened when your app ran. Each environment gets its own log file.
- **public** is the document "root" directory of the app. It contains static files that are served directly by the web server. For example, `404.html` lives here and gets served when a page can't be found.
- **storage** is where Active Storage stores uploaded files when running in the development environment.
- **test** is where you put test files if you use the default testing library.
- **tmp** is where Rails stores any temporary files needed by the app. We can't recall the last time we even peeked in this directory.
- **vendor** is where we put third-party code that's not packaged as a gem that's declared in the `Gemfile`.
- Last, but by no means least, the **Gemfile** file in the top-level directory contains a list of third-party dependencies (Ruby gems) that your application needs.

See, it wasn't too intimidating after all. :-)

Wrap Up

Pat yourself on the back—you just created your first Rails app! Now you're ready to start customizing it with your own code. Onward and upward to the next section!

Views and Controllers

Exercises

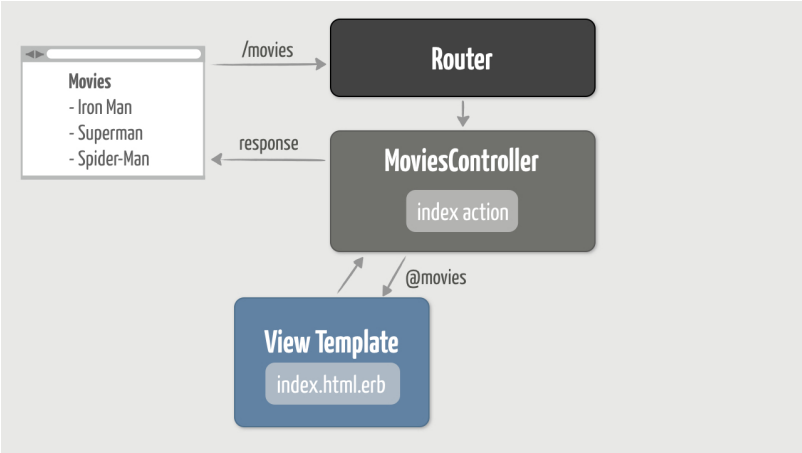
Objective

Now that we've generated a skeleton application, it's time to start customizing it by adding features. The first feature we want to add is a page that lists movie titles. This is conventionally called the "index" page. The index page will be displayed when a user requests the URL `http://localhost:3000/movies`.

To make that work, we have three primary tasks:

1. Add a route to handle requests for `/movies`
2. Generate a `MoviesController` and define an `index` action that prepares an array of movie titles
3. Create a view template (`index.html.erb`) that dynamically generates an HTML list of movie titles

Here's a visual of our objective:



We'll take this step-by-step, using the error messages as a guide along the way. So let's get started!

1. Display a List of Movie Titles

We'll start by displaying a hard-coded list of movie titles just to shake everything out before making things more dynamic.

1. Start by making sure you have the `flix` app running in a Terminal or command prompt window:

   ```
   rails s
   ```

   If the app is already running on port 3000, then you'll get an "Address already in use" error.

2. It's common to leave the server running in one window, and using a second Terminal or command prompt window to type in commands. So make sure you have a second command prompt ready to accept commands inside of the `flix` project directory.

3. Then use your web browser to browse to http://localhost:3000/movies. You should get the following error message:

   ```
   Routing Error

   No route matches [GET] "/movies"
   ```

   Error messages like this are your friend. They're actually trying to help you fix (and understand) problems. Unfortunately, we're accustomed to errors being bad things, and when we see them we panic! Here's a little secret: Even the best Rails programmers get error messages—often! But they don't panic. Instead, good programmers actually *read* the error messages. Seriously, if you take the time to understand what Rails is trying to tell you in an error message, after a while you'll be able to quickly recognize (and fix) common errors.

   In this case, here's what happened: The web browser issued a request for the URL `http://localhost:3000/movies`. The `http://localhost:3000` part of that URL is the address of the web server that fired up when we started the Rails app. So the web server received the request, then forwarded the `/movies` part of the URL on to the *router*. (Here comes the gist of the error message.) The router picked up the request and tried to find a route that matched a `GET` request for `/movies`. But we haven't told the router what to do with those types of requests, so we get the error.

4. You tell the router how to handle certain requests by changing the `config/routes.rb` file. Open that file and remove all the comments until your file simply looks like this:

   ```
   Rails.application.routes.draw do
   end
   ```

   What we're left with is an empty Ruby block. Out of the gate, the router doesn't have any routes.

5. Returning to our objective, when the router receives a `GET` request for `/movies`, we want a listing of movie titles to get sent back to the user's browser. The router's role in that process is simply to call some Ruby code we write to handle the request. In Rails parlance, the router calls an *action* (a Ruby method) that's defined in a *controller* (a Ruby class). The syntax for adding a route is a bit unorthodox, so here's a reminder of the generic format:

   ```
   verb "url" => "name_of_controller#name_of_action"
   ```

   The stuff on the left-hand side of the `=>` identifies what the request looks like and the stuff on the right-hand side identifies the code to run to handle that request. We know what the request looks like, but what should we use as the names on the right-hand side?

   Because we want a listing of movies, by convention the name of the controller will be `movies` and the name of the action will be `index`. We'll actually create those a bit later, but for now it's enough just to know their names.

   In the `config/routes.rb` file, add a route that maps a `GET` request for `/movies` to the `index` action of the `movies` controller.

   Answer:

   ```
   Rails.application.routes.draw do
     get "movies" => "movies#index"
   end
   ```

6. Refresh your browser (you're still accessing http://localhost:3000/movies) and this time you should get a different error message:

   ```
   Routing Error

   uninitialized constant MoviesController
   ```

Oh no, another error message! Yup, and you're not panicking. You're calmly reading what it says. It tells us that the route we added is trying to do what we told it to do. It's trying to send the request to a controller. We said the name of the controller was `movies`, but the router applied a naming convention and went looking for a Ruby class named `MoviesController`. We don't have a class with that name in our project, so the next step is to create it.

7. Back at your command line prompt, inside of the `flix` project directory, generate a `MoviesController` class by typing:

   ```
   rails generate controller movies
   ```

   Or if you're feeling frisky, you can use the shortcut:

   ```
   rails g controller movies
   ```

   Note that the generator takes the name of the controller, which by convention should be plural. In this case, we want a controller named "movies".

8. You should end up with an empty `MoviesController` class defined in the `app/controllers/movies_controller.rb` file, like so

   ```
   class MoviesController < ApplicationController
   end
   ```

   Although it's not important right now, it's worth noticing that the class inherits (subclasses) from the `ApplicationController` class. `ApplicationController` is the parent class of all controllers and can be found in the `app/controllers/application_controller.rb` file. It's through this inheritance relationship that `MoviesController` knows how to act like a controller.

9. Refresh your browser again and this time you should get yet another (different!) error message:

   ```
   Unknown action

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

   OK, now we're getting somewhere. This time the router found the `MoviesController` class, and then tried to call the `index` action. That's what the route says to do. But as the error so smartly points out, the `MoviesController` doesn't have a method named `index`.

10. In the `MoviesController` class, define an empty `index` action. Remember, an *action* is simply a publicly-accessible Ruby method defined in a controller class.

    Answer:

    ```
    class MoviesController < ApplicationController
      def index
      end
    end
    ```

11. Refresh your browser again and—you guessed it!—you should get another error message (it's like we're poking a hole through our app, one error message at a time):

    ```
    No template for interactive request

    MoviesController#index is missing a template...
    ```

    We're almost there! Again, the error message is fairly helpful. The router has now successfully sent the request to the `index` action of the `MoviesController`. The `index` action runs, but we're missing something. We need to send a list of movie titles back to the browser. To do that, the action needs to render a *view template* which in turn generates HTML. So we get the error because we don't have a view template for the `index` action.

    Wait, our `index` action is empty. So how does Rails know which view template to render? The short answer is "convention over configuration." Whenever an action runs and you don't explicitly tell it which view template to render, Rails uses a simple naming convention to find the corresponding view template. In this case, because the name of the action is `index`, which is defined in the `MoviesController` class, Rails assumes it should look for a view template file called `index.html.erb` in the `app/views/movies` directory. And the error message tells us that we're missing that file.

12. Create a file named `index.html.erb` in the `app/views/movies` directory.

    Then inside the new file, add the following HTML snippet:

    ```
    <ul>
      <li>Iron Man</li>
      <li>Superman</li>
      <li>Spider-Man</li>
    </ul>
    ```

13. Make sure to save your new file!

14. Refresh your browser, and this time you should be rewarded with a list of blockbuster movie titles.

Excellent! Now we know that the request is successfully flowing through our application: from the router, to the action in the controller, through the view template, and back out to the web browser.

2. Use an Instance Variable

This works, but we'd like to make things more dynamic. After all, these are superhero movies, and they deserve a little POW! The first step toward that is to realign the responsibilities of the view and the controller.

Currently the view has a hard-coded list of movie titles. We'll likely want to add or remove movies, and later on we'll want to pull them from a database. To that end, we'd rather the view not know the specifics of how movies are created. The view should only be concerned with how to display the movies. It's the controller action's job to set up data for the view to display. And the way an action passes data to a view is by setting instance variables. So any data we want to make available to the view template needs to be assigned to instance variables in the action.

1. In the `index` action, assign an array of movie titles (strings) to an instance variable named `@movies`.

   Answer:

```
def index
  @movies = ["Iron Man", "Superman", "Spider-Man"]
end
```

2. As a quick check that the movies are accessible in the view, add the following ERb snippet to the bottom of the `index.html.erb` view template:

```
<%= @movies %>
```

3. Refresh your browser and you should see the movie titles displayed in an array format, like this:

```
["Iron Man", "Superman", "Spider-Man"]
```

That format isn't very user friendly (unless all your users are programmers), but we'll fix that next...

3. Generate the List Dynamically

Finally, we want to replace the hard-coded list of movie titles in the `index.html.erb` file with a dynamically generated list that reflects the titles in the `@movies` instance variable. To do that, we'll need to use a mix of HTML and Ruby code in the `index.html.erb` file. We'll use Ruby to iterate through all the strings in the `@movies` instance variable. And for each movie title, we'll generate an HTML list item.

This is where the name of view template files becomes important. The name of our view file is `index.html.erb`. It has three parts: `index` is the name of the action, `html` is the type of content the view generates, and `erb` is the templating system used to generate the content.

In a Rails app, ERb (Embedded Ruby) is the default templating system. It lets us mix Ruby code with HTML in a view template. Then, when the view template is rendered, the embedded Ruby code runs and we end up with a dynamically generated response.

So a typical Rails view template is just a mixture of HTML tags and ERb tags. ERb tags come in two flavors:

- `<%= a Ruby expression %>`: runs the Ruby expression and substitutes the result of the expression into the view template
- `<% a Ruby expression %>`: runs the Ruby expression but does **not** substitute the result into the template (no text is generated). This form is typically used for Ruby conditonal and control flow statements.

The difference between the two is really subtle: the first flavor has an equal sign (=). Every Rails programmer on the planet has at one point mistakenly forgotten to include that extra character. Then they scratched their head for a few minutes wondering why nothing showed up in their browser. Consider yourself warned!

1. In the `index.html.erb` file, use Ruby to iterate (loop) through each movie title in the `@movies` array. For each movie title, generate an HTML `li` element that contains the movie title. The goal is to end up with the same HTML as the hard-coded list of movie titles.

   Answer:

   ```
   <ul>
   <% @movies.each do |movie| %>
     <li><%= movie %></li>
   <% end %>
   </ul>
   ```

2. Once you have a nicely-formatted listing of all the movie titles that were set up in the `index` action, go ahead and remove the hard-coded list of movie titles. Follow that up with a celebratory KA-CHOW!

3. Finally, now that you've separated the concerns of where the data comes from (the controller action) from how it's displayed (the view template), you can easily make changes in one place without adversely affecting the other. For example, add another movie title to the `@movies` array.

   Answer:

   ```
   class MoviesController < ApplicationController
     def index
       @movies = ["Iron Man", "Superman", "Spider-Man", "Batman"]
     end
   end
   ```

   Refresh and the new movie title should be displayed in the listing. What's important here is you didn't have to change the view template. It simply takes whatever data it's given and displays it.

The benefits of this separation of concerns aren't very pronounced at this point, but being able to make changes in one place becomes increasingly important as the application grows.

4. View the Server Log

It's useful to have a look at what happens behind the scenes. Every time Rails handles a request it leaves an audit trail in the server log.

To see this, look in the Terminal or command prompt window where your Rails server is running. Then refresh the movie listing page in your browser. Then, back in the server log, you should see something like this:

```
Started GET "/movies" for ::1 at YYYY-07-08 15:42:39 -0700
Processing by MoviesController#index as TURBO_STREAM
  Rendering layout layouts/application.html.erb
  Rendering movies/index.html.erb within layouts/application
  Rendered movies/index.html.erb within layouts/application (Duration: 0.5ms | Allocations: 224)
  Rendered layout layouts/application.html.erb (Duration: 24.2ms | Allocations: 14785)
Completed 200 OK in 27ms (Views: 26.2ms | Allocations: 16025)
```

Notice that it shows the incoming request, the controller and action that handled it, and the view template that was rendered. Every time you refresh the page, this request-response cycle happens. Go ahead and refresh a few more times just to give Rails a workout.

So when you want to know what *really* happened, have a look at the server log. It's also a great tool for trouble-shooting. For example, if you click a link or button and nothing seems to happen, peek at the server log for any error messages or other clues.

A permanent record of every request is kept in the `log/development.log` file. In fact, what you see in the Terminal or command prompt window where your Rails server is running is simply a reflection of what Rails adds to the server log file.

Solution

The full solution for this exercise is in the `views-and-controllers` directory of the code bundle.

Wrap Up

Whew! We accomplished a lot in this exercise. We took a stock Rails app and customized it to display a listing of movies. Not too shabby for the first exercise. As with learning anything new, the initial hurdle is to get comfortable with the vocabulary. We touched on all the major stages a request goes through when it enters a Rails app. Along the way you:

- added your first route
- generated your first controller
- defined your first action
- wrote your first view template

Early on we said that Rails uses an "MVC" design. In this section we focused on the view (V) and controller (C) to get something up and running quickly. In the next section we'll shift gears and focus on the model (M) which will let us put the movies in a database.

Onward and upward to the next section!

Dive Deeper

If you're new to the Ruby programming language or found yourself struggling with the Ruby syntax in this exercise, check out our Ruby course.

It follows the same style as this course with both videos and exercises, and by the end of it you'll really understand Ruby syntax, design principles, and techniques. And once you're comfortable with the Ruby language, you'll have a much smoother ride building Rails apps. 🙌

We think you're gonna love it!

Models

Exercises

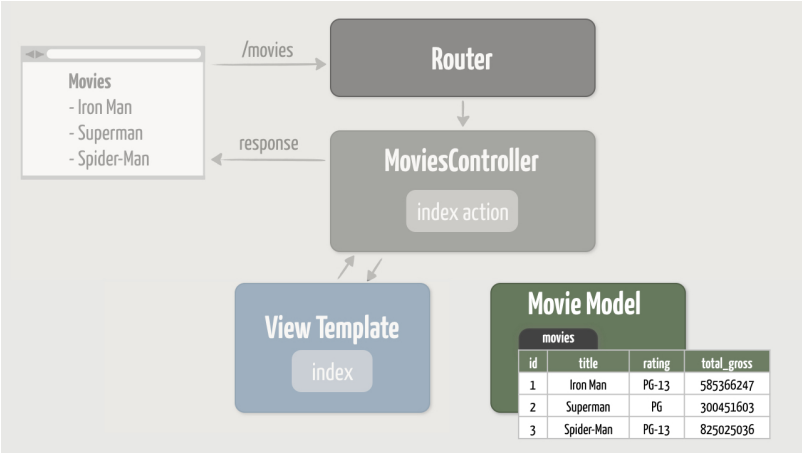
Objective

In the previous exercise, we brought our web app to life in the browser using a controller and a view. In this exercise, we'll turn our attention to the model with the goal being to store movies in the database.

The objectives of this exercise are to:

- generate a `Movie` model
- use a migration to create a `movies` database table
- create three movies and save them in the database
- read movies from the database
- update a movie's attributes in the database
- delete a movie from the database

In the end we'll have a `Movie` model that we can use to create, read, update, and delete (CRUD) movies in the database. Here's visually what we want:



In the next section we'll actually connect the model to the controller, but that's getting ahead of ourselves. First things first.

1. Generate the Movie Model & Migration

In Rails, a *model* is simply a Ruby class that lives in the `app/models` directory. The role of a model is to provide access to application-level data and encapsulate the application's business logic. Although it's not a requirement, most models in a Rails app are connected to an underlying database since that's where application data is typically stored.

Up to this point we've been representing a movie simply as a title string. Now we want a movie to have three attributes: a title, a rating such as PG, and the total gross (in dollars) the movie has earned. So we'll represent the movies in the database with the following fields and types:

name	type
title	string
rating	string
total_gross	decimal

To do that, we'll need to create a database table to store the movies and also define a `Movie` model that accesses that table. Sounds like a lot of tedious configuration work, right? Thankfully, Rails takes all the grunt work off our hands.

In the same way that we used a generator to quickly create a controller in the previous exercise, we'll use another generator to generate the `Movie` model and the instructions for creating the corresponding database table. And by following a simple set of conventions, Rails lets us avoid any configuration. It's another example of "convention over configuration."

1. First, just to get a feel for how to use the model generator, print the usage information by typing the following while inside of your `flix` project directory:

   ```
   rails generate model
   ```

   At the top you'll see that the generator takes the model name followed by a list of fields and types separated by colons:

   ```
   rails generate model NAME [field[:type][:index] field[:type][:index]] [options]
   ```

   For future reference, here's a list of supported database column types:

   - string
   - text
   - integer
   - decimal
   - float
   - boolean
   - binary
   - date
   - time
   - datetime
   - primary_key
   - timestamp

2. Now use the generator to generate a model named `Movie` (singular) with the fields and types listed above.

   Answer:

   ```
   rails g model movie title:string rating:string total_gross:decimal
   ```

   Running that command should generate several files.

3. Open the generated `Movie` model in the `app/models/movie.rb` file and it should look like this:

   ```
   class Movie < ApplicationRecord
   end
   ```

   That's all we got!? Just an (empty) Ruby class! It doesn't have any attributes or methods. Not even a gratuitous comment. The only intriguing part is that it inherits from the `ApplicationRecord` class. Believe it or not, that's all we need for a model to connect to its underlying database. It's not magic; it's just an example of the power of Rails conventions (and Ruby meta-programming). We'll dig into this a bit more later.

4. The generator also generated a timestamped database migration file in the `db/migrate/YYYYMMDDHHMMSS_create_movies.rb` file. The YYYYMMDDHHMMSS timestamp is embedded in the filename so migrations can be kept in chronological order. Open that file and you should see the following:

```
class CreateMovies < ActiveRecord::Migration[7.0]
  def change
    create_table :movies do |t|
      t.string  :title
      t.string  :rating
      t.decimal :total_gross

      t.timestamps
    end
  end
end
```

Think of a migration file as instructions for modifying your database. In this case, when we told the generator we wanted a `Movie` model that had specific fields (attributes), the generator was smart enough to generate the instructions for creating the corresponding database table. And the instructions are written in Ruby. You have to admit, that's pretty handy!

All the action happens in the `change` method. This is where we "change" the database. Since this migration needs to create the `movies` table, the generated code calls the `create_table` method and passes it the name of the table as a symbol (`:movies`). It's important to note that the name of the model is singular (movie) and the name of the database table is plural (movies). Rails uses this simple naming convention to automatically connect the model to the database table.

The `create_table` method also has an attached block. Inside of the block, table columns are created by calling methods on the `t` object which references the table being created. In this case, the generated code calls the appropriate methods to create the three columns we asked for when we ran the generator. The generator also added the `t.timestamps` line which is a shortcut that ends up creating two additional columns: `created_at` and `updated_at`. Finally, although there's no line for it here, an `id` column will be automatically created. We'll see a bit later how Rails takes care of populating these additional columns.

Think about that: You can express your migrations in generic Ruby code, and Rails automatically translates the code into a language the database understands. So no matter which database you use—SQLite, MySQL, PostgreSQL, Oracle, or the like—the same migration files will work on any of them. And that's pretty darn convenient!

2. Run the Migration

OK, so we've generated the instructions (migration) for creating the database table, but the generator doesn't actually carry out the instructions. To do that, we need to run the migration.

1. Before we pull the lever, you might be wondering which database will get changed when the migration is run. As usual, Rails is one step ahead of us. Peek inside the `config/database.yml` file and you should see something like the following (we've removed any comments):

```
default: &default
  adapter: sqlite3
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  timeout: 5000

development:
  <<: *default
  database: db/development.sqlite3

test:
  <<: *default
  database: db/test.sqlite3

production:
  <<: *default
  database: db/production.sqlite3
```

This YAML file tells Rails which database to use depending on which environment it's running in: **development**, **test**, or **production**. By default, your Rails app runs in the **development** environment. We'll talk more about the other environments a bit later in the course.

The `default` stanza is special: it specifies the configuration information that's common across all environments. Then each environment stanza references the `default` configuration to pull in those bits of configuration. In other words, using `default` avoids having to duplicate all the common lines in each environment stanza.

Notice that SQLite is configured as the default `development` database. SQLite is a lightweight, single-user database that works great for development. SQLite databases are stored in files, and the default development database will live in the `db/development.sqlite3` file. That file doesn't exist (yet).

You generally don't need to mess around with `database.yml` until you're ready to deploy your app to a production environment. When that time comes, you'll need to change the `production` settings to use another database such as MySQL, PostgreSQL, Oracle, or the like. But we'll talk more about that when we get to deployment.

The takeaway is we don't have to worry about configuring a database!

2. Now on to running the migration. This is a fairly common task that Rails automates for you. To see all the tasks at your disposal, inside the `flix` application directory type:

```
rails -T
```

The resulting list of tasks is fairly long, so here's a tip: You can filter it down to only show the database-specific tasks by adding `db` after the `-T` option, like so

```
rails -T db
```

From that list of tasks, identify the task that migrates the database and run it.

Answer:

```
rails db:migrate
```

You should see the following output:

```
== YYYY0708225131 CreateMovies: migrating =============
-- create_table(:movies)
   -> 0.0021s
== YYYY0708225131 CreateMovies: migrated (0.0022s) ====
```

Cool! Running the task executed the instructions in our migration file. It did that by looking at all the migration files in the `db/migrate` directory and running any migrations that had not already been run. In this case, we only have one migration file to run. To "run" the migration, the `change` method in that file was automatically called which created the `movies` database table.

3. Next, check the status of the migration by running the `db:migrate:status` task:

   ```
   rails db:migrate:status
   ```

   You should get the following output, though your migration id will be different:

   ```
   database: db/development.sqlite3

    Status   Migration ID    Migration Name
   --------------------------------------------------
      up     YYYY0708225131  Create movies
   ```

   Notice the word "up" in the `Status` column. That means that the migration has been run (applied). So far, so good!

4. Now try running the migration again using:

   ```
   rails db:migrate
   ```

   This time you shouldn't see any output. That's because all the migrations in the `db/migrate` directory have already been run. Rails is clever about keeping track of which migrations have already run, and only running migrations that haven't already been applied to the database. At this point we only have one migration, but in future exercises we'll create more migrations.

3. Create a Few Movies

Now that we have the `movies` database table and a `Movie` model, we can use the model to create, read, update, and delete movies in the database. We don't yet have a web interface for these operations, so we'll use the Rails `console` in the meantime. The `console` lets you interact with your app from the command line, and you get instant feedback without having to launch a browser.

Let's start by putting a few movies in the database...

1. While still inside of your `flix` directory, fire up a Rails `console` session by typing:

   ```
   rails console
   ```

   Yup, there's a shortcut for that:

   ```
   rails c
   ```

   Once it starts, you'll see a prompt. (We'll show the prompt as `>>`, but don't worry if your prompt looks different.) You're now in an `irb` session, so you can type in any Ruby expression. For example, type the following to get a random number:

   ```
   >> rand(100)
   ```

   Notice when you hit **Return**, the Ruby expression you typed in is evaluated and the result is printed out on the subsequent line (`=>`) like this

   ```
   => 82
   ```

2. What's unique (and very powerful) about this `irb` session is that it automatically loads the Rails environment. That means you can access components of your application, such as your models. For example, go ahead and try this:

   ```
   >> Movie.all
   ```

   You should get the following output:

   ```
   Movie Load (0.1ms)  SELECT "movies".* FROM "movies"
   => []
   ```

   Notice that the `console` prints out the SQL that was executed. In this case, we called the `all` method and behind the scenes Rails generated a SQL `SELECT` statement to query all the movies in the `movies` database table. We don't have any movies in our database yet, so an empty array was returned.

   So right off the bat it's looking like the `Movie` model already has some smarts about the database. Let's drill down a little deeper...

Create the First Movie

1. Start by instantiating a new `Movie` object and assigning it to a variable named `movie` so we can use it later:

   ```
   >> movie = Movie.new
   ```

   By default, when you assign to a variable, the `console` prints the value of that variable in a truncated form. So you'll see something like this:

   ```
   >> #<Movie:0x00007fa2012c7d88
   ...
   ```

   To actually see what's in the `movie` object, type:

   ```
   >> movie
   ```

Now you should get the following output:

```
#<Movie:0x00007fa2012c7d88
 id: nil,
 title: nil,
 rating: nil,
 total_gross: nil,
 created_at: nil,
 updated_at: nil>
```

Hey, this is interesting! Notice that this new object has attributes for every column in the `movies` table: `id`, `title`, `rating`, `total_gross`, `created_at`, and `updated_at`. The values are all `nil`, but the attributes exist.

Wait just a doggone minute! Let's look back at the definition of the `Movie` class:

```
class Movie < ApplicationRecord
end
```

There's nothing in this code that would indicate that the `Movie` class knows *anything* about the `movies` database table. So, how did it "magically" get all those attributes? The answer isn't as impressive as magic, but it's still very cool.

The real key lies in the `Movie` class being a subclass of the `ApplicationRecord` class (defined in the `app/models/application_record.rb` file) which in turn is a subclass of the `ActiveRecord::Base` class. ActiveRecord is a library (gem) that's included with Rails. It does all the heavy lifting when it comes to interacting with the database. So because `Movie` is a subclass of `ApplicationRecord`, it inherits special database powers from the `ActiveRecord::Base` class.

Here's what happens: When the `Movie` class is declared (like we did in the console), Rails looks at the class name (`Movie`) and assumes it should be mapped to a database table with the plural form of the name (`movies`). Then it goes off and queries the `movies` table schema to see which columns exist. It figures we probably want to be able to read and write values for each of those columns. So, through a nifty bit of meta-programming, ActiveRecord dynamically defines attributes for each column. And that's why in the output above we see those attributes on the new `Movie` object.

So, without any configuration, the `Movie` model has attributes that reflect the columns in the `movies` database table. Now we can use those attributes to finish creating our movie.

2. Returning to the `movie` object in the `console`, all of its attributes currently have `nil` values. Let's fix that. Start by assigning a value to the `title` attribute, like so:

   ```
   >> movie.title = "Iron Man"
   ```

   Then use the attribute to print the movie's title:

   ```
   >> movie.title
   ```

   What's neat about this is you read and write a model's attributes exactly as you would any attributes of a standard Ruby class. The only difference here is that you didn't have to define the attributes yourself. ActiveRecord takes care of that for you.

3. Now it's your turn. Assign values for the movie's `rating` (a string) and `total_gross` (a number). Then turn around and print each of their values.

   Hint:

   The type of the `total_gross` attribute is a `BigDecimal`, so to print it in a friendly format you'll need to use `movie.total_gross.to_s`.

   Answer:

   ```
   >> movie.rating = "PG-13"
   >> movie.total_gross = 585366247

   >> movie.rating
   >> movie.total_gross.to_s
   ```

4. With those attributes set, we're ready to put the movie in the database. Remember, we instantiated the `Movie` object by calling the `new` method, but that doesn't create a new movie in the database. To save the movie to the database, you have to explicitly call `save` at the end. Go ahead and do that now, and make sure to check out the generated SQL.

   Answer:

   ```
   >> movie.save
   ```

5. What about those other columns: `id`, `created_at`, and `updated_at`?

   Print the value of the `id` attribute:

   Answer:

   ```
   >> movie.id
   => 1
   ```

   When the movie was created, Rails automatically assigned a unique `id` as the primary key for the row in the database where this movie is stored. (Don't worry if the `id` of your movie is different.)

   Then print the values of the `created_at` and `updated_at` attributes:

   Answer:

   ```
   >> movie.created_at

   >> movie.updated_at
   ```

   When the movie was created, Rails automatically put a timestamp in the `created_at` and `updated_at` columns.

6. In addition to having dynamic attributes, the `Movie` class also inherits a number of methods for conveniently accessing the database. For example, we've already used the `all` method to fetch all the movies from the database. Use the `all` method again to verify that the movie we just created exists.

   Answer:

   ```
   >> Movie.all
   ```

   You should get an array that contains one `Movie` object representing the "Iron Man" movie.

7. You can also count all the movies in the database. Any guess as to which method does that for you?

   Answer:

   ```
   >> Movie.count
   => 1
   ```

Create A Second Movie

OK, now let's take what we learned and apply it toward putting a second movie in the database. We'll do it slightly differently this time.

1. To create our first movie, we used the `new` method and assigned attributes individually. This time create a second movie by calling the `new` method and passing it a hash with the following attribute names and values:

name	value
title	"Superman"
rating	"PG"
total_gross	300451603

   Make sure to save the movie to the database.

   Answer:

   ```
   >> movie = Movie.new(title: "Superman", rating: "PG", total_gross: 300451603)
   >> movie.save
   ```

2. Now you should have **two** movies in the database. Use a method to verify the count.

   Answer:

   ```
   >> Movie.count
   => 2
   ```

3. Then use another method to fetch all the movies in the database.

   Answer:

   ```
   >> Movie.all
   ```

   The returned array should contain two `Movie` objects: the "Iron Man" and "Superman" movies.

Create A Third Movie

Let's try that again, this time doing it all in one fell swoop...

1. Create a third movie by calling the `create` method and passing it a hash of the following attribute names and values:

name	value
title	"Spider-Man"
rating	"PG-13"
total_gross	825025036

   Hint:

   The `create` method instantiates an object in memory and, if it's valid, automatically inserts it in the database. In other words, you don't have to call `save` if you use `create`.

   Answer:

   ```
   >> Movie.create(title: "Spider-Man", rating: "PG-13", total_gross: 825025036)
   ```

2. Check that you now have *three* movies in the database.

   Answer:

   ```
   >> Movie.count
   => 3
   ```

3. Finish off by fetching all the movies.

   Answer:

   ```
   >> Movie.all
   ```

   The returned array should contain the "Iron Man", "Superman", and "Spider-Man" movies.

4. Update a Movie

Now suppose we need to update a movie's information in the database. We can use the `Movie` model to do that, too.

1. First we need to read the movie we want to update into the `console`. The `Movie` class inherits a number of methods for finding a specific movie. For example, the `find` method takes an id as the parameter and returns the movie with that primary key.

   Still in the Rails `console`, find the "Iron Man" movie by its primary key (id) and assign the resulting object to a variable named `iron_man`.

   Answer:

   ```
   >> iron_man = Movie.find(1)
   ```

2. Now change the movie's title to "Iron Man 2" and double its total gross. Make sure to save the changes to the database.

   Answer:

   ```
   >> iron_man.title = "Iron Man 2"
   >> iron_man.total_gross *= 2.0
   >> iron_man.save
   ```

3. Since we updated the record, we'd expect the `updated_at` column to have a new timestamp. Check that it was updated.

   Answer:

   ```
   >> iron_man.updated_at
   ```

4. Now let's suppose we want to change the movie's title and total gross in one operation. Use the `update` method to change the title back to "Iron Man" and the total gross to `585366247`.

   Hint:

   Call `update` with a hash of attributes to update the movie and save it in one fell swoop.

   Answer:

   ```
   >> iron_man.update(title: "Iron Man", total_gross: 585366247)
   ```

   The `update` method is a handy convenience, but it exists for a far more important reason. In most cases you'll create and update records from data submitted in an HTML form. (We'll do that later.) When the form is submitted, the form data is captured in a hash of attribute names and values. It's no coincidence then that the `update` method takes a hash of attribute names and values.

5. To check your work, print the values of the `title` and `total_gross` attributes to make sure they were updated.

   Answer:

   ```
   >> iron_man.title
   >> iron_man.total_gross.to_s
   ```

5. Delete a Movie

Finally, we might want to delete a movie from the database. Again, the `Movie` model inherits methods that make that really easy.

1. As before, the first step is to read the movie we want to delete into the console. Let's suppose we want to delete the "Spider-Man" movie, but we don't know its id. Thankfully, ActiveRecord automatically creates a set of finder methods based on the names of database columns.

   Use the `find_by` method to find the "Spider-Man" movie and assign the resulting object to a variable named `spider_man`.

   Answer:

   ```
   >> spider_man = Movie.find_by(title: "Spider-Man")
   ```

2. Then use the `destroy` method to delete the "Spider-Man" movie from the database.

   Answer:

   ```
   >> spider_man.destroy
   ```

3. To check your work, call the `find_by` method again to verify that the movie no longer exists in the database. You should get a response of `nil`.

   Answer:

   ```
   >> Movie.find_by(title: "Spider-Man")
   ```

4. Finally, just to practice what you've learned, recreate the "Spider-Man" movie using any of the creation methods you used previously.

5. When you're done, exit the `console` session by typing, wait for it... `exit` (or `Ctrl-D`):

   ```
   >> exit
   ```

Solution

The full solution for this exercise is in the `models` directory of the code bundle.

Bonus Round

Poke Around in the Database

Running the first migration automatically created the database in the `db/development.sqlite3` file. Bear in mind, you generally won't need to interact directly with the database. Instead, you'll use a model to access the database as we've done in this exercise. But just to better understand what the migration did for us, we'll dig around in the database for a moment.

1. To see what's inside the `db/development.sqlite3` file, change into your `flix` application directory and start an interactive console for the database by typing:

   ```
   rails dbconsole
   ```

   This command figures out which database you're using and opens a command-line interface to that database. We're using SQLite, so once it starts you'll see the following prompt where you can type in commands:

   ```
   sqlite>
   ```

2. First let's list the names of tables in the database. SQLite commands begin with a dot ("."), so at the prompt type `.tables` like so and hit Return:

   ```
   sqlite> .tables
   ```

   You should see the names of two tables: `movies` and `schema_migrations`.

3. Now, to see the schema for the `movies` table, type:

   ```
   sqlite> .schema movies
   ```

   You should see the following output:

   ```
   CREATE TABLE IF NOT EXISTS "movies" ("id" integer PRIMARY KEY AUTOINCREMENT NOT NULL, "title" varchar, "rating" varchar, "total_gross" de
   ```

   This is the SQL command that was executed to actually create the `movies` table in the SQLite database. What's interesting about this is that our migration file was written in Ruby, but then when the migration ran it created the table with a schema appropriate for the configured database (SQLite in this case). For example, the `string` type we specified in the migration got translated into a `varchar(255)` type in SQLite. In other words, Rails uses the database adapter to figure out the appropriate column type depending on the configured database.

4. What about that `schema_migrations` table that showed up earlier? It's a special table that Rails uses to figure out which migrations have already been run. To show its schema, type:

   ```
   sqlite> .schema schema_migrations
   ```

   You should see the following output:

   ```
   CREATE TABLE "schema_migrations" ("version" varchar NOT NULL PRIMARY KEY);
   ```

   It's a table with a single column named `version`. To see what's in the table, type the following SQL command with a trailing semi-colon:

   ```
   sqlite> select * from schema_migrations;
   ```

   You should see a fairly big number. That's actually a timestamp. In fact, it's the same timestamp that's embedded in our migration filename. That's how Rails keeps track of which migrations in the `db/migrate` directory have already been run!

5. Finally, to quit the SQLite console session, use the `.quit` command:

   ```
   sqlite> .quit
   ```

Wrap Up

Now you can easily create, read, update, and delete (CRUD) movies in the database! That's pretty exciting. It may seem like we took a lot of steps to get here, but look back and you'll notice we didn't do much typing and we were able to:

- generate a model and migration
- run a migration
- create database records three different ways
- fetch records from the database
- update records in the database
- delete records from the database

That's the power of Rails conventions. Initially, all the conventions can feel quite magical, and even confusing. But once you get comfortable with them, the conventions make things go a lot smoother.

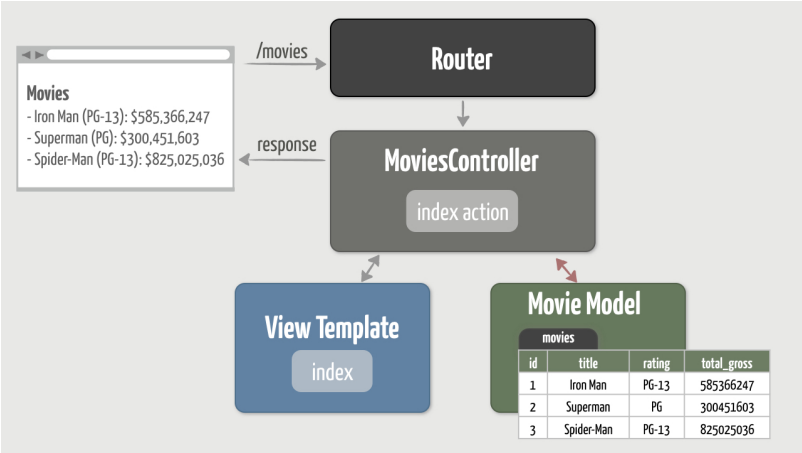
In the next section we'll connect our `Movie` model to the `MoviesController` so that the list of movies in the browser reflects the movie information in the database.

Connecting MVC

Exercises

Objective

Now that we have movies stored in the database and a `Movie` model that gives us access to those movies, it's time to connect the model to the rest of our application. The goal is to end up with a movie listing page that shows the movie information that's in the database. It's a quick exercise that puts everything together to complete the MVC triad, like so:



So let's get right to it...

1. Connect Model and Controller

We'll start by bringing the model into the mix so the controller has less responsibilities.

1. As a reminder of where things stand, jump back into your browser and go to the movie listing page (http://localhost:3000/movies). You should see three movie titles. Recall that we hard-coded these titles in an array in the `index` action of the `MoviesController`, like so:

   ```
   def index
     @movies = ["Iron Man", "Superman", "Spider-Man"]
   end
   ```

2. Now we want to arrange things so that the array contains `Movie` objects that reflect the movies in the database.

   Change the `index` action to fetch all the movies from the database. Assign the resulting array to the `@movies` instance variable.

   Answer:

   ```
   def index
     @movies = Movie.all
   end
   ```

   Notice there's no need to `require` the `Movie` class as you'd need to do in a typical Ruby program. Rails auto-loads files in certain directories, including the `models` directory.

3. Then refresh the movie listing page and you should see something like this:

   ```
   #<Movie:0x007f9e341d03b0>
   #<Movie:0x007f9e34167130>
   #<Movie:0x007f9e34165dd0>
   ```

   That's not quite what we're aiming for, but it's a good start. What we're seeing is the result of calling the default `to_s` method on a `Movie` object. Where's that being called? Look back in the `index.html.erb` view template and you'll see this line:

   ```
   <%= movie %>
   ```

   We're asking ERb to display the value of the `movie` object. To do that, ERb implicitly calls the object's `to_s` method which returns that cryptic-looking string. This is progress! It means we're actually dealing with `Movie` *objects* now, not just movie title strings.

2. Update the Movie Listing

Now that the `@movies` array contains actual `Movie` objects, let's update the movie listing page to show each movie's attributes.

1. Change the `index.html.erb` view template to display the values for each movie's title, rating, and total gross. Here's the output you're shooting for:

   ```
   Iron Man (PG-13): $585,366,247
   Superman (PG): $300,451,603
   Spider-Man (PG-13): $825,025,036
   ```

   Answer:

   ```
   <h1>Movies</h1>

   <ul>
   <% @movies.each do |movie| %>
     <li>
       <strong><%= movie.title %></strong> (<%= movie.rating %>):
       <%= movie.total_gross %>
     </li>
   <% end %>
   </ul>
   ```

2. Now your movie listing should dynamically show what's in the database. That means you can create, update, or delete movies in the database and the movie listing page will change accordingly.

Give it a whirl! Hop back into a Rails `console` session and use what you learned in the previous exercise to add, update, or delete movies. Then refresh your browser and you should see the movie listing change accordingly.

3. Then, just to show off a little, let's format the total gross amount as currency. Formatting numbers as currency is so common that Rails has a built-in *view helper* method called `number_to_currency`. If you look at the documentation for this helper method, you'll see that it takes a number and some options. The documentation also includes some helpful examples.

   Go ahead and give it a try! Use the `precision` option to remove the cents from the total gross value.

   Answer:

   ```
   <%= number_to_currency(movie.total_gross, precision: 0) %>
   ```

   We'll look at helpers in more detail in a future exercise.

Cheat Sheets

We've covered a number of commands and conventions so far, and now might be a good time to download the following cheat sheets for quick reference as you proceed through the rest of the course:

- Rails Cheats PDF: A cheat sheet of Rails commands and methods used throughout the course.
- Rails Conventions PDF: An overview of common Rails conventions.

Solution

The full solution for this exercise is in the `connecting-mvc` directory of the code bundle.

Bonus Round

Vocabulary Check

Learning a new web development framework can feel a bit like learning a foreign language. Here's a bonus exercise to help you practice your new skills.

1. The _____ connects the requested URL to the appropriate code in the controller.

   Answer:

   This is the job of the **router**.

2. Controllers and models share two similar characteristics. They are both Ruby _____ and they both _____ capabilities from a parent class.

   Answer:

   Controllers and models are both Ruby **classes** and they both **inherit** capabilities from a parent class.

3. Are controller names singular or plural?

   Answer:

   They are **plural**, as in `EventsController` or `MoviesController`.

4. Are model names singular or plural?

   Answer:

   They are **singular**, as in `Movie` model.

5. Are database table names singular or plural?

   Answer:

   They are **plural** because the table contains many records (or rows) that each represent one event, movie, or so on.

6. The model gets its power to create, read, update, and delete objects not by magic, but because it _____ from the _____ class.

   Answer:

   The model **inherits** from the **ApplicationRecord** class. `ApplicationRecord` in turn inherits from `ActiveRecord::Base` which does all the heavy lifting when it comes to interacting with the database. Subclasses of `ApplicationRecord` inherit these special database powers.

7. An _____ is simply a Ruby method defined inside a controller class.

   Answer:

   An **action**, such as `index`, is simply a Ruby method. Once again, nothing magical going on here.

8. A controller action passes data to the view via _____ _____.

   Answer:

   A controller action passes data to the view via **instance variables**, such as `@movies`.

9. A typical _____ template is a mixture of _____ tags and _____ tags.

Answer:

A typical **view** template is a mixture of **HTML** tags and **ERb** tags.

10. Which ERb tag runs the Ruby code and substitutes the result back into the template: `<%= %>` or `<% %>`?

    Answer:

    `<%= %>` tags run the Ruby code and substitute in the result. `<% %>` tags run the Ruby code, but do **not** generate any output.

Wrap Up

Now you have a model, view, and controller all working in harmony. Each component has its own responsibilities:

- The **controller** acts as a middleman. With one hand, it asks the model for application data (movies in this case) and assigns the data to instance variables. With the other hand it tells a view template to display the data in a view. Notice that the controller doesn't tell the model or view *how* to do their jobs.
- The **model** provides convenient access to application-level data. In this case, our `Movie` model is connected to a database table that contains movie information. The model completely encapsulates the lower-level details of how each row of the database table is translated into a usable object. The model isn't concerned with how the data will be used. Models can also define business logic (behavior) which we'll look at a bit later.
- The **view** is simply the presentation of data in an arbitrary format. In this case we used a view template to generate an HTML view. The view doesn't concern itself with how the data it's displaying was assembled.

Here's the thing: It's easy to start adding code to a Rails project and impress your manager with the sheer speed of Rails development. But before you put your hands on the keyboard, it's important to think about *where* to add the code. Is it a model, view, or controller concern? Because if you start mixing the MVC responsibilities, then you end up with an application that can be very difficult to change down the road. And since change is inevitable, it's best if we're prepared for it. So as we move forward with adding more features, we'll need to be mindful of separating the model, view, and controller concerns.

And on that note, in the next exercise we'll add a few more attributes to our `Movie` model. That change will force us to think through the parts of MVC that are affected.

Migrations

Exercises

Objective

Oops—we forgot a few fields when we created the `movies` database table! In addition to the existing fields, a movie could also use a textual description and the date it was released. So we need to modify the database, and that means it's time for a new migration!

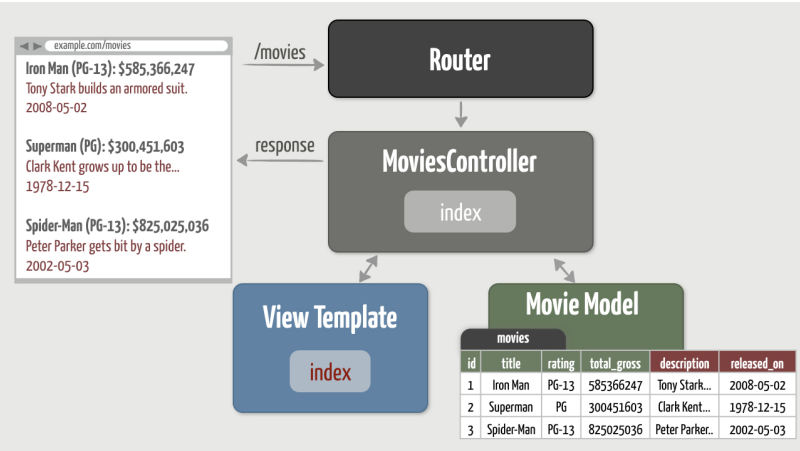
Migrations are used not only to create database tables, but also to incrementally modify them. Indeed, almost any modification you can make to a database using an admin tool or the database's native language can be done through a migration. The benefit of using a migration is you end up with a more repeatable (and automated) way for everyone on the project to keep their database in step with the code. And since migrations make it easy to add or change database columns later, we can quickly adapt to new requirements. You'll end up creating many migrations while developing a full-featured Rails app, so it's good to get more practice with them.

Here's what we need to do:

1. Generate a new migration file that adds two fields to the `movies` database table
2. Update the existing movies in the database to have values for the new fields
3. Change the movie listing page to display the new movie fields

Notice that adding fields to the database has a small ripple effect. This is fairly typical, and it gives us an opportunity to learn how to realign things. It will also help reinforce a few things we learned in previous exercises.

Here are the changes we'll make (in red):



So let's jump right into it...

1. Add New Database Fields

First, we need to add the following columns and types to the `movies` database:

name	type
description	text
released_on	date

Note that we're using a `date` type for the `released_on` field rather than `datetime` as we did for events. It makes more sense that a movie is released on a date and an event starts at a date and time.

To add those columns to the database, we'll need a new migration file. Now, when we generated the `Movie` model we also got a migration file for creating the `movies` database table. This time we'd like to only generate a new migration file. Not surprisingly, this is so common that Rails has a *migration generator*.

1. Start by printing the usage information for the migration generator:

   ```
   rails generate migration
   ```

   At the top you'll see that the generator takes the migration name followed by a list of fields and types (and an optional index) separated by colons:

   ```
   rails generate migration NAME [field[:type][:index] field[:type][:index]] [options]
   ```

   Note that each field/type pair is specified using the format `field:type` without any spaces before or after the colon. If a type isn't specified, then the default type is `string`.

2. Armed with this helpful information, now use the generator to generate a migration named `AddFieldsToMovies` with the two fields and types listed above.

   Answer:

   ```
   rails g migration AddFieldsToMovies description:text released_on:date
   ```

   Running that command should generate a `YYYYMMDDHHMMSS_add_fields_to_movies.rb` file in the `db/migrate` directory. We now have our second migration file!

3. Open the generated migration file and you should see the following:

   ```
   class AddFieldsToMovies < ActiveRecord::Migration[7.0]
     def change
       add_column :movies, :description, :text
       add_column :movies, :released_on, :date
     end
   end
   ```

   This time instead of using `create_table` the `change` method is using the `add_column` method to add both columns to the `movies` table. The `add_column` method takes three parameters: the name of the table, the name of the column, and the column type.

   How did the generator know that we wanted columns added to that specific table? That's another naming convention. If you name your migration using the format `AddXXXToYYY`, Rails assumes you want to add columns with the listed field names and types to the specified table (`YYY`). Pretty smart!

   It's worth pointing out that you could have generated the migration like this:

   ```
   rails g migration add_fields_to_movies
   ```

   In this case, because the field names and types weren't listed, you would end up with an empty `change` method, like so:

   ```
   class AddFieldsToMovies < ActiveRecord::Migration[7.0]
     def change
     end
   end
   ```

   You'd then need to put the `add_column` lines in the `change` method.

   So, if you follow the naming conventions, Rails is able to generate the complete migration for you. In cases where the naming convention doesn't make sense, you'll just need to edit the `change` method yourself.

4. Now that you have a new migration file, check the migration status by running:

   ```
   rails db:migrate:status
   ```

   This time you should see two migrations:

   ```
   database: db/development.sqlite3

    Status   Migration ID    Migration Name
   --------------------------------------------------
      up     YYYY0708225131  Create movies
     down    YYYY0708231037  Add fields to movies
   ```

   The second (new) migration has a "down" status because we haven't yet run it. Rails knows about the migration because it's in the `db/migrate` directory.

5. Go ahead and run the migration.

   Answer:

   ```
   rails db:migrate
   ```

   You should get the following output:

```
== YYYY0708231037 AddFieldsToMovies: migrating ================================================
-- add_column(:movies, :released_on, :date)
   -> 0.0006s
-- add_column(:movies, :description, :text)
   -> 0.0003s
== YYYY0708231037 AddFieldsToMovies: migrated (0.0010s) =======================================
```

Great—the migration successfully added both columns! It's important to note that only the second migration was run. The first migration wasn't re-run. Remember, when you run the `db:migrate` task, Rails looks at all the migration files in the `db/migrate` directory and only runs the migrations that haven't already been run. In other words, it only runs the migrations that have a status of "down".

6. Finally, re-check the migration status.

   Answer:

   ```
   rails db:migrate:status
   ```

   You should see both migrations marked as "up":

   ```
   database: db/development.sqlite3

    Status   Migration ID    Migration Name
   --------------------------------------------------
      up     YYYY0708225131  Create movies
      up     YYYY708231037  Add fields to movies
   ```

So now we've run two migrations. The first migration created the `movies` table and the second migration added new columns to that table.

2. Update the Movies

Next we need to update the movies in our database so that they have values for the new fields that we just added. To do that, we'll need to read each movie into the `console` and assign values to the new `description` and `released_on` attributes. And you already know how to do that!

1. Over in your `console` session, make sure to load the latest version of your application code by using the `reload!` command:

   ```
   >> reload!
   => true
   ```

2. Then find the "Iron Man" movie and assign it to a `movie` variable:

   Answer:

   ```
   >> movie = Movie.find_by(title: "Iron Man")
   ```

   Then if you evaluate the `movie` object, you should get the following output:

   ```
   #<Movie:0x00007fe6c99957e8
    id: 1,
    title: "Iron Man",
    rating: "PG-13",
    total_gross: 0.585366247e9,
    created_at: Day, DD MMM YYYY 23:00:35.870642000 UTC +00:00,
    updated_at: Day, DD MMM YYYY 23:04:14.306895000 UTC +00:00,
    description: nil,
    released_on: nil>
   ```

   Notice that the movie now has `description` and `released_on` attributes matching the new columns we added to the `movies` database table. We didn't have to change the `Movie` model. When the `Movie` class was declared in the `console`, ActiveRecord queried the `movies` table schema and automatically defined attributes for each column.

3. You probably noticed that the movie's `description` and `released_on` attributes have `nil` values, but go ahead and print them out just for practice.

   Answer:

   ```
   >> movie.description
   => nil
   >> movie.released_on
   => nil
   ```

4. Now assign values to the `description` and `released_on` attributes, and save the movie. For the date, you can assign a string with the format "YYYY-MM-DD" and it will get converted to a date.

   Answer:

   ```
   >> movie.description = "Tony Stark builds an armored suit to fight the throes of evil"
   >> movie.released_on = "2008-05-02"
   >> movie.save
   ```

5. Next, follow suit by updating the "Superman" movie.

   Answer:

   ```
   >> movie = Movie.find_by(title: "Superman")
   >> movie.description = "Clark Kent grows up to be the greatest super-hero"
   >> movie.released_on = "1978-12-15"
   >> movie.save
   ```

6. Finally, update the "Spider-Man" movie. This time, update the attributes and save the movie in one fell swoop.

   Answer:

```
>> movie = Movie.find_by(title: "Spider-Man")
>> movie.update(description: "Peter Parker gets bitten by a genetically modified spider", released_on: "2002-05-03")
```

And with that, our database is all set!

3. Update the Movie Listing

The last step is to update the movie listing page so that it displays the new movie fields. You already know how to do this, too!

1. Refresh the index page and it should come as no surprise that the movies don't appear to have a description or release date.
2. Fix that by updating the `index.html.erb` template. Just to keep things simple for now, put each new field in a paragraph tag.

   Answer:

```
<p>
  <%= movie.description %>
</p>
<p>
  <%= movie.released_on %>
</p>
```

That completes this feature! We added missing columns to the database and now those changes are being reflected on the movie listing page. It's interesting to note that we didn't need to change the `index` action in the `MoviesController`. And that's exactly as it should be!

Remember, the controller is just a middleman between the model and the view. It doesn't concern itself with the details of the model or how the data is displayed in the view. So in this case, because we didn't have to change the controller, we're confident that we have the MVC responsibilities properly divided.

Solution

The full solution for this exercise is in the `migrations` directory of the code bundle.

Bonus Round

Convention Check

Rails is full of conventions that make apps easier to build, test, change, and ultimately, pass on to the next developer to test, change, and maintain. To practice using the Rails conventions you've learned so far, suppose you wanted your Rails app to be a listing of books for sale.

1. By convention, the book listing route would map a _____ request for the URL _____ .

   Answer:

   The book listing route would map a **GET** request for the URL **/books**.

2. The name of the controller would be _____ and the name of the action would be _____.

   Answer:

   The controller would be **BooksController** and the action would be **index**.

3. If we don't explicitly tell the `index` action the name of the view template to render, what will Rails do by convention?

   Answer:

   Because the name of the action is `index`, Rails will assume it should look for a view template with a similar name: `index.html.erb`. Because the name of the controller is `BooksController`, Rails will look for the view template in the `app/views/books` directory.

4. The model would be named _____.

   Answer:

   The name of the model would be **Book** (singular).

5. The corresponding database table would be named _____.

   Answer:

   The database table would be named **books**. It is plural because it contains many records (or rows), each representing one book.

6. If the model was named `Person`, the database table would be named _____.

   Answer:

   The database table would be **people**. Yup, Rails is smart enough to figure that out!

7. After a migration has been run (or applied), its status changes from _____ to _____.

   Answer:

   Its status changes from **down** to **up**.

Wrap Up

Any time you have to make a change to the database, think "new migration." Missing a column? Time for a new migration. Need to delete a column? That's another migration. Want to rename a table or column? Yup, migrations can do that, too. In fact, we'll talk even more about migrations a little later in this course.

Indeed, creating new migration files is *really* common. A typical Rails app will end up with tens, if not hundreds, of migration files where each migration represents an incremental database change. On team projects, all those migration files get checked in to a version control system as part of the Rails project. Then, whenever someone on the project checks out a version of the application, they can get their database schema in sync with the code simply by running `rails db:migrate`. That's the beauty of migrations. You end up with an automated, repeatable way to make modifications to the database.

OK, so now that we're showing more information on the movie listing page, it looks like we could use a bit of formatting. We'll tackle that using *view helpers* in the next section.

Dive Deeper

We'll write a couple more migrations a bit later in the course so you see them used in different situations. To learn more about migrations, refer to the Rails Guides: Migrations.

View Helpers

Exercises

Objective

Next we want to format how some of the movie information is displayed on the movie listing page:

- The movie description should be truncated to 40 characters.
- When the total gross of a movie is less than $225M, it should show the word "Flop!" in place of the total gross amount.
- The movie release date should be presented in a friendly format.

These formatting requirements are all presentation (view-level) concerns, so all of our work will be done in the view layer, with one exception we'll discuss when we get there.

Now, there's a slippery slope when beginning to format view content. It's tempting to start sprinkling little bits of Ruby logic amongst the HTML tags in view templates. It's just so darn easy to do! The problem is that it gets messy really quickly, and once you start down that slope the mess tends to spread (the copy/paste way) to other templates. Before long the templates are a rat's nest and common logic is duplicated across multiple templates. Trust us, we've been there and it's not a pretty place.

To avoid those problems, we'll use *view helpers*. A view helper is simply a Ruby method that a view calls when it needs help generating output. The helper typically runs view-related logic and, depending on the result, returns the appropriate text that then gets rendered into HTML.

1. Use a Built-In View Helper

Rails is chock full of built-in helpers for things that are common across all web apps: formatting numbers, manipulating text, generating hyperlinks and form elements, and so on. So we might as well start by using what we get for free...

Use a built-in helper method to shorten the movie description to 40 characters. As a bonus, make sure the shortened description breaks on a space rather than chopping off a word.

Answer:

```
<%= truncate(movie.description, length: 40, separator: ' ') %>
```

Make sure to refresh and check your work before moving on!

2. Write a Custom View Helper

Next, when the total gross of a movie is less than $225M, we'll declare it a flop. Flop movies in our application don't even deserve to have their total gross displayed. Instead, we want to display the word "Flop!".

Rails doesn't have a helper method that handles flop movies—it's not *that* smart. But this can't be difficult. All we need is a Ruby conditional that generates different text depending on the value of the `total_gross` attribute. And where should we put that snippet of view-related logic? In a custom view helper, of course.

1. To get your bearings, remember that we're already using the built-in `number_to_currency` helper to format the total gross as currency. Find this line in your `index.html.erb` template:

   ```
   <%= number_to_currency(movie.total_gross, precision: 0) %>
   ```

   That'll work, but only in the case where the movie isn't a flop.

2. Let's start with some wishful thinking, working from the outside in. Suppose we want our custom view helper to be called `total_gross` and take a `movie` object as its parameter. Go ahead and replace the `number_to_currency` line with the following:

   ```
   <%= total_gross(movie) %>
   ```

   There, that neatly encapsulates what we want.

3. Refresh the index page and you should get the following error:

   ```
   undefined method `total_gross' for #<ActionView::Base:0x0000000000b388>
   ```

   No surprise. We haven't defined the `total_gross` method yet.

4. So where exactly should we define the `total_gross` helper? Unfortunately, the error doesn't give us a clue, but the fact that our project has an `app/helpers` directory is conspicuous. And when we generated the `MoviesController`, Rails took the liberty of creating a file named `movies_helper.rb`. Open that file and you'll see that it contains an empty Ruby module:

   ```
   module MoviesHelper
   end
   ```

   Ruby modules have a number of uses, but in this case you can think of the module as a bucket of methods. Any helper methods defined in this module will be accessible by any view. The module basically serves as an organizational aid. And it's a good idea to group related helpers into separate modules. As our `total_gross` helper is related to displaying movies, this module seems like a reasonable home.

5. Let's take another incremental step toward our goal just to get things working. Start by implementing the `total_gross` method so that it simply returns the result of calling the built-in `number_to_currency` method (yup, helpers can call other helpers).

   Hint:

   Remember that the `total_gross` method takes a `movie` object as a parameter. You'll need to pass the movie's `total_gross` value to the `number_to_currency` helper.

   Answer:

   ```
   module MoviesHelper
     def total_gross(movie)
       number_to_currency(movie.total_gross, precision: 0)
     end
   end
   ```

6. Refresh the movie listing page and the error should go away. Each movie's total gross should be displayed as currency, just as it was before. That's because we don't have any movies that are flops. But now we know that our custom helper method is being called without errors.

7. Jump into a `console` and either change one of the movies so that it has a total gross less than $225M, or create a new flop movie.

   Answer:

   ```
   >> movie = Movie.new
   >> movie.title = "Fantastic Four"
   >> movie.rating = "PG-13"
   >> movie.total_gross = 168_257_860
   >> movie.description = "Four young outsiders teleport to an alternate and dangerous universe"
   >> movie.released_on = "2015-08-07"
   >> movie.save
   ```

   Then refresh the movie listing and you should see the flop movie's total gross. Oh, the shame. We just can't allow that!

8. Next, change the `total_gross` helper to use a conditional. If the `total_gross` is less than $225M, return the string "**Flop!**". Otherwise return the total gross amount formatted as currency.

   Answer:

   ```
   module MoviesHelper
     def total_gross(movie)
       if movie.total_gross < 225_000_000
         "Flop!"
       else
         number_to_currency(movie.total_gross, precision: 0)
       end
     end
   end
   ```

9. Refresh and this time the flop movie should stick out like a sore thumb. And to think, $225M used to go a long way, even in Hollywood.

10. Before you think about crossing this task off the to-do list, here comes the part that a lot of developers unfortunately skip. In the helper, we added this innocent little comparison expression:

    ```
    movie.total_gross < 225_000_000
    ```

    That single expression is the *definition* of what it means for a movie to be a flop in our application. That's not really a view-level concern, is it? You can imagine other areas of our app wanting to know if a movie is a flop or not. And, if you really stretch your imagination, you can envision a time when some business person decides to change that definition to be less than $300M, for example. So this is actually *business logic*, and we need to encapsulate it in one definitive place in our app: the `Movie` model.

    Define an instance method in the `Movie` class called `flop?`. (By convention, Ruby methods that end in a question mark (?) return `true` or `false`.) Implement the `flop?` method so that it returns `true` if the movie's total gross is blank or less than $225M. Otherwise the method should return `false`.

    Answer:

    ```
    class Movie < ApplicationRecord
      def flop?
        total_gross.blank? || total_gross < 225_000_000
      end
    end
    ```

    Then change the `total_gross` helper to call the `flop?` method to make the decision about whether the movie is a flop or not.

    Answer:

    ```
    module MoviesHelper
      def total_gross(movie)
        if movie.flop?
          "Flop!"
    ```

```
      else
        number_to_currency(movie.total_gross, precision: 0)
      end
    end
  end
```

Refresh the movie listing to make sure everything still works as you'd expect.

11. So we shuffled code around a little, pushing logic that was in the helper back to the model. Was it worth it? Absolutely! One of the benefits of creating the `flop?` method in the `Movie` model is you can now call that method from anywhere in your app, or even from the `console`. Suppose, for example, you wanted to know whether "Iron Man" was a flop. How would you do that in the `console`?

    Answer:

    ```
    >> reload!

    >> movie = Movie.find_by(title: "Iron Man")
    >> movie.flop?
    ```

    Having your business logic totally decoupled from the web like this is not only really handy, it's also one of the secrets of building flexible applications.

That's all there is to writing custom view helpers!

3. Format the Release Date

Finally, as a nice touch we'd like to change the format of the movie's release date. Instead of showing the year, month, and day ("2008-05-02" for example) we just want to show the year the movie was released ("2008"). And for practice, we want to encapsulate that formatting in a reusable view helper method.

1. Write a custom view helper method named `year_of` that takes a `movie` object as its parameter and returns the year the movie was released. In the video we didn't show the strftime directive for formatting the year, but you can probably guess it.

    Answer:

    ```
    module MoviesHelper
      def year_of(movie)
        movie.released_on.strftime("%Y")
      end
    end
    ```

2. Then use that helper method in the `index.html.erb` template to display the year the movie was released.

    Answer:

    ```
    <%= year_of(movie) %>
    ```

3. Once you have that working, you might be delighted to learn that Rails adds a `year` method to all `Date` objects. And the `released_on` attribute is a `Date` object, so you can ask for the year. Give it a try!

    Answer:

    ```
    module MoviesHelper
      def year_of(movie)
        movie.released_on.year
      end
    end
    ```

And that completes all our formatting tasks!

Solution

The full solution for this exercise is in the `helpers` directory of the code bundle.

Bonus Round

Trying Helpers in the Console

It's often handy to experiment with view helpers in a Rails `console` session. To do that, you need to use the special `helper` object. For example, here's how to try the `number_to_currency` view helper from inside a Rails `console` session:

```
>> helper.number_to_currency(1234567890.50)
=> "$1,234,567,890.50"
```

Notice you call the helper method on the `helper` object, which you don't need to do when calling a helper method inside of a view template.

And of course in the `console` you have access to your models, so you could find a movie and format its total gross as currency, like so:

```
>> movie = Movie.find_by(title: "Iron Man")
>> helper.number_to_currency(movie.total_gross, precision: 0)
=> "$585,366,247"
```

Another useful helper method is documentation for the `pluralize`. You can copy the examples straight from the documentation and try them out in the `console`:

```
>> helper.pluralize(1, 'person')
=> "1 person"

>> helper.pluralize(2, 'person')
=> "2 people"
```

Then once you have the hang of how a helper works, you can confidently use it in a view template. Just remember that you don't use the `helper` object when inside of a view template. The `helper` object only exists in the `console`.

Wrap Up

If you take away one lesson from this exercise, let it be this: Strive to always keep your view templates as clean and concise as possible. The Ruby code should be kept to a minimum, basically just ERb tags for outputting values and iterating through collections. If it's more complicated than that, it's time to either use a built-in view helper or write a custom helper. Later on we'll talk about how to decompose view into partials, which is another design technique for eliminating duplication and generally keeping views maintainable over time.

But first, speaking of maintainable views, in the next section we'll learn how to use layouts to give the app a consistent look and feel.

Dive Deeper

Before writing your own custom helper, it's wise to spend a few minutes getting familiar with what Rails gives you for free. Check out the helper methods in these modules:

- NumberHelper
- TextHelper
- TagHelper
- DateHelper

The point isn't to challenge yourself to use every single one of these in an app. Rather, it's just good to know what's available should you need it.

You can also search for a helper method in the Rails API documentation.

Layouts

Exercises

Objective

Every good web app has some sort of consistent layout across all the pages: a header at the top, a footer at the bottom, and perhaps even a sidebar next to the main content. What's considered fashionable in terms of shapes, sizes, colors, and other design choices tends to change (usually about the time you launch your app). But when it comes to the layout of your app, consistency never goes out of style.

And to keep things consistent, we need to be able to put all the common layout elements in one definitive place. In Rails, that's called the *layout* file. A layout file is just a regular ERb view template that "wraps" the other view templates in the application. In other words, the layout generates HTML for the header, footer, and sidebar that surrounds the HTML generated by an action's view template. That way, we end up with a complete HTML document to send back to the browser. By having the layout in a single file, we can easily change the layout of our app.

We've gotten this far using the default HTML layout that was generated by Rails. That helped us get an app up and running quickly. Now it's time to start customizing the layout to better suit our app.

This turns out to be a fairly easy exercise. So let's jump right into it!

1. Review the Default Layout

Let's start by reviewing what we get for free...

1. Open the default layout file in `app/views/layouts/application.html.erb` and you should see the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flix</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag "application", "data-turbo-track": "reload" %>
    <%= javascript_importmap_tags %>
  </head>

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

   Notice that the layout file is a mix of HTML and ERb tags, just like a regular view template.

2. To see what it generates, browse to any page in the app and view the generated source. What you'll see is the HTML that was generated by a particular view template, `index` for example, is surrounded by the HTML generated by the layout template. Let's unpack how that happens.

   Back in the layout file, at the top you'll find a few built-in helpers that we haven't seen before:

   - `stylesheet_link_tag` generates the following link tag for the app's stylesheet, which we'll explore in the next section:

     ```
     <link rel="stylesheet" href="/assets/application-e0cf9d8fcb18bf7f909d8d91a5e78499f82ac29523d475bf3a9ab265d5e2b451.css" data-turbo-t
     ```

   - `javascript_importmap_tags` generates the following script tag which imports the JavaScript files used by the app (beyond the scope of this course):

```
<script type="importmap" data-turbo-track="reload">{
  "imports": {
    ...
  }
}</script>
```

- `csrf_meta_tags` generates meta tags for a token that's used internally by Rails to add hidden security fields to forms (don't worry about it):

```
<meta name="csrf-param" content="authenticity_token" />
<meta name="csrf-token" content="A77bbuGOpWTuGPhKBPP8C3WhYR3bKDv3FiJl8vd_Z0NrhgEEzehdhsSOfZoC8EJ1zuCiqOR9QpeRhEnS-f17NQ" />
```

- `csp_meta_tag` generates meta tag for a per-session nonce value for allowing inline `<script>` tags (don't worry about it either).

So far, so good. That's all fairly straight-forward.

Next comes the most non-intuitive part of the layout, which is also the most important part:

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

The `yield` returns the output (HTML) that an action generates. That output is then substituted right into the layout file because the `<%= %>` ERb tag was used. For example, if you visit the `index` page, the `yield` line in the layout file is replaced by the HTML that the `index.html.erb` template generates:

```
<ul>
  <li>
    <strong>Iron Man</strong> (PG-13): $585,366,247
    ...
  </li>
  ...
</ul>
```

So whichever page you visit, the main content changes but the surrounding layout remains consistent. That's the essence of a layout file!

3. Just as a quick test, go ahead and change the contents of the `title` tag to a custom name for your app. Then refresh and you should see the new page title appear at the top of your browser window for every page in the app.

   Answer:

```
<title>Super-Heroic Flix</title>
```

2. Add a Header and Footer

Now let's start customizing. To keep the marketing folks happy, each page needs some marquee header text at the top. And to keep the legal folks happy (a tall order), each page needs a footer at the bottom with a copyright.

1. Inside the `body` tag, start by adding a `header` element above the main content. Inside the header, display the name of the app in an `h1` tag.

   Answer:

```
<header>
  <h1>Flix</h1>
</header>
```

2. Next, put the `yield` line in a `div` with the class of `content`, like so:

```
<div class="content">
  <%= yield %>
</div>
```

3. Finally, add a `footer` element below the main content and inside the footer display a copyright notice and a link back to your company. Want to keep your copyright year up to date? No problem! Just use `Time.now.year` to display the current year.

   Answer:

```
<footer>
  <p>
    Copyright &copy; <%= Time.now.year %>
    <%= link_to "The Pragmatic Studio", "https://pragmaticstudio.com" %>
  </p>
</footer>
```

Here's the final, full layout file:

Answer:

```
<!DOCTYPE html>
<html>
<head>
  <title>Super-Heroic Flix</title>
  ...
</head>
<body>

  <header>
    <h1>Flix</h1>
  </header>

  <div class="content">
    <%= yield %>
  </div>

  <footer>
    <p>
```

```
        Copyright &copy; <%= Time.now.year %>
        <%= link_to "The Pragmatic Studio", "https://pragmaticstudio.com" %>
      </p>
    </footer>

  </body>
  </html>
```

4. Then browse to http://localhost:3000/movies and you should see the header text at the top, the result of calling the `index` action in the middle, and the footer text at the bottom.

And now you have a consistent layout that will get applied to every page of the application! This will become really convenient as we add more pages in upcoming exercises.

Solution

The full solution for this exercise is in the `layouts` directory of the code bundle.

Wrap Up

And that's all there really is to layouts! They're just regular ERb templates that "wrap" the other view templates so your application is displayed in a consistent layout.

Structuring the app around a basic layout is a good start. But once you've been introduced to the layout file, you start thinking about colors, sizes, fonts, and so on. Yup, this is the point where application development crosses over into *web design*. That's a huge topic that's beyond the scope of this course, but in the next section we'll have a quick look at styling the app with CSS and images.

Dive Deeper

The `application.html.erb` layout file is the *application-wide* layout. By default, it's applied to every action in the application. But Rails offers lots of options for rendering a custom layout file depending on the controller, action, or even arbitrary application logic. To learn more about custom layouts, refer to 2.2.13 Finding Layouts in the Rails Guide: Layouts and Rendering.

Stylesheet and Image Assets

Exercises

Objective

Since this isn't a course about web design, we prepared a CSS stylesheet, images, and sample movie data for you. It's always more enjoyable to work on an application that has some style, images, and realistic data. And this gives us an opportunity to learn the basics of how Rails handles assets such as stylesheets and images.

In this exercise we'll copy in the prepared asset files and explore how they're handled. In the end you'll know where to put web design assets in a Rails app, whether you get those assets from a web designer or you apply your own web design skills!

1. Install the Bootstrap Gem

Our design uses the popular Bootstrap framework as the foundation and then adds a some custom CSS on top of that. The bootstrap gem makes it easy to use Bootstrap in Rails, so our first step is to add that gem as a dependency of the `flix` app.

1. First, stop your currently-running `flix` app by typing `CTRL-C` in the console window where it's running.

2. Then add the following line to the `Gemfile` in the `flix` directory:

   ```
   gem 'bootstrap', '~> 5.1', '>= 5.1.3'
   ```

   Remember, all Rails apps have a `Gemfile` that lists all the gems necessary to run the app. It generally doesn't matter where you add gems in the file, though we tend to put them after the primary gems that are listed by default.

3. Now go back to the console window. You should still be in the `flix` directory. Install the gem by typing

   ```
   bundle install
   ```

   You should see output indicating that the `bootstrap` gem was installed, as well as other gems it depends on.

2. Copy Prepared Files

The next step is to copy the prepared files into the appropriate directories of the `flix` app. Feel free to do this however is most comfortable to you on your operating system:

1. First, locate the course code bundle. Inside that bundle you should see a directory named `prepared-files` at the top level. And inside that directory you'll find a directory named `flix`.

2. Copy all the image files contained in the `prepared-files/flix/images` directory into your `flix/app/assets/images` directory.

3. Then copy the `prepared-files/flix/custom.scss` file into your `flix/app/assets/stylesheets` directory.

4. Finally, copy the `prepared-files/flix/seeds.rb` file into your `flix/db` directory, overwriting the existing `seeds.rb` file. Take a peek at this file and you'll see that it uses `Movie.create!` to create a more comprehensive set of movies with more realistic data.

3. Run the Styled App with Seed Data

With the prepared files copied, let's take the app for a spin:

1. First, the development database currently has some movies we created from the Rails `console` in previous exercises. We've included those same movies in the `seeds.rb` file, as well as some newer movies. So we want to clear out the database and populate it with the data in the `seeds.rb` file. To do that, use the following command:

   ```
   rails db:reset
   ```

   That handy command drops and re-creates the development database, applies all the migrations, and runs the `db/seeds.rb` file which creates example movies in the database.

2. Then go ahead and fire up the server:

   ```
   rails s
   ```

3. Now if you browse to http://localhost:3000/movies the page should have a blue header and a black footer. You'll also see more example movies in the main content of the page.

4. Update the Index Template and Layout File

The header and footer are now styled, but the movie listing isn't yet taking advantage of the CSS styles we copied into the app. We'd also like to display the application's logo image in the header.

1. First, to apply the styles to the movie listing, paste the following into your `app/views/movies/index.html.erb` file, replacing what's already in that file:

   ```
   <% @movies.each do |movie| %>
     <section class="movie">
       <div class="summary">
         <h2>
           <%= movie.title %>
         </h2>
         <h3>
           <%= total_gross(movie) %>
         </h3>
         <p>
           <%= truncate(movie.description, length: 150, separator: ' ') %>
         </p>
       </div>
     </section>
   <% end %>
   ```

   Don't worry: All we've done is rearranged things slightly to use different HTML tags with class names that trigger the corresponding CSS rules in our `custom.scss` stylesheet. We also decided not to display the movie ratings and the year the movie was released on this page, but those attributes will make a comeback when we add the movie detail page in the next section.

2. Then display the application's logo image in the header.

   Hint:

   Remember, the header is defined in the layout file: `app/views/layouts/application.html.erb`. Use the `image_tag` helper to display the image file named "logo.png" in the `header` tag in place of the static text.

   Answer:

   ```
   <header>
     <%= image_tag("logo.png") %>
   </header>
   ```

3. Now if you refresh the http://localhost:3000/movies page you should see the logo in the upper-left corner and the listed movies should be styled with a border separating each movie. Voila!

How Stylesheets Are Processed

Just to recap how stylesheets get picked up by Rails, open the `application.html.erb` layout file and narrow in on this line:

```
<%= stylesheet_link_tag "application", "data-turbo-track": "reload" %>
```

The `stylesheet_link_tag` helper generates a link tag for the specified stylesheet file. The `.css` extension is automatically appended. So, in this case, our layout file references a stylesheet named `application.css`.

Where does that stylesheet file live? Well, whenever you reference an *asset* — be it a stylesheet, image, or JavaScript file — Rails actually searches for it in the following three directories:

- `app/assets` contains assets that are specific to this application
- `lib/assets` contains assets for custom libraries that you write and share across your Rails apps
- `vendor/assets` contains third-party assets that are used by this application such as an open-source lightbox or calendaring library

Since `application.css` is a stylesheet that's specific to this app (hence the name), it lives in the `app/assets/stylesheets` directory.

Open the `application.css` file and you should see one big CSS comment that looks something like this (don't worry about the ordering of the lines):

```
/*
 * This is a manifest file...
 *
 *= require_tree .
```

```
 *= require_self
 */
```

If you've ever laid eyes on a CSS file, you'll quickly notice that this doesn't look like your run-of-the-mill stylesheet. It's actually a valid CSS file, but all the lines are in CSS comments. Embedded in those comments are special *directives* that start with an equal sign (=). Rails uses these directives to figure out which stylesheet files should be automatically included when the `application.css` file is requested.

The default `application.css` has two directives, which is all you really need:

```
*= require_tree .
```

The `require_tree` directive includes all files in the specified directory and any subdirectories. Using a dot (`.`) as the directory name means the "current directory", so all CSS files in the `app/assets/stylesheets` directory (and any subdirectories) will be included.

```
*= require_self
```

The `require_self` directive simply includes the contents of the current file (`application.css`). So if we had any CSS below the comments, it would get included.

So the `application.css` file is really just a *manifest*: a list of other files to include in the final `application.css` file. So you can drop stylesheet files in the `app/assets/stylesheets` directory and the `require_tree` directive will make sure they're included. The benefit of this approach is that we can organize styles into different stylesheets, in much the same way we strive to organize our source code into separate files.

It's worth noting that the `bootstrap` gem installs another gem that supports Sass and it's the default pre-processor for files with the `.scss` extenion. So our `custom.scss` file will get automatically pre-processed by Sass to generate valid CSS. But you can always use straight CSS if you prefer.

Solution

The full solution for this exercise is in the `assets` directory of the code bundle.

Bonus Round

If you have some favorite movies that aren't in the movie listing, feel free to add them to the `db/seeds.rb` file. Then to reseed the data, run

```
rails db:reset
```

Wrap Up

It's fun playing around with colors, fonts, rounded corners, drop-shadows, and all the other stuff that makes your app light up in the browser. And Rails makes it really easy to get started and keep things organized. So if you have web design chops, by all means go for it!

On the other hand, if web design isn't exactly your strong suit, then here's our recommendation: when it comes to building your own Rails app, now's the time to get a web designer. Web design is a skill, and like any skill it can take years to master. It's a rare programmer who has invested the time to learn both disciplines well. With what you've just learned about assets, you can meet a designer in the middle. And together you can create an app that works *and* looks great!

Anyway, at this point the movie listing page is looking pretty good. But given that we've truncated the movie description and removed a couple movie attributes from this page, we need another page that shows a movie's full details. And wouldn't you know it, that's our task for the next section.

But first, it's **break time!** Stand up. Stretch. Refresh yourself. Then enjoy a little movie-related humor in the Avengers Gag Reel.

Dive Deeper

Web design is an entire course in itself. If you're new to HTML and CSS, check out the following resources:

- HTML and CSS track on Codecademy
- Learn to Code HTML & CSS by Shay Howe
- HTML & CSS: Design and Build Websites by Jon Duckett

To learn more about the Rails asset pipeline, refer to the Rails Guides: Asset Pipeline

Routes: Show Page

Exercises

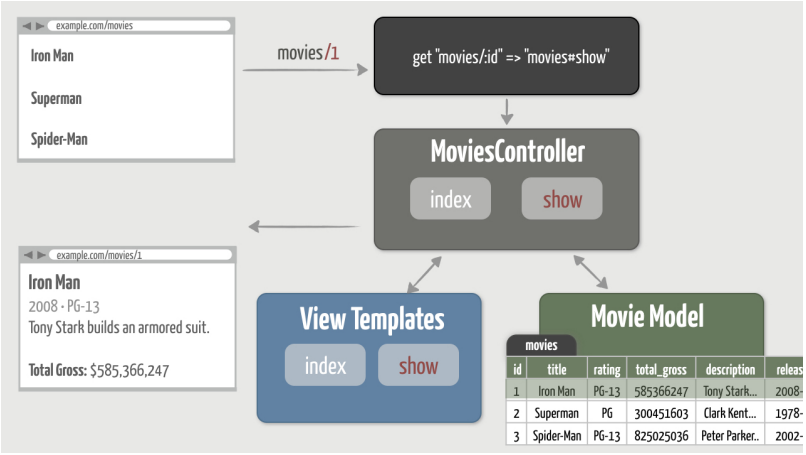
Objective

As our app now stands, we have one page that lists all of the movies in the database. When we visit `/movies`, the `index` action runs and fetches all the movies from the database. Then it renders the `index.html.erb` template which generates an HTML list of movies and sends it back to the browser. That makes for a nice summary page.

In this exercise we'll create the *second* page of our app. When we visit `/movies/1`, for example, we want to see the details for that particular movie. By convention, Rails calls this the "show" page. To make that work, we need to do three things:

1. Add a generic route to handle requests for `/movies/1`, `/movies/2`, `/movies/3`, and so on.
2. Define a `show` action in the `MoviesController` that finds the movie with the id specified in the URL.
3. Create a `show.html.erb` template that generates HTML to display the movie's details.

If you're new to working with MVC, keeping everything in your head can be tricky. So, here's a visual of our objective:



1. Show One Movie

Again, we'll work through this from the outside in, using the error messages to light our path forward.

1. Start by browsing to http://localhost:3000/movies/1 and you should get the following error:

   ```
   Routing Error

   No route matches [GET] "/movies/1"
   ```

2. Remember how to resolve this error? Add a route to the `config/routes.rb` file that maps `GET` requests for `movies/1` to the `show` action of the `movies` controller. Don't worry about supporting other movie ids for now. We'll follow the literal error and work our way back to a more generic solution once we have the basic flow down.

   Answer:

   ```
   Rails.application.routes.draw
     get "movies"   => "movies#index"
     get "movies/1" => "movies#show"
   end
   ```

3. Refresh your browser (you're still accessing http://localhost:3000/movies/1) and this time you should get a different error message:

   ```
   Unknown action

   The action 'show' could not be found for MoviesController
   ```

4. That's your cue—and you know exactly what to do next! Define the `show` action such that it finds the movie in the database that has a primary key (id) of 1 and assigns it to an instance variable named `@movie`.

   Answer:

   ```
   def show
     @movie = Movie.find(1)
   end
   ```

5. Refresh your browser again and—you probably anticipated it— we're missing something:

   ```
   No template for interactive request

   MoviesController#show is missing a template...
   ```

   Remember, unless we tell it otherwise, after running an action Rails will always try to find a view template using a naming convention. In this case, the name of the action is `show`, and it's in the `MoviesController` class, so Rails tries to render the `app/views/movies/show.html.erb` view template file.

6. Create a file named `show.html.erb` in the `app/views/movies` directory. Inside that file, start by simply displaying the movie's title to get confidence that the correct movie is being fetched from the database.

   Answer:

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

   Refresh and you should see the movie title! That tells us we have the model, view, and controller happily connected.

7. Now update the `show.html.erb` template to display all the movie's information. Keep it simple by putting each attribute in a paragraph tag. In terms of helper methods, we don't want to truncate the movie description on this page. However, on this page we do want to display the movie rating and year it was released. (Remember, we removed those movie attributes from the movie listing page.) Since we went to the trouble of creating the `total_gross` and `year_of` view helpers in a previous exercise, we might as well call those helpers here.

8. Once you get a basic show page working, go ahead and copy in the version in the answer. It uses HTML elements and class names that trigger the styles in our `custom.scss` stylesheet.

   Answer:

   ```
   <section class="movie-details">
     <div class="details">
       <h1><%= @movie.title %></h1>
       <h2>
   ```

```
          <%= year_of(@movie) %> &bull; <%= @movie.rating %>
        </h2>
        <p>
          <%= @movie.description %>
        </p>
        <table>
          <tr>
            <th>Total Gross:</th>
            <td><%= total_gross(@movie) %></td>
          </tr>
        </table>
      </div>
    </section>
```

That's a great start!

2. Show Any Movie

Now that we have the MVC flow working, let's make this more generic.

1. Browse to http://localhost:3000/movies/2 and you should get the following error:

   ```
   Routing Error

   No route matches [GET] "/movies/2"
   ```

2. We knew that would happen because we only added a route for `movies/1`, and that route doesn't match this request. At this point we could add another route to handle `movies/2`, but clearly we need to make this more dynamic. The route needs to support a *variable* number of movie ids.

   Update the route to match requests for showing *any* movie.

   Hint:

   You need to replace the hard-coded "1" in the route with a placeholder, or variable, that represents the movie's id. Variables in routes start with a colon (:). In this case, you want a variable named `:id`.

   Answer:

   ```
   Rails.application.routes.draw
     get "movies"     => "movies#index"
     get "movies/:id" => "movies#show"
   end
   ```

3. Now you should be able to browse to any of these URLs without getting any errors:

   - http://localhost:3000/movies/1
   - http://localhost:3000/movies/2
   - http://localhost:3000/movies/3

4. There's just one problem: All of those pages show the details for the first movie! Fix that by updating the `show` action to use the number at the end of the URL to find the movie in the database.

   Hint:

   When the route contains a colon (:) followed by a name, think of it like a placeholder that gets filled in with whatever shows up at that place in the requested URL. For example, given the route "movies/:id", a request for `/movies/7` will fill in the `:id` placeholder with the value 7. And you can get access to that value using `params[:id]` in the `show` action. The `params` hash includes all URL request parameters. And you can access this hash inside of any controller action.

   Answer:

   ```
   def show
     @movie = Movie.find(params[:id])
   end
   ```

5. Now try visiting all three URLs above and you should see the matching movie's details.

Great—now we have our "show" page implemented!

Solution

The full solution for this exercise is in the `show-page` directory of the code bundle.

Wrap Up

This exercise was a good opportunity to take a round trip back through the entire MVC cycle. Now we have two different paths through our application: `/movies` shows all the movies and `/movies/:id` shows the details of any single movie. You probably noticed that adding the second path involved the same high-level steps as the first:

- add a route
- define an action
- create a template

You'll end up following those same three steps over and over again as you develop Rails apps. It's the recipe for accepting requests and generating responses. The details vary depending on how you want the request to be handled, and we'll see more examples of that in future exercises, but you can flesh out a basic flow simply by following the errors as we've done here.

Hey, now that we have two pages, it sure would be nice if we could easily navigate between them! Typing these URLs in the browser's address bar is getting kinda tedious. So in the next section we'll generate hyperlinks so we can easily navigate between pages.

Routes: Linking Pages

Exercises

Objective

The last exercise was a bit of a cliffhanger. We ended up with two different pages, but no way to easily navigate between them. That's not very web-worthy. In this exercise we'll add links so we can navigate between those pages. You know, like the World Wide Web was designed to work!

1. Link to Index Page

First, we want an "All Movies" link in the header that takes us back to the `index` page where all the movies are listed.

To do that, we could just hard-wire a link to `/movies`, like so:

```
<a href="/movies">All Movies</a>
```

That would work, but it makes our application really brittle because we would have link information in two places. First, we have the information in our routes file where we've told it how to handle `/movies`. And secondly, we're explicitly typing out the URL here, and assuming there's a matching route. So, if we decide later to change the URL, we'll need to make changes in *two* places. And that's always a bad sign.

So instead of hard-wiring the URL, we'll ask the router to generate the URL for us. That way the link will *always* have a matching route that points back into our application. Better yet, if we want to change what the URL looks like down the road, we can do that in one place: the routes file.

1. Start by navigating to http://localhost:3000/rails/info/routes to see all the routes we've defined so far.

   You should see a list of routes represented in a table, something like this (we'll ignore the other default routes):

   ```
   Helper        HTTP Verb        Path               Controller#Action
   movies_path     GET        /movies(.:format)      movies#index
                   GET        /movies/:id(.:format)  movies#show
   ```

   What we have here is a handy inventory of the requests that the router recognizes and the matching controller/action pair to handle the request. It's the same thing we have in the `config/routes.rb` file, but it's easier to see here what's actually going on.

   The first row is the route that we want to be matched when we click the "All Movies" link, so let's take a closer look at it:

   ```
   movies_path  GET /movies(.:format)     movies#index
   ```

   - The first column contains the name of the **route helper method** used to generate a URL that's matched (recognized) by this route. Notice the method name includes the plural form `movies` because this route deals with multiple movies.
   - The second column is the **HTTP verb**, which is a `GET` in this case. This is the most common HTTP verb because you typically "get" web pages, but we'll see other verbs a bit later.
   - The third column is the **path**. You can think of it as a pattern that's used to match against the requested URL. Here, the pattern is a literal `/movies`, followed by an optional format in parentheses. (Don't worry about the format.)
   - The last column is the **controller/action pair** that will handle the route. We often say that the request is "mapped" to the controller/action pair.

   So this route will match a `GET` request for the URL `http://localhost:3000/movies`. Remember, the `http://localhost:3000` part of the URL identifies the web server and it then forwards the `/movies` part on to the router. The request then gets dispatched to the `index` action of the `MoviesController`.

2. So, how can this help us? Well, in addition to being able to map requests based on defined routes, the router can also generate URLs that match these routes. And to generate a URL, we simply call the appropriate route helper method. For example, to generate a link to the movie listing page, Rails gives us two possible methods to choose from:

helper method	**generates**
movies_path	/movies
movies_url	http://www.example.com/movies

   Notice that the `_path` method generates the path part of the URL and the `_url` method generates the full URL. In practice you want to use the `_path` variant in view templates and the `_url` variant in controllers as redirects (which we'll learn about later) require fully-formed URLs to be technically valid.

3. So now we know how to generate the URL that leads back to the movie listing page. The next step is to actually generate a hyperlink. And Rails has a helper method for doing that, too. The `link_to` helper is a method that takes two parameters:

   - The first parameter is the **text to link**, such "All Movies".
   - The second parameter is the **destination URL (or path)**, such as the URL path returned from the `movies_path` method.

   So with that in mind, we're ready to add an "All Movies" link in the header. To take advantage of styling, in the `app/views/layouts/application.html.erb` layout file change the `header` to include a `nav` and an empty list of links, like so:

   ```
   <header>
     <nav>
       <%= image_tag("logo.png") %>
       <ul>
         <li>
   ```

```
        # link goes here
      </li>
    </ul>
  </nav>
</header>
```

Then generate an "All Movies" link that points back to the `index` page.

Hint:

Call the `link_to` helper with two parameters: "All Movies" and the result of calling the `movies_path` route helper. Remember that parentheses are optional when calling methods in Ruby, so you can leave off the parentheses if you prefer.

Answer:

```
<%= link_to("All Movies", movies_path) %>
```

4. Then refresh the `show` page and you should see the link. View the page source and you should see that the link points to the same URL we avoided hard-wiring.

```
<a href="/movies">All Movies</a>
```

5. Finally, enjoy the thrill of clicking that link! It's ok if you act totally surprised when you see the movie listing.

2. Link Index Page to Show Page

Next, let's reciprocate that link love. We want each movie title on the `index` page to be a link to the movie's `show` page. This is very similar to what we just did, but there's a slight twist...

1. Check out the list of defined routes again:

```
Helper        HTTP Verb        Path               Controller#Action
movies_path     GET        /movies(.:format)      movies#index
                GET        /movies/:id(.:format)  movies#show
```

This time we want to generate a URL that gets routed to the `show` action, so the second row will be the matching route. But which route helper method should we call to generate the URL? Well, there is no helper in the leftmost column of the route we're interested in. So to get a helper method we need to give the route a name.

2. In the `config/routes.rb` file, give the route for showing a movie the name `movie` using the `as:` option. By convention, the name of the route that maps to the `show` action should be singular because we're dealing with one thing.

Answer:

```
get "movies/:id" => "movies#show", as: "movie"
```

3. Then refresh http://localhost:3000/rails/info/routes and in the leftmost column you should now have helpers for both routes:

```
Helper        HTTP Verb        Path               Controller#Action
movies_path     GET        /movies(.:format)      movies#index
movie_path      GET        /movies/:id(.:format)  movies#show
```

The second route now has a helper named `movie_path`. Be mindful here of the subtle difference between the two helpers in the leftmost column: one includes the plural form `movies` and the other includes the singular form `movie`. It's really easy to get these confused! Just remember that the plural-form helper is used when dealing with multiple movies and the singular-form helper is used when dealing with one specific movie.

4. Now we're ready to actually generate the links to the show page! Keep a couple things in mind:

   - The first parameter to `link_to` will be the movie's title.
   - The second parameter will be the result of calling the `movie_path` helper method. Remember, because the route for that helper has an `:id` placeholder, you must pass the movie's id as a parameter so that the `:id` placeholder will get assigned. As a shortcut, if you instead pass the movie object the router will assume that it should use the object's `id` attribute to fill in the `:id` placeholder.

OK, go ahead and update the `index` template so that the movie title is a link to the movie's `show` page.

Answer:

```
<%= link_to(movie.title, movie_path(movie)) %>
```

5. Then refresh the `index` page and you should see links for each movie title. View the page source and notice that each link points to a distinct movie, like so:

```
<a href="/movies/1">Avengers: Endgame</a>
<a href="/movies/2">Captain Marvel</a>
<a href="/movies/3">Black Panther</a>
```

6. As a small reward for getting that working, here's another shortcut. When generating links to an object's `show` page, you can pass that object as the second parameter to `link_to`, like this:

```
<%= link_to movie.title, movie %>
```

In this particular case, you don't have to call the route helper method. Given the `movie` object, the `link_to` method assumes that it should call the `movie_path` method for you. It's a small bit of syntactic sugar that makes the code more compact and readable.

7. Finally, have you experienced the thrill of clicking any of those links yet? By all means, click away! And be sure to check that all your links work before moving on.

3. Add a Root Route

Now that you're feeling like a routing and linking pro, let's fix something that's been annoying us from the get-go.

If you navigate to http://localhost:3000, you'll see the default Rails welcome page. It would be a lot more useful if browsing to that URL always showed our movie listing, instead of the default welcome page. To do that, we can define something called a *root route*.

1. First, in the `config/routes.rb` file, add a root route that maps to the `index` action of the `MoviesController`.

   Answer:

   ```
   root "movies#index"
   ```

2. Now browse to http://localhost:3000 and you should see the movie listing. Hurray!

3. Next, we might as well change the logo image in the header to link to http://localhost:3000. Do we have a route helper method available to generate that URL? Let's look at the defined routes:

   ```
   Helper        HTTP Verb        Path               Controller#Action
   root_path       GET        /                    movies#index
   movies_path     GET        /movies(.:format)    movies#index
   movie_path      GET        /movies/:id(.:format)  movies#show
   ```

   What's the name of the helper method that generates the URL? Using that helper, change the logo image in the header to point back to the root URL.

   Answer:

   ```
   <%= link_to image_tag("logo.png"), root_path %>
   ```

Solution

The full solution for this exercise is in the `link-pages` directory of the code bundle.

Bonus Round

Displaying Routes

You can also print out a list of all the defined routes by going to a command prompt and typing:

```
rails routes
```

You should see a list like this (we'll ignore the other default routes):

```
Prefix   Verb   URI Pattern             Controller#Action
movies   GET    /movies(.:format)       movies#index
movie    GET    /movies/:id(.:format)   movies#show
```

The output is slightly different than listing the routes by navigating to http://localhost:3000/rails/info/routes. Instead of the leftmost column containing the name of a helper method, it just shows a prefix. For example, the route in the first row has the prefix `movies`. The trick to deciphering this is knowing that the name of the helper method is derived from the prefix of the route. In this case the route prefix is `movies`, and therefore the helper methods are named `movies_path` and `movies_url`.

Trying Route Helpers in the Console

You can experiment with route helper methods in a Rails `console` session using the special `app` object. For example, here's how to try the `movies_path` route helper from inside a Rails `console` session:

```
>> app.movies_path
=> "/movies"
```

Notice it generates the path part of the URL.

To generate the full URL, you need to use the `movies_url` helper, like so:

```
>> app.movies_url
=> "http://www.example.com/movies"
```

To generate the path to a specific event, you use the `movie_path` helper and either pass it a movie's id or (as a shortcut) just the movie:

```
>> movie = Movie.find(1)

>> app.movie_path(movie.id)
=> "/movies/1"

>> app.movie_path(movie)
=> "/movies/1"
```

And to generate a link to the "root" page, you use the `root_path` helper:

```
>> app.root_path
=> "/"
```

Pop Quiz

Sharpen your pencil, it's time for a pop quiz! Let's say that you are building a Rails app with books for sale, and that your routes so far look like this:

```
books_path  GET /books(.:format)      books#index
            GET /books/:id(.:format)  books#show
```

1. What URL will calling the `books_path` method generate?

   Answer:

The `_path` route method generates a relative path, in this case `/books`

2. What URL will calling the `books_url` method generate?

   Answer:

   The `_url` route method generates the full URL, in this case `http://www.example.com/books`

3. The `link_to` helper method takes 2 parameters:
   the name of the _____, and the _____.

   Answer:

   The `link_to` helper method takes 2 parameters: the **text to link (such as "All Books")**, and the **destination URL (or path)**.

4. How would you link the words "See All Books" to the index page of the books app?

   Answer:

   ```
   <%= link_to("See All Books", books_path) %>
   ```

   or

   ```
   <%= link_to "See All Books", books_path %>
   ```

5. By convention, the name of the index route is `books`. What would you name the route for showing a particular book?

   Answer:

   The name of the route would be `book` because, by convention, the name of the route that maps to the `show` action should be singular since we're dealing with one thing.

6. How would you link a book's title to the book's show page? There are 2 possible answers.

   Answer:

   ```
   <%= link_to(book.title, book_path(book)) %>
   ```

   or

   ```
   <%= link_to(book.title, book) %>
   ```

   You don't have to call the route helper method. Given the book object, the `link_to` method assumes that it should call the `_path` method for you. But remember, this shortcut only works for generating links to an object's show page.

7. How would you add a root route that maps `http://localhost:3000` to the index page?

   Answer:

   ```
   root "books#index"
   ```

8. Lastly, with our root route in place, how would you change the "See All Books" link on the show page to now point to the root URL.

   Answer:

   ```
   <%= link_to("See All Books", root_path) %>
   ```

Nicely done! Routes can be a little tricky to wrap your head around, so have patience with yourself as you work with them. In time, they'll become as familiar as the route to your favorite coffee shop. (We never promised our jokes would be funny.)

Wrap Up

The router is a super powerful part of Rails. It also tends to be one of the most challenging, particularly because there are so many conventions at work behind the scenes. When you're in doubt, the best thing to do is follow the routes. Look at the defined routes to see what's going on, identify the name of the route you're interested in, and then play with the route helper methods in the `console` until you're confident. Oh, you think that's what we tell all the beginners? Actually, no. That's exactly how we debug routing problems!

Now that we have a full MVC stack with multiple pages linked together, in the next section we'll start adding pages with web forms. You know, so we can edit and create movies from the comfort of our browser!

Forms: Editing Records Part 1

Exercises

Objective

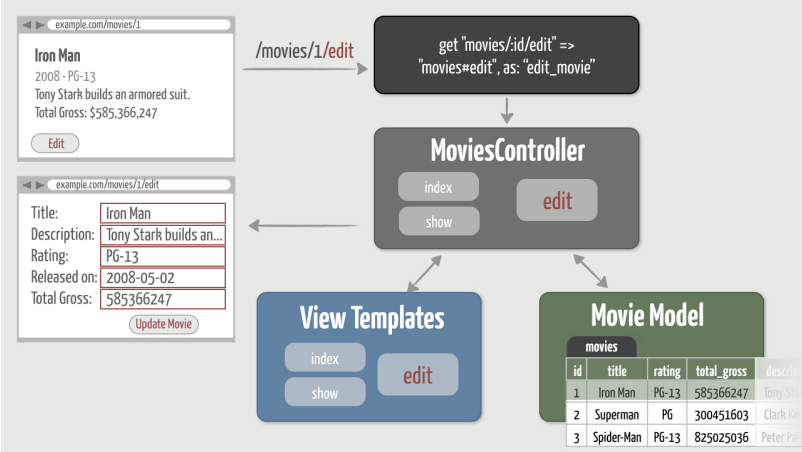
At this point we have some fundamental features under our belt and our movies app is shaping up nicely. The movies are neatly tucked away in a database and we can list and show them in the browser. That's a good start!

Now, what if we want to change a movie's details? The only way to do that currently is to fire up a `console` session and programmatically change movie attributes. But this is supposed to be a *web app*! So the next step is to create a web interface for editing (changing) movie details. We'll tackle this in two parts, similar to the way we did it in the video.

First we need to display a web form so a user can edit a movie's details. Doing that involves the following high-level tasks:

1. Add a route to handle requests for `/movies/1/edit`, for example.
2. Generate an "Edit" link on the `show` page.
3. Define an `edit` action in the `MoviesController` that finds the movie we want to update and displays an HTML form.
4. Create an `edit.html.erb` view template that generates the HTML form pre-populated with the movie's details.

Visually, here's what we want to do:



Then, in the second part, we'll need to define an `update` action in the `MoviesController` that saves movie changes to the database when the form is submitted.

We have our work cut out for us, so let's get started!

1. Add an Edit Route and Link

Let's start with the URL we want and work our way from the outside-in. By convention in Rails, the URL for editing a movie would be `http://localhost:3000/movies/1/edit`, for example. To make it easy to navigate there, we'll generate an "Edit" link on the `show` page.

1. As a jumping-off point, browse to http://localhost:3000/movies/1/edit and you should get the following error:

   ```
   Routing Error

   No route matches [GET] "/movies/1/edit"
   ```

2. Fix it by adding a route that sends requests for `movies/1/edit`, for example, to the `edit` action of the `MoviesController`. Make the route flexible enough to recognize any arbitrary movie id (not just 1). We'll need to be able to generate an "Edit" link, so go ahead and name the route `edit_movie`.

   Hint:

   Remember, to name a route you use the `as:` option.

   Answer:

   ```
   get "movies/:id/edit" => "movies#edit", as: "edit_movie"
   ```

3. Now list the defined routes and you should have the following four routes, with the new route listed last:

   ```
   Helper          HTTP Verb        Path                  Controller#Action
   root_path          GET        /
   movies_path        GET        /movies(.:format)           movies#index
   movie_path         GET        /movies/:id(.:format)       movies#show
   edit_movie_path    GET        /movies/:id/edit(.:format)  movies#edit
   ```

4. Great! Now use the route helper method to generate an "Edit" link at the bottom of the `show` template. Once you get the link working, go ahead and copy in the version in the answer that uses HTML elements and class names that trigger the styles in our `custom.scss` stylesheet.

   Answer:

   ```
   <section class="admin">
     <%= link_to "Edit", edit_movie_path(@movie), class: "button" %>
   </section>
   ```

5. Finally, hop back into your browser, navigate to a movie's show page, and click the newly-generated "Edit" link. You should get the following error:

   ```
   Unknown action

   The action 'edit' could not be found for MoviesController
   ```

   Clearly we need to implement the `edit` action next.

   Before moving to the next step, think about how you'll do it! It's good to start internalizing the high-level steps before jumping right to the code.

2. Create the Edit Action and Form

Did you figure it out? The `edit` action needs to do two things. First, similar to the `show` action, the `edit` action needs to use the ID embedded in the URL to find the appropriate movie in the database. Then the action needs to render a form and populate the form fields with the respective movie's attributes.

1. First up, define an `edit` action in the `MoviesController` that finds the requested movie and assigns it to an instance variable named `@movie`.

   Hint:

   Remember that because the route has a variable named `:id`, you can get the movie id embedded in the URL by accessing `params[:id]` from within the `edit` action.

   Answer:

   ```
   def edit
     @movie = Movie.find(params[:id])
   end
   ```

2. Refresh the page (you're still accessing http://localhost:3000/movies/1/edit) and you should totally expect this error:

   ```
   No template for interactive request

   MoviesController#edit is missing a template...
   ```

3. Following the error, create a file named `edit.html.erb` in the `app/views/movies` directory. Inside that file, start by simply displaying the movie's title as a quick sanity check that the correct movie is being fetched from the database.

   Answer:

   ```
   <h1>Editing <%= @movie.title %></h1>
   ```

   Refresh and you should see the movie title. So far, so good!

4. Now that we have a movie object in the template, let's start incrementally working on the form. First, update the `edit.html.erb` template to generate a form using the `form_with` helper method. For now, just add a label and a text field for the movie's `title` attribute.

   Hint:

   Remember that the `form_with` helper needs to be called with the `model:` option set to the `@movie` object. Think of it as "binding" the form to the movie. Then, inside the `form_with` block, you can use the block parameter (typically named `f`) which is a form builder to call helpers that generate HTML form elements. For example, `f.text_field :title` generates a text field for the movie's `title` attribute.

   Answer:

   ```
   <%= form_with(model: @movie) do |f| %>
     <%= f.label :title %>
     <%= f.text_field :title %>
   <% end %>
   ```

   Refresh and you should see a label and a text field populated with the the movie's title.

5. Then incrementally add form elements so you can edit a movie's description, rating, released on date, and total gross. You'll need to generate the appropriate HTML form element depending on the type of attribute:

   - `description` (a text attribute) goes in a text area with 7 rows, for example
   - `rating` (a string attribute) goes in a text field
   - `released_on` (a date attribute) goes in a date select with the HTML option `class` set to `"date"`.
   - `total_gross` (a decimal attribute) goes in a number field

   As you work through these you might find it helpful to use the documentation at http://api.rubyonrails.org/. For example, search for "text_field" and "date_select" to find the corresponding form helpers.

   Answer:

   ```
   <%= form_with(model: @movie) do |f| %>
     <%= f.label :title %>
     <%= f.text_field :title %>

     <%= f.label :description %>
     <%= f.text_area :description, rows: 7 %>

     <%= f.label :rating %>
     <%= f.text_field :rating %>

     <%= f.label :released_on %>
     <%= f.date_select :released_on, {}, {class: "date"} %>

     <%= f.label :total_gross %>
     <%= f.number_field :total_gross %>
   <% end %>
   ```

6. When you're done, refresh the form and all the fields should contain the values for the attributes in the `@movie` object. In other words, the values in the form should reflect the movie details that are in the database.

7. Finally, add a submit button at the bottom of the form:

   Answer:

   ```
   <%= form_with(model: @movie) do |f| %>
     <%= f.label :title %>
     <%= f.text_field :title %>

     <%= f.label :description %>
   ```

```
    <%= f.text_area :description, rows: 7 %>

    <%= f.label :rating %>
    <%= f.text_field :rating %>

    <%= f.label :released_on %>
    <%= f.date_select :released_on, {}, {class: "date"} %>

    <%= f.label :total_gross %>
    <%= f.number_field :total_gross %>

    <%= f.submit %>
  <% end %>
```

Refresh and you should see an "Update Movie" button. Wait a minute! How did Rails know that the submit button should say "Update Movie"? It figured that out because the `@movie` object you passed to `form_with` is an existing record in the database, so you must be trying to update it. Pretty clever, eh?

8. **Bonus:** As an alternative to using the `date_select` form helper, you can instead use the `date_field` helper. This helper creates an input of type "date", which in the Chrome browser will show a little triangle next to the input box. Click the triangle, and a calendar pops up so you can easily pick a date! Note that not all browsers support the HTML 5 "date" type.

   Answer:

   ```
   <%= f.date_field :released_on %>
   ```

It's tempting to want to submit the form, but that's the challenge of the next exercise. So if you're seeing a form populated with a movie's data, then you're good to go for this exercise!

Solution

The full solution for this exercise is in the `forms-edit-1` directory of the code bundle.

Wrap Up

Excellent! Now we're showing a form for changing a movie's details. Next we need to handle what happens when a user submits the form...

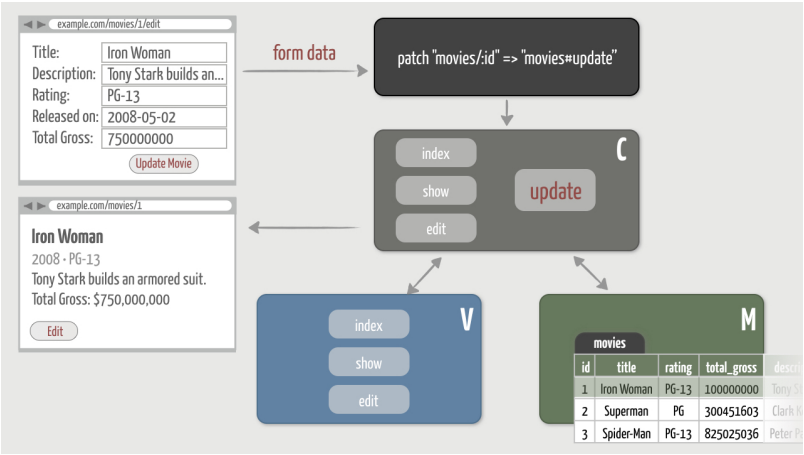
Forms: Editing Records Part 2

Exercises

Objective

Remember, the way forms work is a two-part story. The first part is all about displaying the form. The second part is about what happens when you enter information in the form and submit it. The form data needs to get sent back into the app so it can be saved in the database. To handle that, our app needs another route and a corresponding action. By convention, that action is called `update` because we're updating an existing record in the database. Once the movie has been updated, we'll redirect the browser to the movie's show page.

If you like diagrams (can you tell we do?), here's our goal:



Let's jump right into it!

1. Add a Route

Let's try submitting the edit form and see where it leads us...

1. Submit the form by clicking the "Update Movie" button and, well, it will appear as if nothing happened! But if you look in the Terminal or command prompt window where the Rails server is running, you should see the following error (you may need to scroll back a bit):

   ```
   ActionController::RoutingError (No route matches [PATCH] "/movies/1")
   ```

   Notice that this is a `PATCH` request rather than a `GET` request as we've been seeing up to this point. In HTTP, `GET` requests are used to *read* things from the server. `PATCH` requests, by comparison, are used to *update* (write) things in the server. And Rails knows that we're trying to update something.

2. The error clearly indicates that submitting the form will send the form data in a `PATCH` request to `/movies/1`. Let's recheck which routes we already have:

```
Helper          HTTP Verb        Path                   Controller#Action
root_path          GET        /
movies_path        GET        /movies(.:format)           movies#index
movie_path         GET        /movies/:id(.:format)       movies#show
edit_movie_path    GET        /movies/:id/edit(.:format)  movies#edit
```

We have a route that recognizes `/movies/1`, but it expects the HTTP verb to be `GET`, not `PATCH`. It's important to remember that the router matches requests based on the unique combination of both the HTTP verb and the URL. The same URL can do something different based on the HTTP verb. That being the case, we're missing a route.

3. Add a route that sends `PATCH` requests for `movies/1`, for example, to the `update` action of the `MoviesController`. Don't worry about giving the route a name because we don't need to generate a link for it. (The `form_with` helper will handle that for us.)

Answer:

```
patch "movies/:id" => "movies#update"
```

Listing the defined routes should then give you five total routes (we'll ignore the other default routes):

```
Helper          HTTP Verb        Path                   Controller#Action
root_path          GET        /
movies_path        GET        /movies(.:format)           movies#index
movie_path         GET        /movies/:id(.:format)       movies#show
edit_movie_path    GET        /movies/:id/edit(.:format)  movies#edit
                   PATCH      /movies/:id(.:format)       movies#update
```

4. Next, back over in your the browser, click the "Update Movie" button again to submit the form. Again, nothing appears to happen. But if you check in the Terminal or command prompt window where the Rails server is running (you may need to scroll back), you should see:

```
AbstractController::ActionNotFound (The action 'update' could not be found for MoviesController)
```

2. Implement the Update Action

Taking that subtle hint, we need to define the `update` action in the `MoviesController`. The `update` action needs to pick up the submitted form data from the request in order to actually update the movie in the database.

1. Just to see the form data that's being submitted, define the `update` action and try this `fail` trick we used in the video:

```
def update
  fail
end
```

The `fail` method in Ruby intentionally raises an exception. It's often used as a poor (lazy) person's debugger. When `fail` is called in Rails app, execution is halted, Rails intercepts the exception, and spills its guts on an error page in the browser. In addition to all the request parameters, the error page also displays any submitted form data.

2. Now submit the form with data again and you'll get an error (or a debugging page, depending on how you look at it). Check out the stuff under the "Request" heading and you should see something like this:

```
{"_method"=>"patch",
 "authenticity_token"=>"[FILTERED]",
 "movie"=>
   {"title"=>"Iron Man",
    "description"=>"When wealthy industrialist Tony Stark...",
    "rating"=>"PG-13",
    "released_on(1i)"=>"2008",
    "released_on(2i)"=>"5",
    "released_on(3i)"=>"2",
    "total_gross"=>"585366247.0"},
 "commit"=>"Update Movie",
 "id"=>"1"}
```

What we're seeing here is a representation of a Ruby hash: a set of keys and values. The highlighted lines are what we're most interested in. Notice that the hash includes an `id` key whose value is the id of the movie we want to update. The hash also includes a `movie` key. Here's where things get interesting. The value of the `movie` key is actually another (nested) hash. And *that* hash contains the form data: the movie attribute keys and values we want to change. In other words, the form data is scoped to `movie` in the request parameters hash. So in the request parameters hash we have all the information we need to actually update the movie in the database!

Now, how exactly do we get access to this information in our `update` action? When Rails receives a request—be it a `GET`, `PATCH`, or whatever—it automatically packages the request parameters in a hash called `params` that's accessible in an action.

Inside the `update` action, to get the movie id we use:

```
params[:id]
```

That returns 1 in this case. Note that in the original hash the key is the string `"id"`, but since we're using the `params` hash we can use the symbol `:id`. The `params` hash lets us look up values using either string or symbol keys, but it's more idiomatic to use symbols.

In the same way, to get the names and values of the movie attributes submitted from the form, we use:

```
params[:movie]
```

That returns a hash of movie attribute keys and values:

```
{
  "title"=>"Iron Man",
  "description"=>"When wealthy industrialist Tony Stark...",
```

```
  "rating"=>"PG-13",
  "released_on(1i)"=>"2008",
  "released_on(2i)"=>"5",
  "released_on(3i)"=>"2",
  "total_gross"=>"585366247.0"
}
```

Handy, dandy!

3. Now that you know how to pull out the form data from the request parameters, remove the `fail` line from the `update` action and instead do two things. First, find the movie being updated and assign it to an instance variable called `@movie`. Then use the submitted form data to update the movie's attributes and save it to the database.

   Hint:

   First you'll need to fetch the movie to be updated from the database using the value in `params[:id]`. Then, to update its attributes with the form data, recall that all models that inherit from `ApplicationRecord` have an `update` method. That method takes a hash of attribute names and values. Conveniently, accessing `params[:movie]` gives you the form data represented as a hash of movie attribute names and values. Combine those two things, and you can use the form data to update the movie and save it back to the database in one line of Ruby!

   Answer:

   ```
   def update
     @movie = Movie.find(params[:id])

     @movie.update(params[:movie])
   end
   ```

4. Now, back in your browser, use the form to edit some details about your first movie. For example, you may want to rename "Iron Man" to "Iron Woman" and set its total gross to $1B!

5. Then submit the form again and this time you should get a different error:

   ```
   ActiveModel::ForbiddenAttributesError
   ```

   Forbidden! What happened was we took all the movie attributes in the form data and tried to assign them in bulk (using `update`) to the movie being updated. Rails grabs us by the collar and says: "You sure about that? Do you really want to trust parameters from the big, bad Internet?" It's actually quite trivial for a malicious user (a hacker) to fake form data and end up changing movie attributes that we don't want them to change.

   To prevent that from happening, Rails requires us to explicitly list the attributes that can be mass assigned from form data. We do that by calling the `permit` method and passing in the list of attributes that are allowed to be mass-assigned from form data, like so:

   ```
   params[:movie].permit(:title, :description, :rating, :released_on, :total_gross)
   ```

   That line of code returns a new hash that includes only the permitted attributes. It also marks the hash as being "permitted" which tells Rails that we've taken the necessary security precautions. In other words, we now have a *white list* of trusted attributes that can safely be sent through the `update` method.

   Alternatively, if we want to lock this down a little tighter, we can call the `require` method on the `params` hash and pass it the parameter key we expect, like so:

   ```
   params.require(:movie).permit(:title, :description, :rating, :released_on, :total_gross)
   ```

   The `require` method returns the same hash as we get by accessing `params[:movie]`, but `require` will raise an exception if the given key (`:movie` in this case) isn't found in the `params` object. So using `require` is preferred because it gives us one more added check on the form data.

   Finally, if you want all of a movie's attributes to be updatable without listing them out, you can call `permit!` like so

   ```
   params.require(:movie).permit!
   ```

   While convenient, using `permit!` is risky because *all* the attributes will *always* be updatable from form data. Instead, it's better to explicitly list the attributes that can be updated from a form.

6. In your `update` action, use the line of code above and assign the result to a variable called `movie_params`, for example. Then pass that variable as a parameter to the `update` method.

   Answer:

   ```
   def update
     @movie = Movie.find(params[:id])

     movie_params =
       params.require(:movie).
         permit(:title, :description, :rating, :released_on, :total_gross)

     @movie.update(movie_params)
   end
   ```

7. Then submit the form with data *again* and Rails shouldn't hassle you about the perils of form data on the big, bad web. However, this time the button should become disabled and it will appear as if nothing happened. But something did indeed happen!

   If you have a look in the Terminal or command prompt window where the Rails server is running, you should see something like the following:

   ```
   Started PATCH "/movies/1" for ::1 at YYYY-07-08 13:42:05 -0600
   Processing by MoviesController#update as TURBO_STREAM
   Parameters: {...}
   Movie Load (0.2ms)  SELECT "movies".* FROM "movies" WHERE "movies"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
   (0.1ms)  begin transaction
   Movie Update (0.3ms)  UPDATE "movies" SET "title" = ?, "total_gross" = ? WHERE "movies"."id" = ?  [["title", "Iron Woman"], ["total_gross
   (0.5ms)  commit transaction
   No template found for MoviesController#update, rendering head :no_content
   ```

That output tells you exactly what happened, right down to the SQL UPDATE statement that was used to update the event in the database. (You'll only see the update if you actually change movie attributes in the form.)

Great! This means the `update` action successfully update the movie details in the database based on the form data. After doing that, however, Rails went looking for a matching `update` view template, which we don't have.

We'll need to figure out what to show in response to the movie being updated, but let's ignore that error for a minute. First let's verify that the movie was indeed updated in the database. Hop into a `console` session, find the movie you updated, and then check that each attribute has the value you set in the form.

Answer:

```
>> movie = Movie.find(1)

>> movie.title
=> "Iron Woman"

>> movie.total_gross.to_s
=> "1000000000.0"
```

A billion dollars?! Now *that's* a hit superhero movie!

3. Redirect to the Show Page

OK, so what should we do after the movie has been successfully updated? Well, it stands to reason that the user would want to see the updated movie's details. And lo and behold, we already have a `show` action that does that. So let's just *redirect* the browser back to that URL. That will force the browser to send a new request into our app. The new request will be a `GET` for `/movies/1`, for example.

In the `update` action, after the movie has been updated, redirect to the movie's `show` page.

Hint:

Use the `redirect_to` helper and pass the URL to the `show` page as a parameter. Remember, you already have a route helper method that will generate the URL to the `show` page. By using a redirect, the action won't try to render an `update` view template, which we don't have. Take that, browser!

Answer:

```
def update
  @movie = Movie.find(params[:id])

  movie_params =
    params.require(:movie).
      permit(:title, :description, :rating, :released_on, :total_gross)

  @movie.update(movie_params)

  redirect_to @movie
end
```

Then submit the form one last time! This time the movie should get updated in the database and you should get redirected to the `show` page where those changes are reflected.

And with that, our two-part, edit-update story comes to an end. Nicely done!

Solution

The full solution for this exercise is in the `forms-edit-2` directory of the code bundle.

Bonus Round

Where Did `PATCH` Come From?

Folks often ask how Rails knows to use the `PATCH` verb when updating data. The answer lies in the HTML that was generated by the `form_with` helper. View the page source of the form and hone in on this part of the HTML form:

```
<form action="/movies/1" method="post">
  <input type="hidden" name="_method" value="patch" />
  ...
</form>
```

The relevant parts here are the `action` and the `method`. The `action` says that submitting this form will send the data to `/movies/1`. But remember that the router recognizes requests based on both a URL pattern to match *and* an HTTP verb. So the `method` attribute in the form specifies the HTTP verb to use. By default, forms are submitted with the `POST` HTTP verb as indicated by the value of the `method` attribute here.

Ah, but the plot thickens: By convention in Rails, `POST` requests are used to create things. But in this case, Rails knows that we're trying to *update* a movie, not create a movie. And by convention in Rails, `PATCH` requests are used to update things. So Rails effectively overrides the HTTP verb by adding a hidden field named `_method` with a value of `patch`. (It has to fake it with a hidden field because web browsers can't natively send `PATCH` requests.)

Anyway, that's where the `PATCH` comes from. Do you need to understand all that before moving on? Nope, not at all. Sometimes it's just comforting to know what's going on behind the curtain.

What Does a Redirect Do?

It's important to understand that when we submit a form, our application actually ends up handling *two* requests: a `PATCH` request and a `GET` request. Let's take a minute to follow those requests in the server log when you submit the form.

The first request you see is the `PATCH` that's handled by the `update` action:

```
Started PATCH "/movies/1" for ::1 at YYYY-07-08 13:51:46 -0600
Processing by MoviesController#update as TURBO_STREAM
Parameters: {...}
Movie Load (0.1ms)  SELECT "movies".* FROM "movies" WHERE "movies"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
(0.0ms)  begin transaction
Movie Update (0.3ms)  UPDATE "movies" SET "title" = ?, "updated_at" = ? WHERE "movies"."id" = ?  [["title", "Iron Man"], ["updated_at", "YYYY-
(0.7ms)  commit transaction
Redirected to http://localhost:3000/movies/1
Completed 302 Found in 6ms (ActiveRecord: 1.1ms | Allocations: 3734)
```

Check out what happens at the end of the request, on the second to the last line. It tells the browser to redirect to `http://localhost:3000/movies/1`. It does that by sending the browser a URL and an HTTP redirect status code (302) as seen on the last line. It's like telling the browser: "Nothing to see here, move along. Go this way instead."

So the browser issues a brand new `GET` request for the updated movie, and we see the second request come into our app:

```
Started GET "/movies/1" for ::1 at YYYY-07-08 13:51:46 -0600
Processing by MoviesController#show as TURBO_STREAM
Parameters: {"id"=>"1"}
Movie Load (0.1ms)  SELECT "movies".* FROM "movies" WHERE "movies"."id" = ? LIMIT ?  [["id", 7], ["LIMIT", 1]]
Rendering movies/show.html.erb within layouts/application
Rendered movies/show.html.erb within layouts/application (Duration: 0.3ms | Allocations: 201)
Completed 200 OK in 24ms (Views: 22.6ms | ActiveRecord: 0.1ms | Allocations: 15221)
```

The result of that request is to render the `show` view template. After doing that, the app sends the generated HTML back to the browser with a 200 HTTP status code which indicates the request was successful.

By following the PATCH-Redirect-GET pattern, we prevent duplicate form submissions, and thereby create a more intuitive interface for our users.

Spend some time watching the server log, and all mysteries will be revealed!

Wrap Up

Congratulations! This was a big step that put together a whole bunch of concepts we learned earlier, and a few new ones. So make sure to reward yourself with a well-deserved break and a tasty snack!

Here's the thing about forms in Rails: They're actually quite elegant once you get comfortable with the conventions. The conventions really do save you time once you grok what's going on, but initially they can feel overwhelming. It's best to start with a simple form and use the errors (and server log) to trace through what's happening at each step. By about your second or third form, everything will click. It just takes some practice to get the hang of it. And that's what this exercise was all about: practice!

Speaking of practicing with forms, in the next section we'll bring all this together again so we can create new movies in the web interface. That'll give us a good opportunity to cement what we've learned about forms and their conventions.

Dive Deeper

For more options on permitting mass-assigned parameters, check out strong parameters.

Forms: Creating Records

Exercises

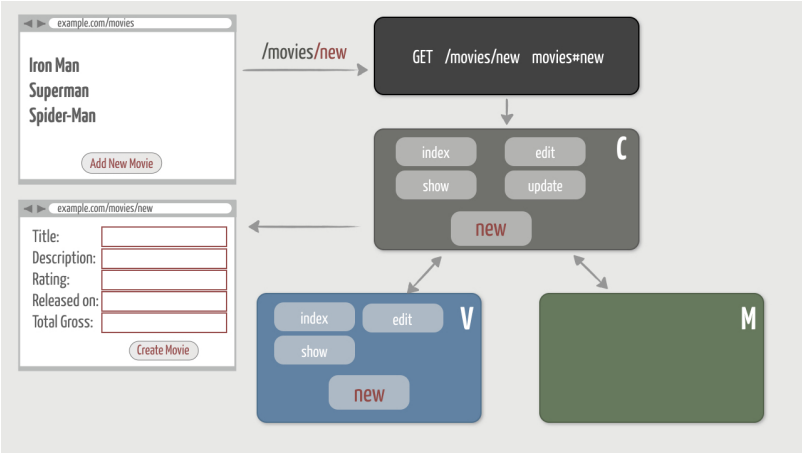
Objective

At this point we can edit movies in the web interface, but we have no way to create new movies outside of the `console`. That hardly makes for a useful web app, so let's fix that!

It turns out there's a lot of symmetry between editing movies and creating movies, so we'll follow similar steps:

1. Add a route to handle requests for `/movies/new` to show a form.
2. Generate an "Add New Movie" link on the `index` page.
3. Define a `new` action in the `MoviesController` that renders a form template.
4. Create a `new.html.erb` template that generates an HTML form with blank fields
5. Define a `create` action in the `MoviesController` that inserts the movie into the database when the form is submitted

Visually, here's what we want to do first:



The good news is there's honestly not much new to learn. We already have a form, and we've learned how to process form data in an action. So we'll pick up the pace a bit...

1. Add a Route and Link

As usual, let's start at a high level with the URL. By convention in Rails, the URL for creating a new movie would be `http://localhost:3000/movies/new`. And that request would get sent to a `new` action because we want to show the form for creating a *new* movie. We'll put an "Add New Movie" link on the `index` page to make it easy to get to the form.

1. Browse to http://localhost:3000/movies/new and you'll get this **brand spanking new error**:

   ```
   ActiveRecord::RecordNotFound in MoviesController#show

   Couldn't find Movie with 'id'=new
   ```

   Whoa! That's not what we expected. The router recognizes the request and sends it to the `show` action. What's up with that?

2. In these cases, your go-to debugging tool is http://localhost:3000/rails/info/routes. It never lies. You'll see that we currently have these five routes defined:

   ```
   Helper          HTTP Verb        Path                   Controller#Action
   root_path          GET        /
   movies_path        GET        /movies(.:format)           movies#index
   movie_path         GET        /movies/:id(.:format)       movies#show
   edit_movie_path    GET        /movies/:id/edit(.:format)  movies#edit
                      PATCH      /movies/:id(.:format)       movies#update
   ```

   According to those routes, a `GET` request for `movies/new` would map to the `show` action in the `MoviesController`. Why? Well, it thinks the word "new" should be used to fill in the `:id` placeholder.

3. So our first step would be to define a new route that matches the literal `movies/new` and sends it to the `new` action. Because routes are evaluated top to bottom for a match, we'd need to put the `movies/new` route before the `movies/:id` route, like so:

   ```
   Rails.application.routes.draw
     root   "movies#index"
     get    "movies"      => "movies#index"
     get    "movies/new" => "movies#new"
     get    "movies/:id" => "movies#show", as: "movie"
     get    "movies/:id/edit" => "movies#edit", as: "edit_movie"
     patch  "movies/:id" => "movies#update"
   end
   ```

   But defining these routes one-by-one is getting kinda tedious. Rails to the rescue! It's common to want routes for all these scenarios, plus a few more, on other entities in your application. Rails calls these entities *resources*. And to eliminate some of the grunt work, Rails has a handy convention for defining all the routes a resource might need in one fell swoop.

4. Open the `config/routes.rb` file and replace all the routes we defined previously (except the *root route*) with the highlighted line below:

   ```
   Rails.application.routes.draw
     root "movies#index"

     resources :movies
   end
   ```

5. Then reload http://localhost:3000/rails/info/routes to see what that does for us. You should see **nine** total routes: the **five** routes we had before plus **four** new routes:

   ```
   Helper          HTTP Verb        Path                   Controller#Action
   root_path          GET        /
   movies_path        GET        /movies(.:format)           movies#index
                      POST       /movies(.:format)           movies#create
   new_movie_path     GET        /movies/new(.:format)       movies#new
   edit_movie_path    GET        /movies/:id/edit(.:format)  movies#edit
   movie_path         GET        /movies/:id(.:format)       movies#show
                      PUT        /movies/:id(.:format)       movies#update
                      PATCH      /movies/:id(.:format)       movies#update
                      DELETE     /movies/:id(.:format)       movies#destroy
   ```

   Magic? No, the `resources :movies` line simply used the built-in Rails conventions to dynamically define all those routes. And because we used the Rails conventions when defining the routes manually, we end up with the same routes plus a couple extras. Even the route names are the same as the names we picked.

And that means our application should work exactly as it did before! It's almost like we planned it that way...

So we don't need to define any new routes after all. Notice that routes for `GET` requests to `/movies/new` and `POST` requests to `/movies` are already defined. And that's exactly what the doctor ordered for this task!

6. Given those routes, use a route helper method to generate an "Add New Movie" link at the bottom of the `index` page. Once you get the link working, go ahead and copy in the version in the answer that uses HTML elements and class names that trigger the styles in our `custom.scss` stylesheet.

   Answer:

   ```
   <section class="admin">
     <%= link_to "Add New Movie", new_movie_path, class: "button" %>
   </section>
   ```

7. Back in your browser, navigate to the movie listing page and click the newly-generated "Add New Movie" link. You should get the following error:

   ```
   Unknown action

   The action 'new' could not be found for MoviesController
   ```

   Now we're off to the races. Think you know the steps to get us to the finish line? Give it some thought before moving on to the implementation in the next step.

2. Create the New Action and Form

The `new` action needs to render a form so we can enter a new movie's information. This is similar to the `edit` action, but in this case we don't have an existing movie in the database. So the `new` action needs to instantiate a new `Movie` object in memory, so that the form can "bind" to that object.

1. Define a `new` action in the `MoviesController` that instantiates a new (empty) movie object and assigns it to an instance variable named `@movie`.

   Answer:

   ```
   def new
     @movie = Movie.new
   end
   ```

2. Refresh the page (you're still accessing http://localhost:3000/movies/new) and by now you should be so comfortable with this error that resolving it is as easy as falling off a log:

   ```
   No template for interactive request

   MoviesController#new is missing a template...
   ```

3. Create a file named `new.html.erb` in the `app/views/movies` directory. Inside that file, just add enough to get some text on the page:

   ```
   <h1>Create a New Movie</h1>
   ```

   Refresh as a blazing-fast smoke test.

4. Finally, we need to generate a form in the `new.html.erb` template that uses the `@movie` object. It turns out that form would be *exactly* the same as the form we already have in the `edit.html.erb` template. So for now, copy the form that's in `edit.html.erb` and paste it into the `new.html.erb` template. (Don't worry: we'll remove this duplication later.)

   Answer:

   ```
   <%= form_with(model: @movie) do |f| %>
     <%= f.label :title %>
     <%= f.text_field :title %>

     <%= f.label :description %>
     <%= f.text_area :description, rows: 7 %>

     <%= f.label :rating %>
     <%= f.text_field :rating %>

     <%= f.label :released_on %>
     <%= f.date_select :released_on, {}, {class: "date"} %>

     <%= f.label :total_gross %>
     <%= f.number_field :total_gross %>

     <%= f.submit %>
   <% end %>
   ```

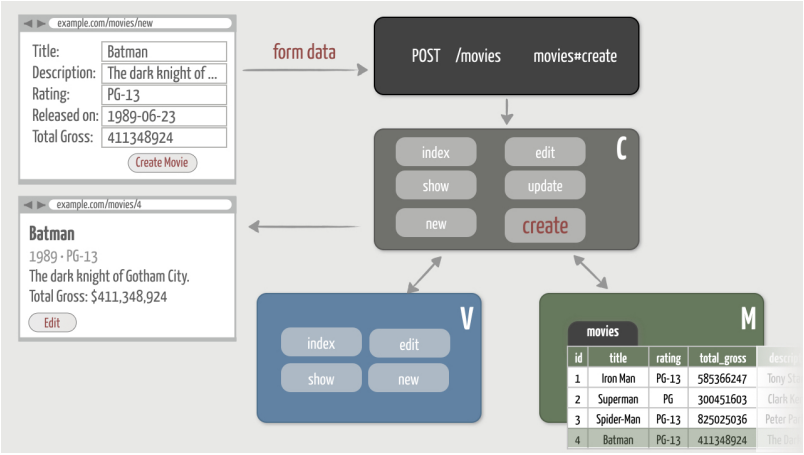
   When you're done, refresh the page and the form should be displayed with all the fields blank. That seems reasonable given that all the attributes in the `@movie` object are `nil`. It's a *new* object! (Remember, we called `Movie.new` in the `new` action.)

   Did you notice that the submit button says "Create Movie"? The `form_with` helper knows that the `@movie` object doesn't yet exist in the database, so you must be trying to create it.

Excellent! We've put up a form for creating new movies. Now on to handling what happens when you submit the form...

3. Implement the Create Action

Our next task is to arrange things in our app so that it creates the movie in the database when the form is submitted. Here's what we need to do, visually:



We're feeling pretty cavalier, so let's just see what happens when we submit the form...

1. Enter a movie title and submit the form by clicking the "Create Movie" button. You probably know why nothing seems to happen, but go ahead and look in the Terminal or command prompt window where the Rails server is running. And if you scroll back a bit, you should see the following error:

   ```
   AbstractController::ActionNotFound (The action 'create' could not be found for MoviesController)
   ```

2. Implement the `create` action in the `MoviesController`. To do that, you'll need to do three things. Remember the steps?

   1. Initialize a new `Movie` object using the submitted form data and assign it to a `@movie` instance variable
   2. Save the movie to the database
   3. Redirect to the movie's show page

   Hint:

   Recall that the `new` method takes a hash of attribute names and values. Conveniently, accessing `params[:movie]` gives you the form data represented as a hash of movie attribute names and values. Put those two things together, and you can instantiate a new `Movie` object that's primed with the form data. Once you've initialized the movie, save it to the database and use the `redirect_to` helper to tell the browser to send a new request for the movie's `show` page.

   Answer:

   ```
   def create
     @movie = Movie.new(params[:movie])
     @movie.save
     redirect_to @movie
   end
   ```

3. Now enter a movie title and submit the form again. KABOOM!

   ```
   ActiveModel::ForbiddenAttributesError
   ```

   Oh hello, our nemesis. Again, we're trying to mass assign attributes using form data and Rails steps in to prevent potentially bad things from happening. Just like with the `update` action, you'll need create a hash of permitted attributes and pass that hash as a parameter to the `new` method. (Feel free to copy the `permit` line from the `update` action for now.)

   Answer:

   ```
   def create
     movie_params =
       params.require(:movie).
         permit(:title, :description, :rating, :released_on, :total_gross)

     @movie = Movie.new(movie_params)
     @movie.save
     redirect_to @movie
   end
   ```

4. Finally, fill out the form for a new movie, preferably one that involves superheroes saving the world from villains. Submit it, and this time the movie should get inserted in the database and you should get redirected to the `show` page where the new movie is displayed.

BOOM! Now we can edit *and* create movies in the web interface!

4. Refactor

Whoa—hold up there a minute, pardner! We're not quite ready to declare victory. Have a look in your `update` and `create` actions and you'll see a wee bit of duplication:

```
movie_params =
  params.require(:movie).
    permit(:title, :description, :rating, :released_on, :total_gross)
```

This sort of unnecessary duplication is what makes apps hard to change. Suppose, for example, that down the road we decide to add new movie attributes and allow them to be mass assigned. We'd have to remember to change the app in two places: in the `update` action *and* the `create` action.

Instead, we'd like to be able to reuse the same list of permitted attributes between those actions. To do that, we'll create a method that returns the permitted attribute list, then call that method from both the `update` and `create` actions. By encapsulating the permit list inside of a method, we can add (or remove) permitted attributes later

simply by changing that single method.

1. Inside of the `MoviesController` class, define a private method called `movie_params` that returns a list of permitted parameters. (The name of the method is arbitrary.) Private methods can't be called from outside of the class, which means private actions aren't treated as web-accessible actions.

   Hint:

   You specify that a method should be private by calling the `private` method on a line by itself within a class definition. Then any methods defined after that point (or until a new access level is set) will be private.

   Answer:

   ```
   class MoviesController < ApplicationController

     # existing public methods (actions)

   private

     def movie_params
       params.require(:movie).
         permit(:title, :description, :rating, :released_on, :total_gross)
     end

   end
   ```

2. Then change the `update` and `create` actions to call your new `movie_params` method to get the permitted parameters, rather than explicitly creating a `movie_params` variable that points to a parameter list.

   Answer:

   ```
   def update
     @movie = Movie.find(params[:id])
     @movie.update(movie_params)
     redirect_to @movie
   end

   def create
     @movie = Movie.new(movie_params)
     @movie.save
     redirect_to @movie
   end
   ```

3. Finally, as a quick test, edit or create a movie. Since we've only refactored code, and not changed any functionality, everything should still work.

Solution

The full solution for this exercise is in the `forms-create` directory of the code bundle.

Bonus Round

Where Did `POST` Come From?

When we submitted the form, the form data was sent to the `create` action. But how did Rails know to do that? Simply put, because that's what the routes dictate.

View the page source of the form and focus in on this part of the HTML form:

```
<form action="/movies" method="post">
  ...
</form>
```

The `action` says that submitting this form will send the data to `/movies`. The `method` says that the HTTP verb will be a `POST`. Rails doesn't override this with a hidden field. So submitting the form will send the form data in a `POST` request to `/movies`.

What do the defined routes have to say about that?

```
Helper          HTTP Verb        Path                  Controller#Action
root_path          GET        /
movies_path        GET        /movies(.:format)           movies#index
                   POST       /movies(.:format)           movies#create
new_movie_path     GET        /movies/new(.:format)       movies#new
edit_movie_path    GET        /movies/:id/edit(.:format)  movies#edit
movie_path         GET        /movies/:id(.:format)       movies#show
                   PUT        /movies/:id(.:format)       movies#update
                   PATCH      /movies/:id(.:format)       movies#update
                   DELETE     /movies/:id(.:format)       movies#destroy
```

Notice there's a route for the verb `GET` and the URL `/movies` *and* a route for the verb `POST` and the URL `/movies`. Given the unique combination of HTTP verb and URL, according to the routes `POST` requests for `/movies` are handled by the `create` action in the controller. It's that simple!

Wrap Up

Now that you're getting good with forms and their conventions, it's time for some celebration! Do your victory dance, give a couple fist pumps, or (if those options just seem *too* crazy) check your Twitter or Slack stream. Between editing movies and creating movies, you've learned a ton!

Up to this point we've implemented all the routes except delete, which we'll get to shortly. In the next section, we'll use partials to remove the duplication we created when we copy/pasted the HTML form code between the `edit` and `new` pages.

Partials

Exercises

Objective

Crafting good Rails apps isn't just about implementing features that do what they should do. Good Rails apps also have clean, well-organized code under the hood. Indeed, what you see on the inside is as important as what you see on the outside. And that's because an online web app is never really done. It's inevitable that you'll need to add new features, be it tomorrow or six months down the road. And when that time comes, you want your app to be in a position where you can make the changes as quickly and efficiently as possible.

So let's take stock of where we left off in the last exercise. We can now edit and create movies. We even removed a bit of duplication in the `update` and `create` actions so that the controller code is easier to change later. But what about the `edit` and `new` forms themselves? We took a deliberate shortcut when we blatantly copied the form in the `edit.html.erb` template and pasted it into the `new.html.erb` template. We did that to quickly get up and running and learn how forms work. But duplicated code anywhere in an application, even in a view template, is a liability. If we need to change the form to accommodate a future feature, we'll need to make the change in two places. And that's both prone to error (we'll likely forget to make the change in *both* places) and also double the work.

So before we strike a big red line through these features and move on, let's take a minute to clean up the duplication in the forms. We'll also restructure our layout file so it's better organized.

1. Remove Duplication Using a Partial

As it stands, the `edit.html.erb` and `new.html.erb` templates both have the exact same form code. We want the form code in *one* place, so we'll put the form in a common *partial file*. Think of a partial as a reusable view "chunk" (yes, that's the technical term) that you can render from other views. Once we have the form in a partial, we'll render it from the `edit.html.erb` and `new.html.erb` templates. It's two forms for the price of one!

1. Start by creating a file named `_form.html.erb` (partial files always start with an underscore) in the `app/views/movies` directory.
2. Then go into the `edit.html.erb` template and cut all the code from `form_with` to `end` and paste it into the `_form.html.erb` file.

   Answer:

   ```
   <%= form_with(model: @movie) do |f| %>
     <%= f.label :title %>
     <%= f.text_field :title %>

     <%= f.label :description %>
     <%= f.text_area :description, rows: 7 %>

     <%= f.label :rating %>
     <%= f.text_field :rating %>

     <%= f.label :released_on %>
     <%= f.date_select :released_on, {}, {class: "date"} %>

     <%= f.label :total_gross %>
     <%= f.number_field :total_gross %>

     <%= f.submit %>
   <% end %>
   ```

3. Then, back in the `edit.html.erb` template, render the `_form` partial.

   Hint:

   Remember that even though the partial file is prefixed with an underscore, you don't use the underscore when calling `render`.

   Answer:

   ```
   <h1>Editing Movie: <%= @movie.title %></h1>

   <%= render "form" %>
   ```

4. Now browse to http://localhost:3000/movies/1/edit and you shouldn't see any evidence that we shuffled code around. The form should appear exactly as it did before. One template down, one to go!
5. Next, go into the `new.html.erb` template and replace the inline form code with the result of rendering our fancy new form partial.

   Answer:

   ```
   <h1>Create a New Movie</h1>

   <%= render "form" %>
   ```

6. As a sanity check, browse to http://localhost:3000/movies/new and again the form should appear exactly as before.

It's like we were never here...

2. Use a Local Variable

Currently the `_form` partial depends on an `@movie` instance variable. This certainly works, but it means the partial is tightly coupled to the controller actions that render it, which in this case is the `new` and `edit` actions.

Rather than using instance variables in partials, it's considered a best practice to instead explicitly pass the partial the data it needs using local variables. That way the partial isn't dependent on something outside of its scope, it just relies on local variables that were passed to it. And that makes the partial easier to reuse and maintain over

time.

So let's rearrange things slightly so the `_form` partial uses a local variable named `movie`.

1. First, in the `_form.html.erb` partial, change the `form_with` to use a local `movie` variable rather than a `@movie` instance variable.

   Answer:

   ```
   <%= form_with(model: movie) do |f| %>
     # existing fields
   <% end %>
   ```

2. Then in the `new.html.erb` template where the form partial is rendered, pass it a local variable named `movie` that has the value of the `@movie` instance variable.

   Answer:

   ```
   <h1>Create a New Movie</h1>

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

3. And do the same in the `edit.html.erb` template where it renders the form partial.

   Answer:

   ```
   <h1>Editing <%= @movie.title %></h1>

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

It might help to think of partials as being like simple functions. Functions get passed arguments, and those arguments are scoped to the function as local variables. In general, functions should avoid relying on global variables that are outside the scope of a function. In the same way, partials should be passed local variables that are then scoped to the partial rather than relying on instance variables that are outside a partial's scope.

3. Accommodate New Features

Here comes the big payoff! Suppose some bright spark decides that all the movie forms should auto-focus the title field. And while we're at it, the description field should have 10 rows instead of 7 rows. The person requesting these features would like an estimate of how long all this will take on his desk by tomorrow morning, preferably hand-signed. What he doesn't know is that we're way ahead of the game now.

1. Change the shared `_form.html.erb` partial to autofocus the title field.

   Answer:

   ```
   <%= form_with(model: movie) do |f| %>
     <%= f.label :title %>
     <%= f.text_field :title, autofocus: true %>

     <%= f.label :description %>
     <%= f.text_area :description, rows: 10 %>

     <%= f.label :rating %>
     <%= f.text_field :rating %>

     <%= f.label :released_on %>
     <%= f.date_select :released_on, {}, {class: "date"} %>

     <%= f.label :total_gross %>
     <%= f.number_field :total_gross %>

     <%= f.submit %>
   <% end %>
   ```

2. Browse to http://localhost:3000/movies/1/edit and http://localhost:3000/movies/new and the title field should automatically have focus and the description field should have 10 rows.

3. Finally, send the person who requested these changes a one-word email: "Done!"

4. Extract Header and Footer Into Partials

We've seen that partials offer an easy way to remove view-level duplication. You can also use partials to decompose a view template into small chunks for better readability and maintenance.

For example, the header and footer sections of a layout can become fairly substantial as an application grows. And as things start to accumulate, the layout file can morph into an unrecognizable jumble of HTML and ERb tags. To help keep that under control, it's generally a good practice to extract sections of views (including layouts) that logically go together into partials.

So let's do ourselves a favor now by decomposing the layout file into separate header and footer partials. When we're done, you won't see any visible changes on the site. But you'll feel better knowing the internal design is cleaner and easier to maintain.

1. Create a file named `_header.html.erb` in the `app/views/layouts` directory. Copy the header section from the `application.html.erb` layout file and paste it into the new `_header.html.erb` partial.

   Answer:

   ```
   <header>
     <nav>
       <%= link_to image_tag("logo"), root_path %>
       <ul>
         <li>
           <%= link_to "All Movies", movies_path %>
         </li>
   ```

```
      </ul>
    </nav>
  </header>
```

2. Then, back in the layout file, replace the header section with a call to render the `layouts/header` partial.

   Hint:

   By default, `render` looks for the partial in the same directory as the current view template being rendered. If there's a slash (`/`) in the partial name, `render` treats the first part as the directory name and the second part as the file name. Remember that you'll need to use the ERb tag `<%= %>` (with an equal sign) in order for the rendered partial to get substituted back into the layout file.

   Answer:

   ```
   <%= render "layouts/header" %>
   ```

3. Getting the hang of this? How about trying the footer on your own.

   Hint:

   Create a file named `_footer.html.erb` in the `app/views/layouts` directory. Copy the footer section from the `application.html.erb` layout file and paste it into the new `_footer.html.erb` partial.

   Answer:

   ```
   # in _footer.html.erb

   <footer>
     <p>
       Copyright &copy; 2005-<%= Time.now.year %>
         <%= link_to 'The Pragmatic Studio','https://pragmaticstudio.com' %>
     </p>
   </footer>

   # in application.html.erb

   <%= render "layouts/footer" %>
   ```

4. Finally, check your work by refreshing the browser. The app should look exactly as it did before, but now we've organized things better into partials. If you run into trouble, double check that your layout file matches up with the version shown in the answer.

   Answer:

   ```
   <%= render "layouts/header" %>

   <div class="content">
     <%= yield %>
   </div>

   <%= render "layouts/footer" %>
   ```

Solution

The full solution for this exercise is in the `partials` directory of the code bundle.

Wrap Up

That's really all there is to partials. Most often you'll use them to remove duplication in views so you can make changes in one place. And in the same way you'd break a big Ruby method into smaller methods for better readability, you can use partials to break big view templates into smaller (reusable) view chunks.

A bit later we'll see more examples of where partials come in handy. But first, in the next section we'll implement the last remaining route: deleting a movie.

Dive Deeper

To learn more about partials, refer to the Rails Guides: Using Partials.

Destroying Records

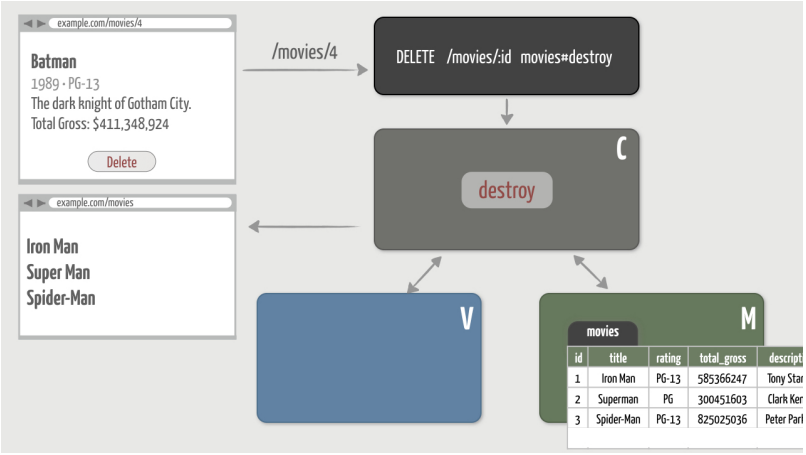
Exercises

Objective

Our web interface is coming together nicely. We can create, read, and update movies in the browser. So what's left? Well, there's currently no way to delete a movie via the web. To do that, we need to:

- Generate a "Delete" link on the `show` page
- Define a `destroy` action in the `MoviesController` that deletes the movie from the database

Visually, here's what we want to do:



Notice that we don't need to add a new route to handle deleting movies. The resource routing conventions we used in a previous exercise already have that route set up for us. So given everything we've learned so far, this task is well within your reach...

1. Add a Delete Link

We'll start by generating a "Delete" link on the `show` page. To do that, we'll enlist the help of a route helper method.

1. Start by reviewing all the routes we have so far:

   ```
   Helper          HTTP Verb        Path                    Controller#Action
   root_path          GET        /
   movies_path        GET        /movies(.:format)          movies#index
                      POST       /movies(.:format)          movies#create
   new_movie_path     GET        /movies/new(.:format)      movies#new
   edit_movie_path    GET        /movies/:id/edit(.:format) movies#edit
   movie_path         GET        /movies/:id(.:format)      movies#show
                      PUT        /movies/:id(.:format)      movies#update
                      PATCH      /movies/:id(.:format)      movies#update
                      DELETE     /movies/:id(.:format)      movies#destroy
   ```

   Whew! This is ending up as a checklist of to-do items, and we're almost done. The only route we haven't implemented is on the last line. According to that route, a `DELETE` request for `/movies/:id` maps to the `destroy` action in the `MoviesController`. (Yeah, it's confusing that the action isn't called `delete`, but what can we say.)

   So, what's the name of the route helper method that generates link matching that route? Well, there's no helper in the leftmost column, but if you look up that column you'll run into the helper `movie_path`. Unless otherwise specified, route names get inherited from one row to the next. So the name of the route helper method to delete a movie is `movie_path`.

2. At the bottom of the `show` page, generate the "Delete" link using the helper method.

   Hint:

   The route has an `:id` placeholder—we have to identify the movie we want to delete—so you have to pass the `@movie` object as a parameter.

   Answer:

   ```
   <section class="admin">
     <%= link_to "Edit", edit_movie_path(@movie), class: "button" %>
     <%= link_to "Delete", movie_path(@movie), class: "button" %>
   </section>
   ```

3. Now go ahead and refresh the `show` page and click the "Delete" link. You should end up right back on the `show` page. But it should have tried to run a not-yet-defined `destroy` action and failed, right? Hmm... something's not quite right.

   This is another one of those times when the easiest way to diagnose what's going on is to watch the log file. Open the console window where your app is running, hit `Return` a few times to create a bunch of white space in the scrolling log, and then in the browser click the "Delete" link again. In the log file you should see an incoming request that looks like this:

   ```
   Started GET "/movies/1" for ::1 at YYYY-07-08 16:04:32 -0600
   Processing by MoviesController#show as TURBO_STREAM
   ```

   The URL `/movies/1` looks right, but notice that the HTTP verb is `GET`. And according to the routes, a `GET` request for `/movies/:id` maps to the `show` action. So the router is doing exactly what we told it to do! The problem is the link we generated isn't triggering the delete route.

4. Returning your attention to the defined routes, notice that the `show` and `destroy` routes share a lot in common:

   ```
   movie_path   GET      /movies/:id(.:format)    movies#show
                DELETE   /movies/:id(.:format)    movies#destroy
   ```

   Interestingly, both of these routes recognize the same requested URL: `/movies/:id`. As well, they both have the same route helper method: `movie_path`. The *only* difference between these two routes is the HTTP verb (in the second column from the left). To show a movie we need to send a `GET` request and to delete a movie we need to send a `DELETE` request.

5. So how exactly do we send a `DELETE` request? Unfortunately, browsers can't send `DELETE` requests natively, so Rails fakes them using some JavaScript.

   Change the "Delete" link to trigger the delete route by passing the `data` option (its expects a hash) and setting `turbo_method` to `:delete`, like so:

```
<section class="admin">
  <%= link_to "Edit", edit_movie_path(@movie), class: "button" %>
  <%= link_to "Delete", movie_path(@movie), class: "button",
        data: { turbo_method: :delete, turbo_confirm: "Are you sure?" } %>
</section>
```

The `turbo_method` data attribute is used to override the HTTP method. We've also used the `turbo_confirm` data attribute to pop up a JavaScript confirmation dialog when the link is clicked. That way movies don't *accidentally* get deleted.

What's Turbo, you ask? Turbo is the thing in Rails that automatically speeds up page navigation. It's not important to understand how it works for the purposes of this course. You can learn all about Turbo later in our Hotwire for Rails Developers course!

6. To see what that generated, refresh the page and view the source. Search for the "Delete" link and you should see this little nugget:

   ```
   <a class="button" data-turbo-method="delete" data-turbo-confirm="Are you sure?" href="/movies/14">Delete</a>
   ```

   Notice how the `data` options got turned into `data-*` attributes and values. For example, the `data-turbo-method` attribute has a value of `delete`. That tells Rails that clicking this link should send a `DELETE` request rather than a `GET` request.

7. Finally, click the "Delete" link again and this time a confirmation dialog should pop up. Click "OK" and in the server log you should see the following error:

   ```
   AbstractController::ActionNotFound (The action 'destroy' could not be found for MoviesController)
   ```

   Perfect! Now we know we have the "Delete" link properly configured to trigger the delete route.

2. Implement the Destroy Action

It's all downhill from here...

1. Implement the `destroy` action in the `MoviesController`. Once the movie has been destroyed, redirect to the movie listing page. And because the "Delete" link changed the HTTP verb from a `GET` to a `DELETE`, you'll need to set the redirection `status` to `:see_other` which is an alias for a 303 HTTP status code. Doing that ensures the redirect will be followed using a `GET` request, rather than a `DELETE` request.

   Hint:

   First, find the movie to delete using `params[:id]` and assign it to an `@movie` instance variable. Then call the `destroy` method on the movie object. Finally, redirect to the movie listing page.

   Answer:

   ```
   def destroy
     @movie = Movie.find(params[:id])
     @movie.destroy
     redirect_to movies_url, status: :see_other
   end
   ```

   Strictly speaking, you don't need to assign the movie to an instance variable since the `destroy` action never shows the movie. It just redirects. That being the case, you could assign the movie to a local variable. But for consistency, we tend to use an instance variable.

2. Then go back to the browser and click the "Delete" link again. This time you should end up back on the movie listing page and the movie you deleted shouldn't be displayed in the listing.

3. Just to see what happened behind the scenes, check the log file (you may need to scroll back a tad). You should see that indeed clicking the link sent a `DELETE` request and the movie was deleted from the database:

   ```
   Started DELETE "/movies/14" for ::1 at YYYY-07-08 16:12:51 -0600
   Processing by MoviesController#destroy as TURBO_STREAM
     Parameters: {"authenticity_token"=>"[FILTERED]", "id"=>"14"}
   Movie Load (0.1ms)  SELECT "movies".* FROM "movies" WHERE "movies"."id" = ? LIMIT ?  [["id", 14], ["LIMIT", 1]]
     (0.0ms)  begin transaction
   Movie Destroy (0.3ms)  DELETE FROM "movies" WHERE "movies"."id" = ?  [["id", 14]]   (1.1ms)  commit transaction
   Redirected to http://localhost:3000/movies
   Completed 302 Found in 5ms (ActiveRecord: 1.6ms | Allocations: 2871)
   ```

   (The HTTP verb is `DELETE` and the SQL statement uses `DELETE`, but the conventional Rails action is called `destroy`. Go figure.)

4. Finally, use your fancy new web interface to re-create the movie you just deleted! :-)

Solution

The full solution for this exercise is in the `destroy` directory of the code bundle.

Bonus Round

Book App

Suppose you had a `Book` resource and you wanted to create a web interface for creating, reading, updating, and deleting (CRUD) books. To do that, you'd tell the Rails router to generate all the routes using:

```
resources :books
```

Given the routing conventions, can you fill in the table below?

resources :books

Helper	HTTP Verb	Path	Controller#Action
	GET	/books	books#index
		/books	
new_book_path	GET		books#new
edit_book_path	GET		books#edit
book_path	GET	/books/:id	
			books#update
	PUT	/books/:id	books#update
	DELETE	/books/:id	

Answer:

resources :books

Helper	HTTP Verb	Path	Controller#Action
books_path	GET	/books	books#index
	POST	/books	books#create
new_book_path	GET	/books/new	books#new
edit_book_path	GET	/books/:id/edit	books#edit
book_path	GET	/books/:id	books#show
	PATCH	/books/:id	books#update
	PUT	/books/:id	books#update
	DELETE	/books/:id	books#destroy

Wrap Up

Congratulations, you've now successfully implemented all these routes!

```
root_path          GET        /
movies_path        GET        /movies(.:format)           movies#index
                   POST       /movies(.:format)           movies#create
new_movie_path     GET        /movies/new(.:format)       movies#new
edit_movie_path    GET        /movies/:id/edit(.:format)  movies#edit
movie_path         GET        /movies/:id(.:format)       movies#show
                   PUT        /movies/:id(.:format)       movies#update
                   PATCH      /movies/:id(.:format)       movies#update
                   DELETE     /movies/:id(.:format)       movies#destroy
```

Here's the really cool part: All resources follow these same routes and conventions. So once you understand how these eight routes work, you're golden. In your own app, tell the Rails router to generate all the routes for you and then you're ready to step through implementing each route, exactly as we did for movies.

When you boil it down, the only thing that changes is the names of things. That's the power of conventions. And through these conventions, Rails does a lot to help us quickly stand up CRUD-based applications. That gives you more time to focus on the stuff that makes your application unique: the business logic.

And in the next section we'll start adding some of that to our app...

Custom Queries

Exercises

Objective

Look under the hood of almost every Rails app and you'll find custom queries. They pop up all over the place, and for good reason. It's very common to want to fetch a subset of data from the database.

Currently our movie listing shows all the movies in the database. Trouble is, the folks in Hollywood often announce upcoming movies before they're officially *released*. It's the oldest marketing trick in the book. You watch an amazing movie trailer only to discover at the end that the movie won't be released until next Christmas! (Do we sound bitter?)

Anyway, we don't want our app to get people's hopes up. So instead of listing all the movies, we only want to list movies that have already been released. You know, the movies you can actually watch! And we'll order the movies so that the most recently-released movie is at the top of the list. To do that, we'll need to define a custom query that fetches a subset of movies from the database in the order we want. Then we'll update the `index` action to fetch the movies using the custom query method.

1. Create a Not-Yet-Released Movie

All the movies we've added so far have been released, so either set the `released_on` date of one of your movies to a date in the future or create a new movie that hasn't yet been released.

Here's a quick example of creating a not-yet-released movie in the `console` if you want to bypass using the web form:

```
>> movie = Movie.new(title: "Batman vs. Godzilla", description: "An epic battle between The Caped Crusader and the fire-breathing dinosaur Goj
>> movie.released_on = 10.days.from_now
>> movie.save
```

2. Write the Query in the Console

Before jumping straight into the task at hand, let's spend a minute in the `console` writing the query we want. We almost always experiment with queries in the `console` first. It's faster and easier than plunking the query code directly into an app and then having to refresh the browser to see if it worked.

In the `console`, the feedback is nearly instantaneous, and that's a huge win when you're learning something new. The `console` is especially useful for playing with queries because it automatically prints the generated SQL. So we suggest always keeping a `console` session open when you're working on a Rails application... and using it frequently!

1. Spark up a `console` session and start by writing a query that returns only the movies that have been released as of right now (hint!). Don't worry about the order yet.

   Hint:

   The `Movie` model inherits a `where` method from its parent class `ApplicationRecord`. You want all the movies where the value of the `released_on` column is less than the current date. Use a placeholder to substitute in the date value to compare against.

   Answer:

   ```
   Movie.where("released_on < ?", Time.now)
   ```

   Double-check that your not-yet-released movie isn't included in the results.

2. Then extend that query so that the released movies are ordered with the most-recently released movie first.

   Hint:

   The `Movie` model also inherits an `order` method from its parent class `ApplicationRecord`. Use that method to order the results of the `where` clause. You want the movies ordered by their `released_on` attribute, in descending order so the most recently-released movie is first.

   Answer:

   ```
   Movie.where("released_on < ?", Time.now).order("released_on desc")

   # or using Ruby symbols:

   Movie.where("released_on < ?", Time.now).order(released_on: :desc)
   ```

   Make sure the first element of your resulting array contains the most-recently released movie.

Good! Now let's use that query in the app...

2. List Released Movies

Returning to our original task, we want the movie `index` page to only show movies that have been released and order them with the most recently-released movie at the top. We now have the query for doing that in our `console` session. So where do we put this query code in the app?

Determining what it means for a movie to be "released" is a type of business logic. You could easily imagine a business person deciding to change the definition of "released" in this app to include movies that will be released within the next week, just to whet everyone's appetite. So it's important that we encapsulate the criteria for released movies in one place in our app. As a general rule, all database-related stuff goes in the model: the `Movie` model in this case.

1. Add a class-level method named `released` to the `Movie` class. It should return all the movies that have been released, ordered with the most recently-released movie first.

   Hint:

   Remember that class-level methods are defined on `self`. You call these methods using the class (e.g. `Movie`).

   Answer:

   ```
   class Movie < ApplicationRecord
     def self.released
       where("released_on < ?", Time.now).order("released_on desc")
     end
   end
   ```

2. Then change the `index` action in your `MoviesController` class so that only released movies are displayed in the movie listing.

   Hint:

   Rather than calling the `all` method, call your new `released` method instead.

   Answer:

   ```
   def index
     @movies = Movie.released
   end
   ```

3. To check your work, browse to http://localhost:3000 to make sure only the released movies are shown in the movie listing and ordered properly.

Solution

The full solution for this exercise is in the `querying` directory of the code bundle.

Bonus Round

Specifying Where Conditions

The `where` method accepts conditions in several forms.

First, we've seen how the conditions can be specified as a string with a placeholder:

```
Movie.where("released_on < ?", Time.now)
```

The first argument is a conditions string, and any additional arguments will replace the question marks (?) in it.

Using a placeholder (?) makes sure that the result of `Time.now` is automatically converted to the proper native database type. You might think you could just interpolate the result of `Time.now` into the string, like this:

```
Movie.where("released_on < #{Time.now}")
```

But that gives an error because `Time.now` returns a datetime type that's not compatible with database. So when specifying conditions with dates, you must use a placeholder.

However, you don't have to use a placeholder for conditions with standard values. For example, the following works because `225000000` is a value that the database recognizes:

```
Movie.where("total_gross < 225000000")
```

But the following won't work because `225_000_000` is a Ruby-specific representation of the same number:

```
Movie.where("total_gross < 225_000_000")
```

In that case, you would need to use a placeholder so that the value is properly converted:

```
Movie.where("total_gross < ?", 225_000_000)
```

The `where` method also accepts a hash of conditions. For example, to find all the movies where the rating is "PG-13", you can use

```
>> Movie.where(rating: "PG-13")
```

Notice `rating` is a Ruby symbol representing a hash key and `"PG-13"` is the corresponding value which is used to qualify the condition.

That's basically a shortcut for doing this:

```
Movie.where("rating = 'PG-13'")
```

Finally, you can pass `where` a range of values. For example, to find all the movies where the total gross is between $250M and $900M, you can use:

```
Movie.where(total_gross: 250_000_000..900_000_000)
```

And here's something cool: you can pass an incomplete range to perform a greater/less than comparison. For example, to find all the movies with a total gross of $250M or greater, use:

```
Movie.where(total_gross: 250_000_000..)
```

Or to find all the movies with a total gross of $900M or less, use:

```
Movie.where(total_gross: ..900_000_000)
```

Neato!

Experiment With More Queries

We've seen that the `Movie` model inherits a number of methods for querying the database from its parent class `ApplicationRecord`. As a recap of the query methods we've used so far, here's a practice drill of sorts:

1. Spark up a `console` session and start with a gimme: Find the movie that has the id 2.

   Answer:

   ```
   >> Movie.find(2)
   ```

2. How would you find the movie that has the title "Iron Man"? (If you've changed your movies, just try a different title.)

   Answer:

   ```
   >> Movie.find_by(title: "Iron Man")
   ```

3. Now count all the movies in the database. At this point you may have added and deleted movies, so just make sure the count is correct.

   Answer:

   ```
   >> Movie.count
   ```

4. Then call the query method that finds all the movies.

Answer:

```
>> Movie.all
```

5. Now order all the movies by their total gross, with the *lowest* grossing movie first.

   Answer:

   ```
   >> Movie.order("total_gross")

   # or

   >> Movie.order("total_gross asc")

   # or

   >> Movie.order(total_gross: :asc)
   ```

6. Flip it around. Reverse the order so the highest-grossing movie is first in the array.

   Answer:

   ```
   >> Movie.order("total_gross desc")

   # or

   >> Movie.order(total_gross: :desc)
   ```

7. OK, how would you find all the flop movies where (hint!) the total gross is less than or equal to $225M?

   Answer:

   ```
   >> Movie.where("total_gross <= ?", 225_000_000)

   # or

   >> Movie.where("total_gross <= 225000000")

   # or

   >> Movie.where(total_gross: ..225_000_000)
   ```

8. What about finding all the hit movies with a total gross greater than or equal to $300M?

   Answer:

   ```
   >> Movie.where("total_gross >= ?", 300_000_000)

   # or

   >> Movie.where("total_gross >= 300000000")

   # or

   >> Movie.where(total_gross: 300_000_000..)
   ```

9. Now chain together two query methods to find all the hit movies and order them so that the highest-grossing movie is first.

   Answer:

   ```
   >> Movie.where("total_gross >= 300000000").order("total_gross desc")
   ```

10. What about counting all the hit movies?

    Answer:

    ```
    >> Movie.where("total_gross >= 300000000").count
    ```

11. How about finding all the movies where the rating is "PG"?

    Answer:

    ```
    >> Movie.where(rating: "PG")
    ```

12. The `not` method generates a SQL NOT condition. Use it to find all the movies where the rating is *not* "PG".

    Answer:

    ```
    >> Movie.where.not(rating: "PG")
    ```

13. The `not` method can also accept an array to ensure multiple values are not in a field. Find all the movies where the rating is *not* "PG" or "PG-13".

    Answer:

    ```
    >> Movie.where.not(rating: ["PG", "PG-13"])
    ```

Great job!

Printing the Generated SQL

Sometimes you want to see the SQL a query method will generate without actually running the query. To do that, you can tack on a `to_sql` method call, like so:

```
>> Movie.where("total_gross >= 300000000").order("total_gross desc").to_sql
```

And prints the generated SQL but doesn't run the query:

```
=> "SELECT "movies".* FROM "movies" WHERE (total_gross >= 300000000) ORDER BY total_gross desc"
```

Handy!

More Custom Query Methods

If you want to flex your query muscles, try declaring custom query methods that return movies matching the following conditions:

- **hit movies:** movies with at least $300M total gross, ordered with the highest grossing movie first
- **flop movies:** movies with less than $225M total gross, ordered with the lowest grossing movie first
- **recently added movies:** the last three movies that have been created, ordered with the most recently-added movie first. Hint: you need to *limit* the result to 3 records.

Answer:

```
def self.hits
  where("total_gross >= 300000000").order(total_gross: :desc)
end

def self.flops
  where("total_gross < 22500000").order(total_gross: :asc)
end

def self.recently_added
  order("created_at desc").limit(3)
end
```

Test each query method in the `console`. (If you already have an active `console` session, you'll need to use `reload!`.) Use the `to_sql` method to print out the generated SQL for each query. And for fun, chain a few of the query methods together and print out the generated SQL.

Answer:

```
>> Movie.hits

>> Movie.hits.to_sql
=> "SELECT "movies".* FROM "movies" WHERE (total_gross >= 300000000) ORDER BY total_gross desc"

>> Movie.flops

>> Movie.flops.to_sql
=> "SELECT "movies".* FROM "movies" WHERE (total_gross < 22500000) ORDER BY total_gross asc"

>> Movie.recently_added

>> Movie.recently_added.to_sql
=> "SELECT "movies".* FROM "movies" ORDER BY created_at desc LIMIT 3"

>> Movie.recently_added.flops
>> Movie.recently_added.hits

>> Movie.released.flops
>> Movie.released.hits

>> Movie.released.hits.to_sql
=> "SELECT "movies".* FROM "movies" WHERE (released_on < '2019-06-11 21:05:03.596359') AND (total_gross >= 300000000) ORDER BY released_on des
```

You can experiment with using these query methods in the `index` action of your `MoviesController`, though we'll continue to use the `released` query in subsequent exercises.

Wrap Up

Now you know how to query the database any way you please! The really nifty part is that the query methods generate the actual SQL and run the database query for you. Basically, all the queries you could perform with standard SQL can be expressed using these high-level query methods in your models.

The machinery that makes all this possible is another gem that's included with Rails called ActiveRelation (ARel). It's a query generation library that knows how to generate SQL for all the supported databases. So just like with migrations, no matter which database you use—SQLite, MySQL, PostgreSQL, Oracle, or the like—the queries that ARel generates will run on any of them. That's one less thing for us to worry about!

In the next section we'll add another migration, which will give us an opportunity to see how migrations are reversible...

Dive Deeper

To learn more about all the built-in query methods, refer to Rails Guide: Active Record Query Interface.

Migrations Revisited

Exercises

Objective

As our app has evolved and you've had a chance to play around with movies in the web interface, you probably noticed that we're missing some important information. Any respectable online movie app needs to show a movie's poster image. It would also be handy to know the movie's director and duration.

We're faced with adding new fields to the database, and that always means we need a new migration. Now, we already know how to generate new migration files, so that part is easy. But any time you migrate the database, you also need to think about how it will affect the other parts of the app. Generally speaking, adding new fields has a ripple effect, and you'll need to make the necessary adjustments. This exercise gives us a good opportunity to work through that process.

To complete this exercise we'll need to draw upon (practice!) a lot of things we've done so far:

1. Generate a new migration file that adds more fields to the `movies` database table.
2. Change the movie `index` and `show` templates to display the new movie fields.
3. Change the form to include form elements for the new fields.
4. Update the existing movies in the database to have values for the new fields.

Again, much of this will be review, but adding new migrations is so common that it's good to put all this together so we're really comfortable with the steps. We'll also learn some new things about migrations along the way.

1. Add More Database Fields

Our first task is to add the following new fields and types to the `movies` database:

name	type
director	string
duration	string
image_file_name	string

1. Use the migration generator to generate a migration named `AddMoreFieldsToMovies` that adds columns for the fields and types listed above to the `movies` table. You can either generate it all in one fell swoop using the naming conventions, or generate the file and then edit the `change` method to add the appropriate columns.

   Hint:

   For a refresher on the migration generator options, run the `rails g migration` command.

   Answer:

   ```
   rails g migration AddMoreFieldsToMovies director:string duration:string image_file_name:string
   ```

2. Then open the generated migration file and set the `default` value of the `image_file_name` column to "placeholder.png". Otherwise, if you don't explicitly set a default, then the `image_file_name` column will have a default value of `nil` which won't work.

   Here's what you want the migration file to look like:

   ```
   class AddMoreFieldsToMovies < ActiveRecord::Migration[7.0]
     def change
       add_column :movies, :director, :string
       add_column :movies, :duration, :string
       add_column :movies, :image_file_name, :string, default: "placeholder.png"
     end
   end
   ```

3. Then apply the new migration.

   Answer:

   ```
   rails db:migrate
   ```

   You should get the following output:

   ```
   == YYYY0708221920 AddMoreFieldsToMovies: migrating ============================================
   -- add_column(:movies, :director, :string)
      -> 0.0056s
   -- add_column(:movies, :duration, :string)
      -> 0.0027s
   -- add_column(:movies, :image_file_name, :string, {:default=>"placeholder.png"})
      -> 0.0029s
   == YYYY0708221920 AddMoreFieldsToMovies: migrated (0.0118s) ===================================
   ```

4. Now try running the migration again, and this time you shouldn't get any output because all the migrations have already run.

5. Just to see where things stand, display the status of the migrations.

   Answer:

   ```
   rails db:migrate:status
   ```

   You should see all the migrations marked as "up":

   ```
   Status   Migration ID    Migration Name
   --------------------------------------------------
      up     YYYY0708225131  Create movies
      up     YYYY0708231037  Add fields to movies
      up     YYYY0709221920  Add more fields to movies
   ```

2. Undo It

Mistakes happen, and sometimes you'll need to "undo" a migration. For example, every Rails developer has at one point misspelled the name of a column when generating a migration and not noticed it until the migration was already applied. And once you run the migration, you can't then just edit the migration and run the migration again because Rails knows the migration has already been run. In those unfortunate cases, you need to "undo" the last migration. Thankfully, that's really easy because migrations are reversible.

1. Just for kicks, roll back the last migration that was applied by typing:

   ```
   rails db:rollback
   ```

   You should get the following output:

   ```
   == YYYY0705221920 AddMoreFieldsToMovies: reverting ============================================
   -- remove_column(:movies, :image_file_name, :string, {:default=>"placeholder.png"})
      -> 0.0356s
   -- remove_column(:movies, :duration, :string)
      -> 0.0379s
   -- remove_column(:movies, :director, :string)
      -> 0.0124s
   == YYYY0705221920 AddMoreFieldsToMovies: reverted (0.0902s) ===================
   ```

   Hey, that's pretty clever! Notice that it reverted the migration by calling the `remove_column` method for each of the fields. The `change` method in the migration file figures out the reverse of `add_column` is `remove_column`, so we don't have to do anything extra to get a reversible migration. Huzzah!

   Note, however, that not all migrations are reversible. For example, if you had a `change` method that dropped a table, the migration wouldn't be able to figure out how to recreate the table. In those cases you can define separate `up` and `down` methods in the migration file for finer control. The `up` method is run when the migration is applied, and the `down` method is run when the migration is reversed.

   Finally, it's important to realize that reversing a migration may cause data loss. If a column is removed, for example, all the data that was in that table's column is now gone forever. In development that's not usually a big deal, but in production it could be *disasterous*! Just something to keep in mind. :-)

2. To see the current state of the migration, go ahead and check the migration status again.

   Answer:

   ```
   rails db:migrate:status
   ```

   You should see the reversed migration is now marked as "down":

   ```
   Status   Migration ID    Migration Name
   --------------------------------------------------
      up     YYYY0708225131  Create movies
      up     YYYY0708231037  Add fields to movies
      down   YYYY0709211855  Add more fields to movies
   ```

3. At this point, if we needed to fix a mistake, we'd just edit the migration file and then re-run the revised migration. We don't have any reason to edit the migration file in this case, so go ahead and re-run the migration.

   Answer:

   ```
   rails db:migrate
   ```

   And now all the migrations should be marked as "up" again:

   ```
   Status   Migration ID    Migration Name
   --------------------------------------------------
      up     YYYY0708225131  Create movies
      up     YYYY0708231037  Add fields to movies
      up     YYYY0709221920  Add more fields to movies
   ```

Excellent—that takes care of the database changes for this feature!

3. Update the Form

Now that we've changed the database schema, we need to update the view templates to reflect those changes.

1. Let's start with the form. Browse to a movie's `show` page and click the "Edit" link. You should totally expect not to see the new fields in the form.

2. Fix that by updating the `form` partial to include text fields for `director`, `duration`, and `image_file_name`.

   Answer:

   ```
   <%= f.label :director %>
   <%= f.text_field :director %>

   <%= f.label :duration %>
   <%= f.text_field :duration %>

   <%= f.label :image_file_name %>
   <%= f.text_field :image_file_name %>
   ```

   Refresh the form and you should see text fields for the new fields. You won't see any values for the `director` or `duration` fields because those values are `nil` by default in the database. The `image_file_name` field, however, should be filled with "placeholder.png" since that's the default value.

3. Before you can submit data for these new form fields, back in your `MoviesController` you'll need to add the new fields to the list of permitted parameters. Otherwise, the values for the new fields will get ignored.

   Answer:

   ```
   def movie_params
     params.require(:movie).
   ```

```
    permit(:title, :description, :rating, :released_on, :total_gross,
           :director, :duration, :image_file_name)
end
```

4. Now use your spiffy new form to update all the movies to have values for the fields we added. You can find example movie info on IMDb or Wikipedia, for example. In an earlier exercise we copied some sample movie poster image files into the `flix/app/assets/images` directory.

   Here are some examples you can paste into the `console` if you want to bypass using the web form.

   Code:

```
movie = Movie.find_by(title: "Avengers: Endgame")
movie.director = "Anthony Russo"
movie.duration = "181 min"
movie.image_file_name = "avengers-end-game.png"
movie.save

movie = Movie.find_by(title: "Captain Marvel")
movie.director = "Anna Boden"
movie.duration = "124 min"
movie.image_file_name = "captain-marvel.png"
movie.save

movie = Movie.find_by(title: "Black Panther")
movie.director = "Ryan Coogler"
movie.duration = "134 min"
movie.image_file_name = "black-panther.png"
movie.save

movie = Movie.find_by(title: "Wonder Woman")
movie.director = "Patty Jenkins"
movie.duration = "141 min"
movie.image_file_name = "wonder-woman.png"
movie.save

movie = Movie.find_by(title: "Avengers: Infinity War")
movie.director = "Anthony Russo"
movie.duration = "149 min"
movie.image_file_name = "avengers-infinity-war.png"
movie.save

movie = Movie.find_by(title: "Green Lantern")
movie.director = "Martin Campbell"
movie.duration = "114 min"
movie.image_file_name = "green-lantern.png"
movie.save

movie = Movie.find_by(title: "Fantastic Four")
movie.director = "Josh Trank"
movie.duration = "100 min"
movie.image_file_name = "fantastic-four.png"
movie.save

movie = Movie.find_by(title: "Iron Man")
movie.director = "Jon Favreau"
movie.duration = "126 min"
movie.image_file_name = "ironman.png"
movie.save

movie = Movie.find_by(title: "Superman")
movie.director = "Richard Donner"
movie.duration = "143 min"
movie.image_file_name = "superman.png"
movie.save

movie = Movie.find_by(title: "Spider-Man")
movie.director = "Sam Raimi"
movie.duration = "121 min"
movie.image_file_name = "spiderman.png"
movie.save

movie = Movie.find_by(title: "Batman")
movie.director = "Tim Burton"
movie.duration = "126 min"
movie.image_file_name = "batman.png"
movie.save

movie = Movie.find_by(title: "Catwoman")
movie.director = "Jean-Christophe 'Pitof' Comar"
movie.duration = "101 min"
movie.image_file_name = "catwoman.png"
movie.save
```

Did you notice that we didn't have to change the form that's shown on the `new` page? Remember, it simply renders the form partial we just changed. How's that for being agile?

4. Update Show and Index Templates

As you updated each movie, you probably noticed that the new fields aren't showing up on either the `show` page or `index` page. But you saw that coming from a mile away, right?

1. Update the `show.html.erb` template to display the new fields. Use the `image_tag` helper to show the image corresponding to the name in the `image_file_name` field.

Answer:

```
<section class="movie-details">
  <div class="image">
    <%= image_tag @movie.image_file_name %>
  </div>
  <div class="details">
   ...
   <table>
      <tr>
        <th>Director:</th>
        <td><%= @movie.director %></td>
      </tr>
      <tr>
        <th>Duration:</th>
        <td><%= @movie.duration %></td>
      </tr>
      <tr>
        <th>Total Gross:</th>
        <td><%= total_gross(@movie) %></td>
      </tr>
    </table>
  </div>
</section>
```

Refresh and you should see all the values you entered in the form, plus a snazzy movie poster image (if you entered one)!

2. Then update the `index.html.erb` template to display the movie's poster image. (We'll only show the `director` and `duration` on the show page.)

   Answer:

   ```
   <section class="movie">
     <div class="image">
       <%= image_tag movie.image_file_name %>
       </div>
       <div class="summary">
         ...
       </div>
     </section>
   ```

   Refresh to see your rockin' new movie listings!

Bonus Round

Update/Add Seed Data

At this point we have a handful of example movies in our database with values for all the fields. Sometime down the road you may want to recreate your database from scratch and automatically "seed" it with these same example movies.

In an earlier exercise, we copied a prepared `seeds.rb` file into your `flix/db` directory, overwriting the existing `seeds.rb` file. Then we used this file to prime (seed) your database with example movies. Unlike a migration file which changes the underlying database *structure*, the `seeds.rb` file simply populates the database with *data*.

But if you take a peek at `flix/db/seeds.rb`, you'll notice that the example movies don't have values for the new fields: `director`, `duration`, and `image_file_name`. So you may want to update that file to include values for the new fields we added in this exercise.

To save you time, you can go ahead and copy/paste the following code into the `flix/db/seeds.rb` file:

Answer:

```
Movie.create!([
  {
    title: 'Avengers: Endgame',
    description:
    %{
      After the devastating events of Avengers: Infinity War, the universe
      is in ruins. With the help of remaining allies, the Avengers assemble
      once more in order to undo Thanos' actions and restore order to the universe.
    }.squish,
    released_on: "2019-04-26",
    rating: 'PG-13',
    total_gross: 1_223_641_414,
    director: 'Anthony Russo',
    duration: '181 min',
    image_file_name: 'avengers-end-game.png'
  },
  {
    title: 'Captain Marvel',
    description:
    %{
      Carol Danvers becomes one of the universe's most powerful heroes when Earth is caught in the middle of a galactic war between two alien
    }.squish,
    released_on: "2019-03-08",
    rating: 'PG-13',
    total_gross: 1_110_662_849,
    director: 'Anna Boden',
    duration: '124 min',
    image_file_name: 'captain-marvel.png'
  },
  {
    title: 'Black Panther',
    description:
    %{
      T'Challa, heir to the hidden but advanced kingdom of Wakanda, must step forward to lead his people into a new future and must confront a
```

```
    }.squish,
    released_on: "2018-02-16",
    rating: 'PG-13',
    total_gross: 1_346_913_161,
    director: 'Ryan Coogler',
    duration: '134 min',
    image_file_name: 'black-panther.png'
  },
  {
    title: 'Avengers: Infinity War',
    description:
    %{
      The Avengers and their allies must be willing to sacrifice all in an attempt to defeat the powerful Thanos before his blitz of devastati
    }.squish,
    released_on: "2018-04-27",
    rating: 'PG-13',
    total_gross: 2_048_359_754,
    director: 'Anthony Russo',
    duration: '149 min',
    image_file_name: 'avengers-infinity-war.png'
  },
  {
    title: 'Green Lantern',
    description:
    %{
      Reckless test pilot Hal Jordan is granted an alien ring that bestows him with otherworldly powers that inducts him into an intergalactic
    }.squish,
    released_on: "2011-06-17",
    rating: 'PG-13',
    total_gross: 219_851_172,
    director: 'Martin Campbell',
    duration: '114 min',
    image_file_name: 'green-lantern.png'
  },
  {
    title: 'Fantastic Four',
    description:
    %{
      Four young outsiders teleport to an alternate and dangerous universe which alters their physical form in shocking ways. The four must le
    }.squish,
    released_on: "2015-08-07",
    rating: 'PG-13',
    total_gross: 168_257_860,
    director: 'Josh Trank',
    duration: '100 min',
    image_file_name: 'fantastic-four.png'
  },
  {
    title: 'Iron Man',
    description:
    %{
      When wealthy industrialist Tony Stark is forced to build an
      armored suit after a life-threatening incident, he ultimately
      decides to use its technology to fight against evil.
    }.squish,
    released_on: "2008-05-02",
    rating: 'PG-13',
    total_gross: 585_366_247,
    director: 'Jon Favreau',
    duration: '126 min',
    image_file_name: 'ironman.png'
  },
  {
    title: 'Superman',
    description:
    %{
      An alien orphan is sent from his dying planet to Earth, where
      he grows up to become his adoptive home's first and greatest
      super-hero.
    }.squish,
    released_on: "1978-12-15",
    rating: 'PG',
    total_gross: 300_451_603,
    director: 'Richard Donner',
    duration: '143 min',
    image_file_name: 'superman.png'
  },
  {
    title: 'Spider-Man',
    description:
    %{
      When bitten by a genetically modified spider, a nerdy, shy, and
      awkward high school student gains spider-like abilities that he
      eventually must use to fight evil as a superhero after tragedy
      befalls his family.
    }.squish,
    released_on: "2002-05-03",
    rating: 'PG-13',
    total_gross: 825_025_036,
    director: 'Sam Raimi',
    duration: '121 min',
    image_file_name: 'spiderman.png'
  },
  {
    title: 'Batman',
    description:
    %{
      The Dark Knight of Gotham City begins his war on crime with his
```

```
      first major enemy being the clownishly homicidal Joker.
    }.squish,
    released_on: "1989-06-23",
    rating: 'PG-13',
    total_gross: 411_348_924,
    director: 'Tim Burton',
    duration: '126 min',
    image_file_name: 'batman.png'
  },
  {
    title: "Catwoman",
    description:
    %{
      Patience Philips seems destined to spend her life apologizing for taking up space. Despite her artistic ability she has a more than resp
    }.squish,
    released_on: "2004-07-23",
    rating: "PG-13",
    total_gross: 82_102_379,
    director: "Jean-Christophe 'Pitof' Comar",
    duration: "101 min",
    image_file_name: "catwoman.png"
  },
  {
    title: "Wonder Woman",
    description:
    %{
      When a pilot crashes and tells of conflict in the outside world, Diana, an Amazonian warrior in training, leaves home to fight a war, di
    }.squish,
    released_on: "2017-06-02",
    rating: "PG-13",
    total_gross: 821_847_012,
    director: "Patty Jenkins",
    duration: "141 min",
    image_file_name: "wonder-woman.png"
  }
])
```

Notice that it uses the `create!` method instead of the usual `create` method. The `!` version of `create` raises an exception if a record can't be created because it's invalid (more on that later). Basically, it means we'll get a heads-up if our seed data is out of whack.

Also notice that we're passing the `create!` method an array, where each array element is a hash of movie attributes. This creation style isn't specific to seeding data. It's just a handy way to create a bunch of records in one fell swoop.

Don't do this now since we already have example movies in the database, but if at some point you recreate the database from scratch and want to populate it with example movies, you would run the following task:

```
rails db:seed
```

We don't want to run the task now because it doesn't automatically clear out existing records in the database. Rather, it's an additive process. So if you were to run the task now, you'd end up adding five more (duplicate) movies to the database.

If instead you want to start from scratch, you can run `rails db:reset`. This task drops and re-creates the database, applies all the migrations, and populates the database with the seed data in `db/seeds.rb`. Or, if you just want to "replant" all the seed data, you can run `rails db:seed:replant` which removes all the data from all the database tables and re-seeds the database tables with the seed data in `db/seeds.rb`.

Solution

The full solution for this exercise is in the `migrations-revisited` directory of the code bundle.

Wrap Up

This exercise was a good opportunity to work through all the steps to accommodate a new migration. Whether you're adding, renaming, or deleting columns, you'll need to go through these same high-level steps:

- generate and apply the migration
- add values for the new fields, if necessary
- update all the affected templates

Before you dive into making the changes, you might consider creating a quick and dirty checklist of what's impacted. Don't worry about this being fancy; it's simply a way to think through all the moving parts. We regularly use the backs of envelopes, napkins, and receipts for these kinds of scribbles.

It's interesting to note that controllers are typically not affected by new migrations, with the exception of the permitted parameter list. As middlemen, controllers just pass the data from models to views without regard for the details of the data. And that's exactly as it should be! The MVC design is all about keeping concerns separated so that changes don't ripple across the *entire* application.

Dive Deeper

To learn more about migrations, refer to the Rails Guides: Migrations.

Model Validations

Exercises

Objective

While creating and updating movies using the web forms, you may have noticed that you can create or update a movie with a blank title. In fact, the app will happily accept other not-so-desirable values, such as "Lots of Money" for the total gross amount. That's a big problem! Without validating what gets entered in the web forms, our database can quickly become corrupted with bad (invalid) data which can also trigger nasty little bugs.

To prevent bad data from getting into the database, we'll declare *validations*. Validations are a type of business rule: they ensure the integrity of your application data and prevent it from being saved in an invalid state. Since models are the gatekeeper to application data, the movie-related validations belong in our `Movie` model.

Over the next two exercises, we'll need to make changes in the model, view, and controller:

- In the `Movie` **model**, we'll declare reasonable data validations.
- Then we'll change the `create` and `update` **controller** actions to handle the cases where the movie wasn't saved in the database because it's not valid.
- Finally in the **view** (the form) we'll display any validation errors to give the customer specific feedback so they can make the necessary corrections.

1. Declare Movie Validations

Let's start in the `Movie` model and work our way back out to the web interface. We'll use the built-in validations that are included with Rails. As you work through this section you might find it helpful to peek at the documentation for the validates method.

Use built-in model validations to enforce the following rules about a movie:

1. Values for the fields `title`, `released_on`, and `duration` must be present.

   Answer:

   ```
   validates :title, :released_on, :duration, presence: true
   ```

2. The `description` field must have a minimum of 25 characters.

   Answer:

   ```
   validates :description, length: { minimum: 25 }
   ```

3. The `total_gross` field must be a number greater than or equal to 0.

   Answer:

   ```
   validates :total_gross, numericality: { greater_than_or_equal_to: 0 }
   ```

4. The `image_file_name` field must be formatted so that the file name has at least one word character and a "jpg" or "png" extension. The regular expression syntax for this one is kinda tricky, so go ahead and copy in the following:

   ```
   validates :image_file_name, format: {
     with: /\w+\.(jpg|png)\z/i,
     message: "must be a JPG or PNG image"
   }
   ```

5. Finally, the `rating` field must have one of the following values: "G", "PG", "PG-13", "R", or "NC-17". Go ahead and add a Ruby constant named `RATINGS` that contains an array of strings like so:

   ```
   RATINGS = %w(G PG PG-13 R NC-17)
   ```

   Can you find a built-in validation that validates whether a value is included in (hint!) a specified array?

   Answer:

   ```
   validates :rating, inclusion: { in: RATINGS }
   ```

6. To make selecting a rating easier in the web interface, change the `rating` form field in `app/views/movies/_form.html.erb` to use a drop-down to select the rating from the list of possible ratings, like so:

   ```
   <%= f.select :rating, Movie::RATINGS, prompt: "Pick one" %>
   ```

   Then browse to either the create or edit form and you should now see a drop-down of options for selecting a movie rating. We've got our belt and suspenders on now! The form only allows you to select specific ratings *and* the model validates the rating, too.

2. Experiment with Validations

Now that we have the validations declared in the `Movie` model, let's try to save an invalid movie using the `console` and see what happens...

1. Fire up a Rails `console` to experiment in.

2. In the `console`, instantiate a new `Movie` object without a title.

   Answer:

   ```
   >> movie = Movie.new
   ```

3. Now try to save the invalid `Movie` object to the database.

   Answer:

   ```
   >> movie.save
   ```

The result should be `false`.

It's important to understand what happened here. When you call the `save` method, it automatically runs all the validations. If a validation fails, a corresponding message is added to the model object's `errors` collection. And if the `errors` collection contains any messages, then the save is abandoned and `false` is returned.

In short, the failed validations are preventing the model from being saved, which is exactly what we want!

4. We know our movie is invalid because the `save` method returned `false`, so there must be some errors waiting for us.

   Inspect the errors by accessing the `errors` collection.

   Answer:

   ```
   >> movie.errors
   ```

   There's quite a bit of information in the output. To dig down into the actual error messages, tack on a call to `full_messages` to get an array of error messages.

   Answer:

   ```
   >> movie.errors.full_messages
   ```

   You can even turn that into a fairly readable English sentence by also tacking on the `to_sentence` method.

   Answer:

   ```
   >> movie.errors.full_messages.to_sentence
   ```

5. How would you get only the error messages that are associated with the `:title` attribute?

   Answer:

   ```
   >> movie.errors[:title]
   ```

6. Finally, create a valid movie object and save it away.

   Answer:

   ```
   >> movie.title = "Hulk"
   >> movie.description = "Bruce Banner transforms into a raging green monster when he gets angry."
   >> movie.released_on = "2003-06-20"
   >> movie.duration = "138 min"
   >> movie.total_gross = 113_107_712
   >> movie.image_file_name = "hulk.png"
   >> movie.rating = "PG-13"

   >> movie.save
   ```

   This time the result should be `true`. The validations ran again, but this time they all passed so no errors were found in the `errors` collection. Consequently, the movie was inserted in the database and the call to `save` returned `true`.

7. Since the `save` was successful, the movie's `errors` collection must be empty. Just for practice, confirm that by asking the movie if it has any errors.

   Answer:

   ```
   >> movie.errors.any?
   ```

Solution

The full solution for this exercise is in the `model-validations` directory of the code bundle.

Wrap Up

Perfect! Now whenever you try to create or save a `Movie`, Rails will run all the validations. The movie only gets saved if the `errors` collection is empty. So our `Movie` model now dutifully prevents bad data from getting into our database.

As you build your own app, keep model validations in mind every time you write a migration and add new fields to your database. Rails includes a bunch of common validations, but you can also create your own custom validations when necessary.

Now that we have that foundation, we'll move up into the web interface to handle cases where users enter invalid data.

Dive Deeper

To learn more about validations, and how to write custom validations, refer to Rails Guides: Active Record Validations.

Handling Validation Errors

Exercises

Objective

Now that we have validations in the `Movie` model, let's turn our attention to the user interface. We still need to:

- Change the `create` action to redisplay the form when a user tries to create an invalid movie.

- Display validation errors to help the user fix the problem.
- Change the `update` action to handle validation errors when editing an existing movie.

1. Handle Errors During Create

The `create` action currently assumes that calling `save` always successfully creates the movie in the database:

```
def create
  @movie = Movie.new(movie_params)
  @movie.save
  redirect_to @movie
end
```

But that's wishful thinking! Now that we have validations, `save` could return either `true` or `false`. So we need to update the action to handle both cases.

1. Change the `create` action so that if the `save` call returns `true` (the model was saved), the action redirects to the movie's detail page.

   If the movie fails to save, redisplay the `new` form populated with any valid data so the user can try again. Since the form is invalid, you'll need to set the response `status` to `:unprocessable_entity`.

   Hint:

   If the save fails, you don't want to redirect to the `new` action because all the valid form data will be lost (the form will be empty). Instead, you simply want to render the same template that the `new` action renders. That way, all the valid form data entered by the customer will show up in the form.

   Answer:

   ```
   def create
     @movie = Movie.new(movie_params)
     if @movie.save
       redirect_to @movie
     else
       render :new, status: :unprocessable_entity
     end
   end
   ```

2. Back in your browser, create a valid movie. The movie should get created and you should end up getting redirected to the new movie's detail page. That's the happy path!

3. Now try creating an *invalid* movie by entering a title, but leaving the rest of the form blank. You should see the form redisplayed. The valid title you entered should be populated in the title field. The other (invalid) fields should be highlighted in red and yellow.

4. How did those invalid fields get highlighted? Rails automatically wraps any form elements that contain invalid data in an HTML `div` with the `class` attribute set to `field_with_errors`. (Take a peek at the the page source. Rails won't mind.)

   And in the `custom.scss` file we have CSS rules that style `field_with_errors` accordingly. If you'd like to change the colors of invalid fields, just search for `field_with_errors` in the `custom.scss` file.

2. Display Validation Error Messages

At this point our controller is doing its job. Unfortunately, the user doesn't have a lot of clues as to *why* the movie wasn't created. To give them some actionable feedback, we need to display the actual error messages. Displaying information is a view's job, and in this case it makes sense to show the errors at the top of the form.

1. In the `_form` partial just inside the `form_with` block, start by simply displaying the errors as an English sentence just like we did earlier in the `console`.

   Answer:

   ```
   <%= form_with(model: movie) |f| %>
     <%= movie.errors.full_messages.to_sentence %>
     ...
   <% end %>
   ```

2. Then back in the browser try creating an invalid movie again and you should see the error messages displayed at the top of the form. It's not very *pretty* output, but at least we're communicating better with the user.

3. Instead of displaying the errors as a sentence, now we want to display the error messages in a neatly-formatted list with a bit of custom styling. We'll want to display errors this way on other forms we'll create later on in the course. Sounds like a great opportunity to practice what we learned earlier about partials!

   First, create an `app/views/shared` directory. Then inside that directory create an `_errors.html.erb` partial and paste in the following code:

   ```
   <% if object.errors.any? %>
     <section class="errors">
       <h2>
         Oops! Your form could not be saved.
       </h2>
       <h3>
         Please correct the following
         <%= pluralize(object.errors.size, "error") %>:
       </h3>
       <ul>
         <% object.errors.full_messages.each do |message| %>
           <li><%= message %></li>
         <% end %>
       </ul>
     </section>
   <% end %>
   ```

Notice that it assumes that a local variable named `object` references an ActiveRecord model object. And if that object has any errors, it displays the number of errors and iterates through all the error messages to generated a styled list. Groovy!

4. Then at the top of the `_form.html.erb` partial, render the `_errors.html.erb` partial (yup, partials can render other partials). When rendering the `_errors.html.erb` partial, you'll need to assign the `movie` object to the `object` variable.

   Hint:

   When calling a partial, we assign local variables using a hash. Each hash key will be the name of a local variable and the value will be that variable's value in the partial. So in this case the key needs to be `object` and the value needs to be the `movie` object. Since the `_errors.html.erb` partial lives in the `app/views/shared` directory, you'll need to refer to it as `shared/errors`.

   Answer:

   ```
   <%= form_with(model: movie) do |f| %>
     <%= render "shared/errors", object: movie %>
     ...
   <% end %>
   ```

5. Then try creating an invalid movie *again* in the browser. This time you should get a snazzy list of validation errors displayed at the top of the form. Don't care for our taste in colors? No problem. Just search for `errors` in the `custom.scss` file and customize to your heart's content!

3. Handle Errors During Update

OK, so we have error handling in place when creating new movies, but what about *editing* existing movies? The `update` action currently assumes that calling `update` always successfully updates the movie attributes in the database:

```
def update
  @movie = Movie.find(params[:id])
  @movie.update(movie_params)
  redirect_to @movie
end
```

Similar to calling `save`, calling `update` returns either `true` or `false` depending on whether the object is valid. So we need to add a conditional here, too.

1. Change the `update` action so that if the `update` call returns `true` (the model was updated), the action redirects to the movie's detail page.

   If the movie fails to update, redisplay the `edit` form populated with any valid data so the user can give it another go. Again, since the form is invalid, you'll need to set the response `status` to `:unprocessable_entity`.

   Hint:

   Again, if the update fails you don't want to redirect to the `edit` action. Instead, you want to render the same template that the `edit` action renders.

   Answer:

   ```
   def update
     @movie = Movie.find(params[:id])
     if @movie.update(movie_params)
       redirect_to @movie
     else
       render :edit, status: :unprocessable_entity
     end
   end
   ```

2. Then, back in your browser, edit a movie and enter valid information. You should get redirected to the movie's detail page and see the updated information. So far, so good!

3. Then try updating a movie by entering a blank title and a negative total gross value, but leaving the other fields intact. You should see the form redisplayed with the list of validation errors at the top. (Remember, the `edit` template uses the same `form` partial as the `new` template.) In the form itself, the title and total gross fields should be highlighted in red, but all the other fields should have their original values.

Solution

The full solution for this exercise is in the `handling-validation-errors` directory of the code bundle.

The Flash

Exercises

Objective

In the previous exercise we gave users feedback when they tried to enter invalid data via forms. But sometimes it's nice to give them short status messages even when things go well. Indeed, it's often the case that web apps need to flash a message up on the page after a specific action takes place. We can easily do that with something Rails calls a *flash* (shocking, we know).

In this exercise our goal is to flash the following three messages after their respective actions:

- "Movie successfully updated!"
- "Movie successfully created!"
- "Movie successfully deleted!"

We'll make quick work of this. We'll have it done in a flash! (OK, we'll stop now.) The first one takes a couple extra steps just to get everything set up, but then it's easy-peasy from there.

1. Flash on Update

First, when a movie has been successfully updated we want to give the user some confirmation that it happened. You know, something upbeat like "Movie successfully updated!".

1. In the `update` action, use the `flash` object directly to assign the status message to the `:notice` key. Remember, the `flash` object acts just like a Ruby hash, so it needs a key (an arbitrary type, in this case `:notice`) and a value (the message you want displayed).

   Answer:

   ```
   def update
     @movie = Movie.find(params[:id])
     if @movie.update(movie_params)
       flash[:notice] = "Movie successfully updated!"
       redirect_to @movie
     else
       render :edit, status: :unprocessable_entity
     end
   end
   ```

   Now, back in your browser, edit a movie and hit submit.

2. Hey, where's our confirmation? Unfortunately, the flash message isn't displayed anywhere. :-(

   We set the flash message in the controller action, but now we need to actually *display* the flash message. That sounds like something a view template should do. But which view template? The `update` action ends up redirecting to the `show` action which then renders the `show` template. So displaying the flash in the `show` template would work, but it's a short-sighted solution.

   Ideally, we want to be able to set a flash message in *any* controller action and have it displayed at the top of the resulting page. For that reason, it's better to render the flash messages in the application-wide layout file. Say, that sounds familiar!

3. In the `application.html.erb` layout file, render the flash message just inside the `content` div. Don't worry about stylin' it, just get something working.

   Hint:

   The `flash` object we used in the `update` action is also available in the layout (or any view). Use the `flash` object to display the message associated with the `:notice` key which we assigned in the `update` action.

   Answer:

   ```
   <div class="content">
     <%= flash[:notice] %>

     <%= yield %>
   </div>
   ```

4. Now edit a movie again, and this time you should see the flash message displayed at the top of the movie's detail page. Reload the page and the **flash should disappear**.

   That makes sense. We don't want the flash message showing up every time we view a movie's detail page. We only want the message to appear after the user has updated the movie. The flash message (the value of the hash) is cleared after every request. It's like a flash in the pan! (Sorry, couldn't resist.)

5. Want to add a little color and styling to the flash message? No problem. Click the answer for a version that displays the flash notice message in a styled paragraph if a notice message exists. It also displays any alert flash messages.

   Answer:

   ```
   <% if flash[:notice] %>
     <div class="flash notice">
       <%= flash[:notice] %>
     </div>
   <% end %>

   <% if flash[:alert] %>
     <div class="flash alert">
       <%= flash[:alert] %>
     </div>
   <% end %>
   ```

   Now after editing a movie you'll see a greenish, centered flash message at the top. It's green because we have CSS rules for flashes. Green's not your favorite flash color? No worries. Just search for `flash` in the `custom.scss` file. You'll find specific rules for `notice` and `alert` messages. Feel free to change the colors!

6. To keep everything in the layout file at basically the same conceptual level, let's put all the flash stuff in a partial, too.

   First, in the `app/views/layouts` directory, create a `_flash.html.erb` partial file. Then cut the flash code out of the `application.html.erb` layout file and paste it into the new `_flash.html.erb` partial.

   Then, back in the layout file, render the `layouts/flash` partial.

   Answer:

   ```
   <div class="content">
     <%= render "layouts/flash" %>

     <%= yield %>
   </div>
   ```

7. Finally, setting notice flashes is so common that Rails provides a shortcut. You can pass an option to `redirect_to` to automatically set the appropriate flash before the redirection happens.

   Change your `update` action to use the shortcut like this:

```
def update
  @movie = Movie.find(params[:id])
  if @movie.update(movie_params)
    redirect_to @movie, notice: "Movie successfully updated!"
  else
    render :edit, status: :unprocessable_entity
  end
end
```

   Give it a try in the browser. Very flashy indeed!

2. Flash on Create

Next we want to flash "Movie successfully created!" when a movie has been successfully created. Given what you've learned, you should be able to do this blindfolded!

1. In the `create` action, assign the notice message to the `flash` object.

   Answer:

```
def create
  @movie = Movie.new(movie_params)
  if @movie.save
    redirect_to @movie, notice: "Movie successfully created!"
  else
    render :new, status: :unprocessable_entity
  end
end
```

2. In the browser, create a movie and the flash message should glow green at the top of the resulting page.

3. We didn't have to change a view template for this to work. Why?

   Answer:

   Flash messages are being rendered in the layout file, so any action can set a flash and it will get picked up by the layout.

3. Flash on Destroy

Try destroying a movie in the browser and you'll notice there's no visual feedback that it actually happened. Let's wrap up this exercise by fixing that, and adding a red twist...

1. Change the `destroy` action to display a flash message when a movie is destroyed. This time make it an `alert` instead of a `notice`, since destroying stuff can be kinda alarming.

   Answer:

```
def destroy
  @movie = Movie.find(params[:id])
  @movie.destroy
  redirect_to movies_url, status: :see_other,
    alert: "Movie successfully deleted!"
end
```

2. Remember that your `_flash` partial displays both notice and alert messages, so you don't need to make any changes to display alert messages.

3. Double-check that it works in the browser by destroying a movie. You should see a reddish flash as a confirmation.

4. If you want to re-populate the database with all the example movies, use:

```
rails db:seed:replant
```

   This task removes all the data from the database tables and "replants" the database with the seed data in `db/seeds.rb`.

Solution

The full solution for this exercise is in the `flash` directory of the code bundle.

Bonus Round

Flexible Flashes

This solution works good, but it has a limitation: the layout file currently only displays `notice` and `alert` flashes. You might want something a bit more flexible that can display arbitrary types of flash messages in a more generic way. It's a nice-to-have thing, so we went ahead and cooked up a solution for you.

Change your `_flash.html.erb` partial file to display flash messages like this:

```
<% flash.each do |type, message| %>
  <%= content_tag(:div, message, class: "flash #{type}") %>
<% end %>
```

Don't let this throw you. It just iterates through all the flash keys and values, and uses the `content_tag` helper to generate a styled `div` tag for each message. That way, whatever you put in the flash will get displayed with a little style. Notice that by using an iterator we don't need a conditional because the iterator only runs for the keys

that are in the flash.

Make sure to remove the `div` tags we added earlier to render the `notice` and `alert` flashes separately. Otherwise you'll get duplicate flash messages!

Give it a whirl in the browser just to make sure the same green and red flashes show up!

Custom Flash Types

By default, Rails supports setting the `:notice` and `:alert` flash types when calling the `redirect_to` method. But sometimes you want to conveniently set a *custom* flash type when redirecting. For example, suppose you wanted to set a `:danger` flash type like so:

```
redirect_to movies_url, danger: "Danger, Will Robinson!"
```

To do that, you'll need to register the `:danger` flash type:

1. Add the following line inside of the `ApplicationController` class, which is defined in the `app/controllers/application_controller.rb` file:

   ```
   class ApplicationController < ActionController::Base
     add_flash_types(:danger)
   end
   ```

   The `ApplicationController` is the base (parent) class that all other controllers inherit from (subclass), so anything you put in here applies to all controllers.

2. As an example of how you might use your custom flash type, comment out the code currently in the `destroy` action in your `MoviesController` and instead just redirect with a danger message, like so:

   ```
   def destroy
     ...
     redirect_to movies_url, status: :see_other,
       danger: "I'm sorry, Dave, I'm afraid I can't do that!"
   end
   ```

3. Now try to delete a movie and you should see the (unstyled) danger message flashed at the top of the page. Before the redirect happened, Rails automatically assigned the message to `flash[:danger]`. Then after the redirect happened the message got displayed because your flexible `_flash.html.erb` partial iterates through *all* the flash keys and values. You saw that coming, right?

4. Finally, add some style to danger messages. You'll need to define a CSS rule in `custom.scss` that matches the CSS classes `flash` and `danger`.

   Answer:

   ```
   .flash {
     /* existing styles here */

     &.danger {
       background-color: #FFA715;
     }
   }
   ```

5. Remember to revert your `destroy` action back to its original form before moving on!

Wrap Up

News flash: our app is now giving lots of good feedback! Flashes are kinda fun, but don't go overboard with them. It's typical to show flash messages to confirm the success of the `create`, `update`, and `destroy` actions as we've done in this exercise. Flashes are also useful for things like a "Welcome back!" message after a user has logged in and a "So long!" message when they log out. But try not to get too flashy, ok?

Now that we have the movie side of the house in good shape, it's time to shift gears and work on adding reviews for each movie. That's up next! But first, you deserve a break, and you knew we couldn't resist recommending this movie.

Dive Deeper

To learn more about the flash, refer to "The Flash" section in the Rails Guide: Action Controller Overview.

One-to-Many: belongs_to

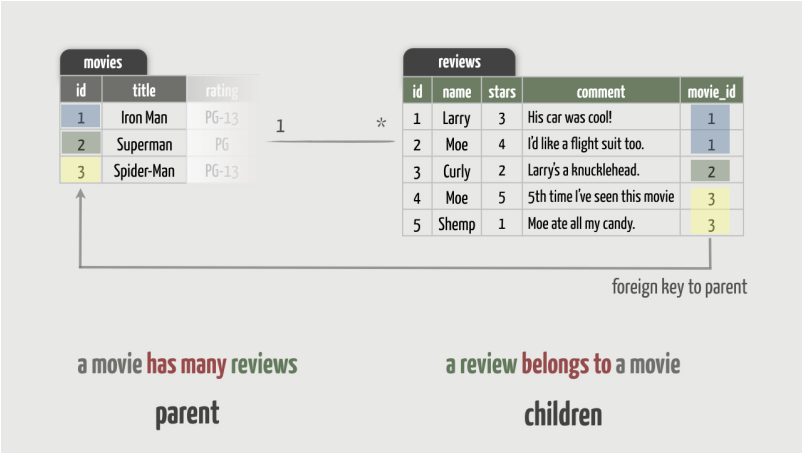
Exercises

Objective

The movie-related features of the app are in really good shape! Now it's time to shift our focus to a new set of features. Most respectable movie apps allow users to write reviews for movies. Think of it as your chance to give your honest opinion about the movies in your app!

Let's think about the relationship between movies and reviews. Any particular movie is watched by many people, so it makes sense that a movie could potentially have many reviews. Exactly how many reviews will be written for a given movie? We can't predict the number. This ambiguity presents a bit of a challenge when it comes to storing reviews in the database. Clearly we can't just add a new `review_comment` column to the `movies` table because then a movie could only have one review. Instead, to accommodate many reviews per movie, we'll need to put the reviews in a new `reviews` table. Then we'll need to somehow connect each review in the `reviews` table to a particular movie in the `movies` table.

Visually, here's what that will look like in the database:



In database terms, this is called a *one-to-many* relationship. A movie potentially has many reviews. On the flip side, a review belongs to a single movie. To join the two tables together, each row of the `reviews` table holds a reference to a movie in the `movies` table. This reference is held in the `movie_id` column. The number in that column is the numeric `id` that corresponds to the associated row in the `movies` table. In database terms, the `movie_id` column is called a *foreign key*.

One-to-many relationships between application data are so common that Rails makes it easy to work with them at the model layer. It does this by relying on a simple set of database conventions:

- The object on the "many" (child) end of the relationship needs a foreign key that points to the object on the "one" (parent) end of the relationship.
- The name of the foreign key needs to be the name of the "one" (parent) object, followed by `_id` (e.g. `movie_id`).

Following those conventions in the database, Rails can glean a lot about the linkage between the tables. But it needs a little help from us. We need to tell Rails what kind of relationship we want between the models. In our case, a review *belongs to* one movie and a movie *has many* reviews. Rails refers to these model relationships as *associations*.

So to start creating the one-to-many association between movies and reviews,we'll need to do two things:

1. Create a new `reviews` database table and a corresponding `Review` model
2. Declare that a `Review` belongs to a `Movie`.

Of course, we'll ultimately need the reverse association (a movie has many reviews) and a web interface for writing reviews. But let's not get ahead of ourselves!

1. Create the Review Resource

First we need to create a new `reviews` database table with the following fields and types:

name	type
name	string
stars	integer
comment	text
movie_id	integer

We'll also need a corresponding `Review` model to access that table. We learned earlier how to generate a model and migration in one fell swoop using the model generator. We could do that again here, but let's take it a step further. Thinking ahead, we'll also need a `ReviewsController` to handle creating reviews. And later on we'll also need some basic routes for interacting with reviews via the web.

Taken together, Rails calls all of this stuff a *resource*. Creating new resources is so common that Rails provides a handy *resource generator*.

1. The resource generator takes the same options as the model generator: the model name followed by a list of fields and types separated by colons. So start by generating a resource named `review` (singular) with the fields and types listed above, like so:

   ```
   rails g resource review name:string stars:integer comment:text movie:references
   ```

   This is a long command that's easy to goof up, so here's a tip: If you end up running a generator with a misspelled field name or any other error, you can delete all the generated files by running `rails destroy` and give it the exact same options you used when running the generator. For example, to delete all the files we just generated, run:

   ```
   rails destroy resource review name:string stars:integer comment:text movie:references
   ```

2. We'll look at everything that got generated in due time. For now, open the generated migration file and you should see the following:

   ```
   class CreateReviews < ActiveRecord::Migration[7.0]
     def change
       create_table :reviews do |t|
         t.string :name
         t.integer :stars
         t.text :comment
         t.references :movie, null: false, foreign_key: true

         t.timestamps
       end
     end
   end
   ```

That's pretty much what we expected. The only surprise is this line:

```
t.references :movie, null: false, foreign_key: true
```

That's simply a shortcut that ends up adding a column named `movie_id`. It's equivalent to doing this in the migration:

```
t.integer :movie_id
```

Using `references` just makes it a bit more readable. Notice that the generator also tacks on the `null: false` option which effectively adds a validation that requires a review to have a non-null `movie_id` before the review can be saved to the database.

It also tacks on the `foreign_key: true` option to add a foreign key constraint to guarantee referential integrity. In general, a foreign key constraint prevents invalid values from being inserted into the foreign key column. In this instance, the value of the `movie_id` column has to be a primary key contained in the `movies` database table.

3. Now go ahead and run the migration to create the `reviews` table.

   Answer:

   ```
   rails db:migrate
   ```

4. Finally, hop into a `console` session and instantiate a new `Review` object:

   ```
   >> r = Review.new
   ```

   Then to see what's in the `r` object, type:

   ```
   >> r
   ```

   And you should get the following output:

   ```
   #<Review:0x000000013a87c860
    id: nil,
    name: nil,
    stars: nil,
    comment: nil,
    movie_id: nil,
    created_at: nil,
    updated_at: nil>
   ```

   That tells us the `Review` model is connected to the `reviews` database table. Notice that the `Review` model has attributes for every column in the `reviews` table, including the `movie_id` column which will contain foreign keys pointing to movies.

Great! Now our database schema follows the conventions.

2. Declare the belongs_to Association

Applying the migration created the `reviews` database table with the `movie_id` foreign key column. So far, so good. But remember, the `Review` model needs to declare that it belongs to a `Movie`.

Open the generated `app/models/review.rb` file and you'll see that it already has that `belongs_to` association declared:

```
class Review < ApplicationRecord
  belongs_to :movie
end
```

Cool—nothing for us to do here! Rails knew to add this declaration for us because we used `movie:references` when generating the `review` resource.

Notice that the `belongs_to` declaration references the singular form of the parent (movie). This declaration tells Rails that the `movie_id` database column holds a reference to a `Movie` object. Rails also dynamically defines methods for reading and writing a review's related movie.

3. Use the belongs_to Association

Now let's see how the `belongs_to` association works by using it in the `console`.

1. First, instantiate a new `Review` object in memory by calling the `new` method on the `Review` class. Assign it to a `review` variable. Go ahead and set the name of the reviewer, the number of stars, and a brief comment. Assign it to a `review` variable.

   Answer:

   ```
   >> review = Review.new(name: "Larry", stars: 5, comment: "Awesome!")
   ```

2. Now print out the value of the review's `movie_id` attribute.

   Answer:

   ```
   >> review.movie_id
   ```

   The foreign key should be `nil` because the review isn't yet associated with a movie.

3. OK, so find an existing `Movie` that you want this review to be associated with. Assign that movie to a `movie` variable.

   Answer:

   ```
   >> movie = Movie.find_by(title: "Captain Marvel")
   ```

4. Now associate the review with the movie by assigning the `movie` object to the review's `movie` attribute.

Hint:

When you declared `belongs_to :movie` in the `Review` model, Rails dynamically defined a `movie` attribute. Assigning a `Movie` object to that attribute simply assigns the movie's primary key (`id`) to the review's `movie_id` foreign key. Reading the `movie` attribute from a `Review` object returns the associated `Movie` object.

Answer:

```
>> review.movie = movie
```

5. To verify that the association has been made, first print the value of the review's `movie_id` attribute again.

   Answer:

   ```
   >> review.movie_id
   ```

   This time you should get the primary key of the movie object. The `movie_id` foreign key was automatically set when the movie was assigned to the review.

   Then access the review's `movie` attribute.

   Answer:

   ```
   >> review.movie
   ```

   You should get back the `Movie` object (the parent) that's associated with the `Review` object (the child). Behind the scenes, unless the movie has already been loaded from the database, Rails ran a database query to select the movie in the `movies` table that has an `id` matching the `movie_id` value for this particular review.

6. We haven't yet saved the review in the database, so go ahead and do that next.

   Answer:

   ```
   >> review.save
   ```

7. Next, initialize a second review for the same movie in one fell swoop by calling `new` with a hash of review attributes, including the movie.

   Hint:

   Since `movie` is an attribute of a review, you can assign it any movie object and the `movie_id` foreign key will be automatically set to point to that movie in the `movies` database table.

   Answer:

   ```
   >> review = Review.new(name: "Daisy", stars: 4, comment: "Great!", movie: movie)
   ```

8. Then print the value of the new review's `movie_id` attribute.

   Answer:

   ```
   >> review.movie_id
   ```

   You should get the primary key of the `movie` object you assigned to the `movie` attribute. In other words, the `movie_id` foreign key was automatically set to point to that `movie` object. Convenient!

9. Remember, since we initialized the review using `new`, it won't be saved to the database until you call `save`.

   Answer:

   ```
   >> review.save
   ```

Great—now we know the `belongs_to` association is working as expected!

Solution

The full solution for this exercise is in the `one-to-many-belongs-to` directory of the code bundle.

Wrap Up

That takes care of one side of the one-to-many association: a review belongs to a movie. In the next section we'll tackle the other side: a movie has many reviews.

One-to-Many: has_many

Exercises

Objective

The one-to-many association currently can only be traversed in one direction. We can ask a review for its associated movie, but we can't ask a movie for all of its reviews. Bi-directional associations aren't required; you only need to define associations in the direction you intend to use them. However, in our case, given a `Movie` object we want to be able to get all of its associated reviews and also create reviews using a movie as the parent object.

To finish creating the one-to-many association between movies and reviews, we need to declare that a `Movie` has many `Reviews`. We'll also add reasonable validations in the `Review` model.

1. Declare the has_many Association

If you look in the `Movie` model you'll discover that it doesn't have a reciprocal association to its reviews. The generator added the `belongs_to` association to the `Review` model, but the generator didn't add an association to the `Movie` model. That's because the generator doesn't know what kind of association we want on that side of the relationship. So we have to explicitly declare the specific association we want in the `Movie` model. In particular, a movie *has many* reviews.

To make that so, update the `Movie` model (the parent) to have a `has_many` association with reviews (the plural form of the child).

Answer:

```
class Movie < ApplicationRecord
  has_many :reviews

  # existing code
end
```

This declaration tells Rails to expect a `movie_id` foreign key column in the table wrapped by the `Review` model, which by convention will be the `reviews` table. Rails also dynamically defines methods for accessing a movie's related reviews.

And with that one-liner, we now have a bi-directional association between a movie and its reviews!

2. Use the has_many Association

Let's give this association a whirl in the `console`...

1. First, make sure to load the latest version of your code by using `reload!`:

   ```
   >> reload!
   ```

2. Then find the movie you added reviews for in the previous exercise.

   Answer:

   ```
   >> movie = Movie.find_by(title: "Captain Marvel")
   ```

3. OK, now ask the movie for its reviews!

   Hint:

   When you declared `has_many :reviews` in the `Movie` model, Rails dynamically defined a `reviews` instance method. Calling that method on a `Movie` object returns an array of `Review` objects for that particular movie. If the movie doesn't have any reviews, the array will be empty.

   Answer:

   ```
   >> movie.reviews
   ```

   You should see the database being queried and get back an array that contains the reviews for that movie. Behind the scenes, Rails ran a database query to select all the reviews in the `reviews` table that have a `movie_id` value matching the `id` for this particular movie.

4. Next, create another review for the movie. This time use the parent `movie` object to create the review. Using the parent object to create child objects is preferred because the child will automatically be associated with the parent.

   Hint:

   Initialize the review using `movie.reviews.new` and pass it a hash of review attributes. This automatically sets the review's `movie_id` foreign key to point to the `movie` object.

   Answer:

   ```
   >> review = movie.reviews.new(name: "Moe", stars: 3, comment: "Spilled my popcorn!")
   ```

5. Then print the value of the new review's `movie_id` attribute.

   Answer:

   ```
   >> review.movie_id
   ```

   You should get the primary key of the `movie` object you used as the parent object. In other words, because we initialized the review using the `reviews` association of the `movie` object, the `movie_id` foreign key was automatically set to point to the `movie` object. Think of it this way: each child is created with a built-in reference back to their parent!

6. Remember, since we initialized the review using `new`, it won't be saved to the database until you call `save`.

   Answer:

   ```
   >> review.save
   ```

7. Now the movie should have *three* reviews. Use the `movie` object to query the database for its reviews and get the number of reviews.

   Answer:

   ```
   >> movie.reviews
   >> movie.reviews.size
   ```

Nicely done!

3. Cascade Deletes

By default, when you delete the parent of a one-to-many relationship, the child rows of that parent remain in the database. That means if we delete a movie then its reviews will be orphaned. (Poor little reviews!) They'll no longer be associated with a movie. And that leads to data integrity issues because a review only has meaning when it's associated with a movie.

So we need to arrange things so that when we delete a movie (the parent), its reviews (the children) are automatically deleted as well. However, we don't want the reverse to happen. Deleting a review should not delete the movie because the movie still has meaning on its own. Plus, the movie likely has other reviews associated with it.

1. Change the `has_many` side of the association to destroy all reviews when a movie is destroyed.

   Hint:

   In the `Movie` model, use the `dependent:` option on the `has_many` association. Set the option to `:destroy`. This causes the `destroy` method to be called on each associated review object when the movie itself is destroyed.

   Answer:

   ```
   class Movie < ApplicationRecord
     has_many :reviews, dependent: :destroy

     # existing code
   end
   ```

2. To see this in action, jump into a new `console` session or call `reload!` in your existing session. Then find a movie that has reviews and destroy it.

   Answer:

   ```
   >> reload!

   >> movie = Movie.find_by(title: "Captain Marvel")

   >> movie.destroy
   ```

   You should see one `DELETE FROM "reviews"` line printed for each associated review that was deleted. That tells you the cascading delete is working!

3. Finally, to resurrect that movie, you can copy its attributes from the `db/seeds.rb` file and pass those attributes to `Movie.create!` in the console as we did in the video. Then if you want some extra practice, recreate the reviews that were associated with that movie.

4. Declare Review Validations

Remember when we said that you'll need to think about validations every time you change the database or add new models? Well, here's a perfect scenario to put that into practice. Similar to the way we declared validations in the `Movie` model, we also need some reasonable validations in our new `Review` model. Here's the set of requirements:

1. A value for the `name` field must be present.

   Answer:

   ```
   validates :name, presence: true
   ```

2. The `comment` field must have a minimum of 4 characters.

   Answer:

   ```
   validates :comment, length: { minimum: 4 }
   ```

3. Lastly, the `stars` field should have a value between 1 and 5. Similar to how we used a `RATINGS` constant for the movie rating validation, use a `STARS` constant for review star validation. Override the default error message for this validation with the custom message "must be between 1 and 5".

   Hint:

   Create a `STARS` constant that points to an array of numbers 1 through 5. Then use a built-in validation to validate whether a star value is included (hint!) in the `STARS` array. To add a custom message, use the `message` option.

   Answer:

   ```
   STARS = [1, 2, 3, 4, 5]

   validates :stars, inclusion: {
     in: STARS,
     message: "must be between 1 and 5"
   }
   ```

4. Now that you have validations declared in the `Review` model, use what you learned earlier about validations to check the validations in the `console`. For example, instantiate a new `Review` object without any attributes and check that it's invalid and has the appropriate error messages. Then put the review into a valid state, with an associated movie, and save it to the database.

   Answer:

   ```
   >> movie = Movie.find_by(title: "Captain Marvel")

   >> review = movie.reviews.new

   >> review.save
   => false

   >> review.errors.full_messages
   => ["Name can't be blank",
       "Comment is too short (minimum is 4 characters)",
       "Stars must be between 1 and 5"]
   ```

```
>> review.name = "Lucy"
>> review.stars = 4
>> review.comment = "Fierce and funny!"

>> review.save
=> true
```

Solution

The full solution for this exercise is in the `one-to-many-has-many` directory of the code bundle.

Wrap Up

One of the most powerful features of Active Record is the ability to form associations between different models. In the case of our `Movie` model, each movie potentially has many reviews. Initially, you'd probably assume that linking the two models would involve complicated database gymnastics. But by following Rails conventions, you now know that it's actually quite easy to do with a couple lines of Ruby code. Based on the type of association, Rails then dynamically generates methods to navigate and manage the associations. And that takes a lot of tedious grunt work off our hands!

Now that we have the associations working in the models, in the next section we'll create a page that lists all the reviews we have so far for each movie.

Dive Deeper

In addition to one-to-many associations, Rails also supports *many-to-many* and *through* associations that are commonly used in more sophisticated apps. Don't worry: we'll learn about those associations in upcoming sections!

To learn more about associations, refer to the Rails Guide: Active Record Associations. You might also want to review the documentation for the belongs_to and has_many methods.

You might be tempted to think that class-level declarations such as `has_many`, `belongs_to`, and other so-called "macros" are magical aspects of Rails. Indeed, they are powerful, but they aren't magic. The Ruby language makes programming in this declarative style fairly straightforward. In this short video, we recreate a simplified version of the `has_many` declaration so you understand how it works and can then apply this same powerful technique in your own code!

One-to-Many: Nested Resources

Exercises

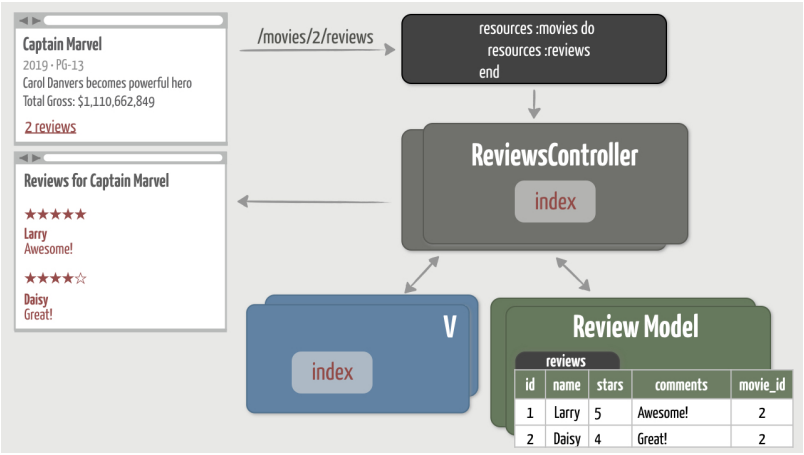
Objective

We're off to a good start! We have a one-to-many association between the `Movie` and `Review` models, and that relationship is reflected in the database structure. In fact, in previous exercises we used the `console` to create a few reviews in the database. We'll get to how to write reviews in the browser in the next section. But first, since we already have a few reviews in the database, let's go ahead and add a page that lists all the reviews for a movie. Inquiring minds want to know: What are people saying about "Captain Marvel"?

To answer that question, we need to:

1. Nest the *review* resource inside of a *movie* resource so that the router recognizes requests for `/movies/2/reviews`, for example.
2. Define an `index` action in the `ReviewsController` that fetches all the reviews for the movie specified in the URL.
3. Create a corresponding `index.html.erb` template that generates an HTML list of the movie's reviews.
4. Generate a "Reviews" link on the movie `show` page so it's easy to navigate to the list of reviews for any movie.

Visually, here's what we want to do:



The steps we'll take to list reviews are similiar to the steps we took earlier to list movies. However, there are two important differences.

First, we're now dealing with review resources rather than movies. So where do we put the code to list reviews? The `index` action in our `MoviesController` is already responsible for listing movies. We don't want to conflate things by also making it responsible for listing reviews. By convention, each controller should only concern itself

with a single resource. When our application grows to include an additional resource such as reviews, it's time to introduce a new controller. So we'll put the code to list reviews in the `index` action of the `ReviewsController`.

The second difference is in how we interact with reviews. Any time we list reviews it will always be in the context of a movie. In other words, we only want to see the reviews for a specific movie. There's no situation in which a user wants to see a list of all the reviews in the database outside the context of a movie. So the URL for listing reviews needs to specify the associated movie. To impose that hierarchy in the URL, we'll use *nested resources*. By nesting the `reviews` resource inside of the `movies` resource, every review-related URL will require a movie ID. For example, accessing the URL `/movies/2/reviews` should list the reviews that belong to the movie with an ID of 2. So in effect we're mirroring the one-to-many association between a movie and its reviews in the URL.

Enough with the theory—let's jump right into it!

1. Nest the Review Resource

As is customary by now, let's start with the URL we want for listing reviews and work our way through the errors...

1. Suppose we want to see all the reviews for "Captain Marvel" which has an ID of 2. Browse to http://localhost:3000/movies/2/reviews and you should get the following error:

   ```
   Routing Error

   No route matches [GET] "/movies/2/reviews"
   ```

   No surprise there.

2. Crack open the `config/routes.rb` file and you'll notice that the resource generator we ran in the previous exercise added this line:

   ```
   resources :reviews
   ```

   That's a polite gesture, but it assumes that reviews are a top-level resource. Instead, we want to mirror the parent-child relationship in the routes by *nesting* reviews inside of movies.

   To do that, on the *movies* resource create a block with `do` and `end` and move the *reviews* resource inside of the block, like so:

   ```
   resources :movies do
     resources :reviews
   end
   ```

   With this minor change, the `reviews` resource will now always be nested within `movies` resources in the URLs.

3. To see how the nesting changes the routes and URLs, reload http://localhost:3000/rails/info/routes. You should see the following **eight** new routes for reviews (we've left out the existing routes for movies):

   ```
   Helper                  HTTP Verb        Path                                        Controller#Action
   movie_reviews_path         GET       /movies/:movie_id/reviews(.:format)            reviews#index
                              POST      /movies/:movie_id/reviews(.:format)            reviews#create
   new_movie_review_path      GET       /movies/:movie_id/reviews/new(.:format)        reviews#new
   edit_movie_review_path     GET       /movies/:movie_id/reviews/:id/edit(.:format)   reviews#edit
   movie_review_path          GET       /movies/:movie_id/reviews/:id(.:format)        reviews#show
                              PATCH     /movies/:movie_id/reviews/:id(.:format)        reviews#update
                              PUT       /movies/:movie_id/reviews/:id(.:format)        reviews#update
                              DELETE    /movies/:movie_id/reviews/:id(.:format)        reviews#destroy
   ```

   We saw similar routes for accessing movies, but these nested routes have two crucial differences.

   First notice that *every* path to the `ReviewsController` requires a `movie_id` parameter in the URL for the route to match! In other words, all the routes for accessing reviews are nested under a specific movie. The `:movie_id` parameter gets automatically filled in with the movie ID that's embedded in the URL. When accessing a specific review, such as deleting a review, the route also requires an `id` in the URL which identifies the ID of the review to delete. Second, notice that with nested routes the names of the route helper methods include the singular name of the parent, `movie` in this case. For example, the route helper to generate a URL to list the reviews for a movie is `movie_reviews_path`. This will come in handy a bit later.

   So how do you remember the names of the parameters and helpers? You don't. Or at least it takes a while to internalize these conventions. In the meantime, just look at the routes! (Yup, even the pros do this.)

4. Back in your browser, refresh the page (you're still accessing http://localhost:3000/movies/2/reviews). This time the router should recognize the URL, and you should get the following all-too-familiar error:

   ```
   Unknown action

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

   Perfect. According to the routes, the request should indeed go to the `index` action of the `ReviewsController`. We already have the controller, but we don't have the action. Remember that the *resource generator* we used in the previous exercise went ahead and created an empty `ReviewsController` class for us. That's a subtle way of Rails saying that each resource should have its own controller. :-)

That error prompts us to take the next step. Think about how you might implement the `index` action before moving on.

2. Create the Index Action and Template

Got a plan in mind? Here's a nudge in the right direction. First, because the `index` action always runs in the context of a nested route, the action needs to find the movie that has the ID supplied in the URL. Then the action needs to fetch all the reviews that belong to that movie. Finally, the action needs to render a template that lists that movie's reviews.

1. First up, define an `index` action in the `ReviewsController`. Begin the action by finding the movie and assigning it to a `@movie` variable. Then use that variable to fetch all of the movie's reviews into an array and assign it to a `@reviews` variable.

   Hint:

Remember, because we're using nested routes, all incoming URLs to the `ReviewsController` will have a `:movie_id` parameter that's filled in with the ID of the movie specified in the URL. You'll need to use that parameter to find the corresponding movie in the database. Then to get all the reviews that belong to that movie, use the `@movie.reviews` association. By using this association method, you ensure that you'll only get back the reviews that are associated with the movie.

Answer:

```
def index
  @movie = Movie.find(params[:movie_id])
  @reviews = @movie.reviews
end
```

2. Refresh the page (you're still accessing http://localhost:3000/movies/2/reviews) and you should have anticipated getting this error:

```
No template for interactive request

ReviewsController#index is missing a template...
```

Hello, old friend! Notice this time it's looking for the template in the `app/views/reviews` directory because we ran the `index` action in the `ReviewsController`. The convention is that a controller's templates live in a subdirectory of `app/views` that matches the controller name.

3. Following the error, create a file named `index.html.erb` in the `app/views/reviews` directory.

In the `index.html.erb` template, for now just use the `@movie` variable to generate header text such as "Reviews for Iron Man". If you want to get fancy, link the movie's title back to the movie detail page.

Answer:

```
<h1>Reviews for <%= link_to @movie.title, @movie %></h1>
```

4. Refresh the page as a quick smoke test. You know it's good to go if the header text reflects the movie specified in the URL. That tells us the `index` action is picking up the correct movie. We often take these incremental steps just as a confidence boost. (It makes problem-solving a lot easier, too.)

5. Carry on by listing all the reviews in the `@reviews` variable. For each review, display the following attributes:

   - number of stars (pluralized, please!)
   - reviewer name
   - comment

Start with a simple list and don't worrying about styling. Just get a quick win. Here's the output you're aiming for, assuming three reviews:

```
5 stars by Larry: Awesome!
4 stars by Daisy: Great!
3 stars by Moe: Spilled my popcorn!
```

Hint:

Iterate through the `@reviews` array, and for each review generate an HTML list item. If you need a refresher on how to display a list of things, go ahead and look at the `app/views/movies/index` template.

Answer:

```
<ul>
  <% @reviews.each do |review| %>
    <li>
      <%= pluralize(review.stars, 'star') %> by <%= review.name %>:
      <%= review.comment %>
    </li>
  <% end %>
</ul>
```

6. Refresh the page and you should see a list of all the reviews for "Captain Marvel", for example. If you don't see any reviews, drop back into the console and make sure reviews exist for the movie.

7. Now for a quick finishing move. Look at the examples of the built-in time_ago_in_words helper. Then use that helper to display how long ago the review was created, such as "about 5 minutes ago".

Answer:

```
<%= time_ago_in_words(review.created_at) %> ago
```

8. Finally, to style up the listing a bit, click the answer for a version that takes advantage of our CSS rules. Note that for this to work, the `<ul>` tag needs to have the CSS `class` set to `reviews`.

Answer:

```
<ul class="reviews">
  <% @reviews.each do |review| %>
    <li>
      <%= pluralize(review.stars, 'star') %>
      <p>
        <%= review.name %>
        <span class="date">
          <%= time_ago_in_words(review.created_at) %> ago
        </span>
      </p>
      <p>
        <%= review.comment %>
      </p>
    </li>
  <% end %>
</ul>
```

WooHoo! Now you can see the reviews for any movie!

3. Generate a "Reviews" Link

Finally, to make it easy to navigate to a movie's list of reviews, we'll generate a "Reviews" link on the movie `show` page.

Before looking at the steps below, can you guess which route helper method we'll use to generate that link? We almost gave it away earlier. Noodle on it for a minute and give it a try yourself. Then follow the steps below if you need some help.

1. First, identify the route helper method that generates a link that goes to the `index` action of the `ReviewsController`. The easiest way to find the helper method is by looking at http://localhost:3000/rails/info/routes.

   Answer:

   ```
   movie_reviews_path
   ```

2. Now use that route helper method to generate a "Reviews" link on the movie `show` template. Put the link under the existing `h2`, in a `div` with a class of `reviews`. Remember, since the route has one placeholder (`:movie_id`), the route helper method needs to get passed a movie id.

   Answer:

   ```
   <div class="details">
     <h1><%= @movie.title %></h1>
     <h2>
       <%= year_of(@movie) %> &bull; <%= @movie.rating %>
     </h2>
     <div class="reviews">
       <%= link_to "Reviews", movie_reviews_path(@movie) %>
     </div>
     <p>
       <%= @movie.description %>
     </p>
     <table>
       ...
     </table>
   </div>
   ```

3. Then, back in your browser, navigate to a movie show page and you should see a tempting "Reviews" link. Go ahead and click it to view all the movie's reviews!

   If you get an error, read it through carefully. It's likely that double-checking the name of the route helper method will lead you to the solution. :-)

4. As a well-deserved bonus, change the "Reviews" link to include the number of reviews. For example, if the movie has 3 reviews, then generate the link "3 Reviews" (properly pluralized, of course).

   Answer:

   ```
   <%= link_to pluralize(@movie.reviews.size, "review"),
               movie_reviews_path(@movie) %>
   ```

High fives all around!

Solution

The full solution for this exercise is in the `one-to-many-nested-resources` directory of the code bundle.

Bonus Round

Want to display a review's star rating as a series of partially filled in stars like you see on popular movie sites? We've got you covered!

1. First we need a way to convert a review's number of stars to a percentage. That sounds like something the `Review` model should be responsible for calculating. So define the following `stars_as_percent` instance method in the `Review` model:

   ```
   class Review < ApplicationRecord
     # existing code

     def stars_as_percent
       (stars / 5.0) * 100.0
     end
   end
   ```

   Nothing tricky here. Just some basic math. Remember that the `Review` model has a `stars` attribute, so this method simply converts the value of that attribute to a percentage.

2. Then we need a way to render a series of partially filled in stars based on the percentage returned by the `stars_as_percent` method. To do that, in the `app/views/shared` directory create a partial file named `_stars.html.erb` and paste in the following:

   ```
   <div class="star-rating">
     <div class="back-stars">
       <span>★</span>
       <span>★</span>
       <span>★</span>
       <span>★</span>
       <span>★</span>
       <div class="front-stars" style="width: <%= percent %>%">
         <span>★</span>
         <span>★</span>
         <span>★</span>
         <span>★</span>
         <span>★</span>
   ```

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

Don't worry about the details here. Basically, it uses some CSS to fill in the stars by setting the width to a certain percent, which is provided in the aptly-named `percent` variable.

3. Now to make it work! Currently in the `app/views/reviews/index.html.erb` file, we're rendering the *number* of stars for each review. Change that file to instead render the `shared/stars` partial, making sure to assign the required `percent` variable to the result of calling the `stars_as_percent` method for each review.

   Answer:

```
<ul class="reviews">
  <% @reviews.each do |review| %>
    <li>
      <%= render "shared/stars", percent: review.stars_as_percent %>
      <p>
        <%= review.name %>
        <span class="date">
          <%= time_ago_in_words(review.created_at) %> ago
        </span>
      </p>
      <p>
        <%= review.comment %>
      </p>
    </li>
  <% end %>
</ul>
```

Cool—*partially* filled in stars!

Wrap Up

It usually takes a few iterations of working with nested resources before you're comfortable with them. If you find yourself getting confused, step back and look at your routes. They tell you what the URLs looks like, where they go, what parameters will be filled in for you, and the names of the route helper methods. Seriously, we look at our routes all the time to stay on track!

In the next section, we'll get more practice with nested routes by creating a web interface for writing reviews. But first, all this talk about nesting has us thinking it might be time to fly the roost, take a walk, and get some sun on our faces. So feel free to get up and stretch your wings. You've earned it!

One-to-Many: Forms

Exercises

Objective

Now that we've gotten our feet wet with nested resources, we're ready to wade in a bit deeper. After all, any online movie app worth its salted popcorn would let users write reviews in the browser! To do that we'll need to implement a few more of the nested review routes.

Here's the high-level flow we're aiming for:

- A user goes to a movie detail page and clicks a "Write Review" link.
- That takes them to a new page with a form where they enter their review and submit it.
- If the review is valid, the user ends up on the page that shows the movie's reviews with a cheery "Thanks for your review!" flash message. Otherwise, if the review is invalid, the user ends up seeing the review form with error messages prompting them to try again.

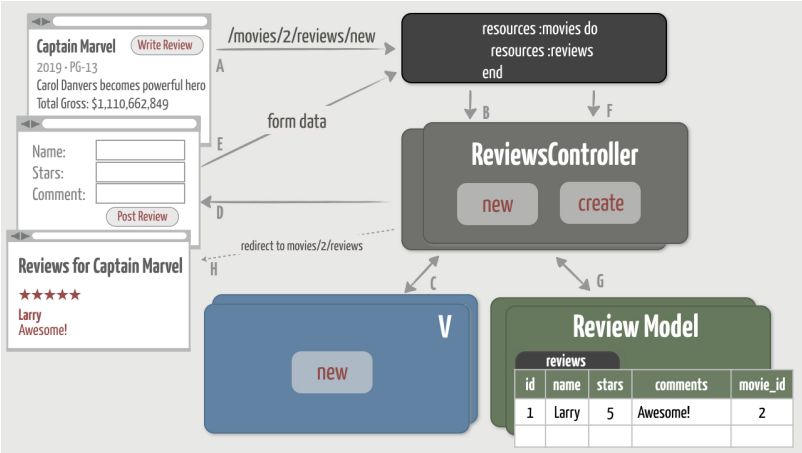
Sound familiar? It should because we did something very similiar to allow users to create movies. Creating reviews follows the same flow. In this situation, however, we need to make sure that reviews are always created in the context of a movie. For example, accessing the URL `/movies/2/reviews/new` should show a form for writing a review for the movie with an ID of 2. And when the form is submitted, the review should get associated with that movie in the database. So both the `new` and `create` actions need to know which movie is being reviewed.

Here's the good news: Because we already configured nested review routes in the previous exercise, every path to the `ReviewsController` already requires a movie ID in the URL. That ID is automatically captured in the `:movie_id` parameter. And that means we can rely on the `:movie_id` parameter being filled in whenever *any* action in the `ReviewsController` runs.

Again, there's a lot of commonality between creating reviews and creating movies, so we'll follow similar steps:

1. Generate a "Write Review" link on the movie `show` page so it's easy to get to the review form for any movie.
2. Define a `new` action in the `ReviewsController` that finds the movie being reviewed and uses it to instantiate a new `Review` object to bind to the form.
3. Create a corresponding `new.html.erb` template that generates an HTML form with fields for the review.
4. Define a `create` action in the `ReviewsController` that uses the form data to create a new review in the database and associates it with the respective movie.

Visually, here's what we want to do:



Nested resources are fairly common in Rails apps, so this will give us a good opportunity to get more practice implementing them. It's also a good "put it all together" exercise that draws from many of the techniques we've already learned. As such, this exercise is fairly involved. If you get stuck along the way and need to backtrack (or rewatch the video), don't beat yourself up. This is completely normal. It may take you a time or two through the material before nested resources really sink in. Be kind to yourself—you're learning!

1. Generate a "Write Review" Link

Reviews are the bread and butter of any online movie review app, so we need a "Write Review" link on the movie `show` page. Let's start there. And perhaps give it a try on your own first before following the steps below. :-)

1. Can you guess which route helper method we'll use to generate the "Write Review" link? Here's a hint: clicking that link should run the `new` action in the `ReviewsController`. If your first intuition was to look at the declared routes, pat yourself on the back!

   Answer:

   ```
   new_movie_review_path
   ```

2. Now use that route helper method to generate a "Write Review" link on the movie `show` page. To take advantage of some pre-defined styles, put it under the existing `table` tag and set the CSS `class` to `review`.

   Answer:

   ```
   <div class="details">
     <h1><%= @movie.title %></h1>
     <h2>
       <%= year_of(@movie) %> &bull; <%= @movie.rating %>
     </h2>
     <div class="reviews">
       <%= link_to pluralize(@movie.reviews.size, "review"),
                   movie_reviews_path(@movie) %>
     </div>
     <p>
       <%= @movie.description %>
     </p>
     <table>
       ...
     </table>
     <%= link_to "Write Review", new_movie_review_path(@movie),
                 class: "review" %>
   </div>
   ```

3. Then, back in your browser, navigate to a movie show page and you should see the freshly-minted "Write Review" link styled as a button. If you get an error, double-check the name of the route helper method.

4. Finally, go ahead and click "Write Review". You should get an obvious error:

   ```
   Unknown action

   The action 'new' could not be found for ReviewsController
   ```

Think about how you might implement the `new` action before moving on...

2. Create the New Action and Template

Think you know what comes next? The `new` action needs to display a review form, of course. (It's not quite as fun as throwing rotten tomatoes, but it'll have to do!) And you already know how to write a `new` action to display a form.

But with nested resources, there's a slight twist to the plot. The form needs to be generated in such a way that submitting it sends the data to the `create` action. And according to the nested routes, the way to get there is to send a `POST` request to `/movies/2/reviews`, for example. In order to create that URL, the form needs a `Movie` object so that it can fill in the `:movie_id` parameter in the URL. The form *also* needs a `Review` object in order to display the form with the review's fields.

So, unlike the form for creating a movie, the review form needs *two* objects. Which means the `new` action needs to set two instance variables. First, the `new` action needs to find the movie specified in the `/movies/2/reviews/new` URL, for example. Then the action needs to instantiate a `Review` object. Finally, the action needs to render a template that generates the review form.

1. Keying off the last error, define a `new` action in the `ReviewsController`. Begin the action by finding the movie and assigning it to a `@movie` variable. Then use that variable to instantiate a new `Review` object as a child of the movie and assign it to a `@review` variable.

   Hint:

   Remember, all the actions in the `ReviewsController` we can rely on the `:movie_id` parameter being filled in. Use that parameter to find the corresponding movie in the database. By looking up the movie in the database, rather than just using the passed in ID, we also verify that the ID corresponds to a valid movie before displaying the form. Then, to initialize a new `Review` object, use `@movie.reviews.new`. Using `Review.new` would technically work, but since a review should always be linked to its associated movie, using the association to initialize the review is preferred. The review's `movie_id` will be automatically assigned when initialized this way.

   Answer:

   ```
   def new
     @movie = Movie.find(params[:movie_id])
     @review = @movie.reviews.new
   end
   ```

2. Refresh the page (you're still accessing http://localhost:3000/movies/2/reviews/new) and you should totally expect this error:

   ```
   No template for interactive request

   ReviewsController#new is missing a template...
   ```

3. Following the error, create a file named `new.html.erb` in the `app/views/reviews` directory.

   In the `new.html.erb` template, for now just use the `@movie` variable to generate header text such as "New Review for Iron Man". Again, if you want to get sporty, link the movie's title back to the movie detail page.

   Answer:

   ```
   <h1>New Review for <%= link_to @movie.title, @movie %></h1>
   ```

   Refresh the page as a quick smoke test. You know everything's wired up correctly if the header text reflects the movie specified in the URL.

4. Carry on by generating a basic form. To keep the form simple for now, just generate a `text_field` for the reviewer's `name` attribute and a "Post Review" submit button. Generating a nested resource form is similar to the other forms we've generated, with one exception: the `form_with` method needs to be passed both the `@movie` and the `@review` as an array.

   Hint:

   To generate the form, use `form_with(model: [@movie, @review])`. The `form_with` helper uses the `@movie` object to generate the POST URL. (Yup, it actually uses the `movie_reviews_path` route helper method, passing in the movie object to fill in the `:movie_id` parameter, to generate that URL.) The `@review` object is used to generate a form "bound" to the review's attributes.

   Answer:

   ```
   <%= form_with(model: [@movie, @review]) do |f| %>
     <%= f.label :name %>
     <%= f.text_field :name, autofocus: true %>

     <%= f.submit "Post Review" %>
   <% end %>
   ```

5. Refresh and you should see an empty text field for the reviewer's name and a "Post Review" button. That's worth another victory dance! If instead you see an error, take a deep breath and calmly read it (out loud). Have both instance variables been assigned? Did you double-check the `form_with` syntax? Is it time for a mental (snack) break?

6. At this point it's worth viewing the page source and narrowing in on this part of the HTML form:

   ```
   <form action="/movies/2/reviews" method="post">
   ```

   The `action` attribute says that submitting this form will send the data to `/movies/2/reviews`. The `method` attribute says that the HTTP verb will be a `POST`. And according to our routes, that verb-URL combination matches the route that goes to the `create` action. So we're golden!

7. Now that the form is being displayed, finish it off by incrementally adding the following form elements:

   - a drop-down to select the number of stars (using `Review::STARS`)
   - a text area to enter the review comment (placeholder text would be helpful)

   Need a little help with the syntax and/or styling? There's absolutely no shame in cheating by looking at the `app/views/movies/_form.html.erb` partial for pointers. Give it an honest try on your own first before looking at the answer.

   Answer:

   ```
   <%= form_with(model: [@movie, @review]) do |f| %>
     <%= f.label :name %>
     <%= f.text_field :name, autofocus: true %>

     <%= f.label :stars %>
     <%= f.select :stars, Review::STARS, prompt: "Pick one" %>

     <%= f.label :comment %>
     <%= f.text_area :comment, placeholder: "What did you think?" %>

     <%= f.submit "Post Review" %>
   <% end %>
   ```

Nicely done! We've displayed a form for writing movie reviews. Next we'll turn our attention to what happens when an intrepid user posts a review...

3. Implement the Create Action

We're on a roll, so let's just see what happens when we submit the form...

1. Click the "Post Review" button and... nothing happens. But if you check the server log, you should see the following run-of-the-mill error:

   ```
   AbstractController::ActionNotFound (The action 'create' could not be found for ReviewsController)
   ```

   Remember how to implement a `create` action? As a refresher, go ahead and look back at the `create` action in the `MoviesController`. The `create` action for reviews needs to handle the form data in similar fashion to the `create` action for movies. Both actions need to instantiate a new model object with the form's contents and save the object to the database if it's valid. But again, there's a twist when it comes to reviews. A review *must* be created so that it's associated with the movie being reviewed.

2. To implement the `create` action in the `ReviewsController`, you'll need to do three things. It actually doesn't require much code, and all the code should be familiar since we've already used it in different scenarios:

   - Before you can create a new review, you need to find the movie being reviewed (the form posts it in the `:movie_id` parameter). Assign that movie to a `@movie` variable.
   - Then initialize a new `Review` object with the contents of the form and (here comes the important part!) remember to use the `reviews.new` method to associate it with the movie. Assign the new review to a `@review` variable.

     Don't forget that you'll need to pass `reviews.new` a list of attributes that are permitted to be mass-assigned from form data. For a refresher on how to do that, cheat by looking at the `MoviesController`.
   - If saving the review is successful, redirect to the page that shows the movie's reviews with a cheery flash notice such as "Thanks for your review!" Otherwise, if the review is invalid, redisplay the `new` form populated with any valid review data.

   Hint:

   The key to pulling this off is to use the `reviews` association to initialize a new `Review` as a child of the existing `Movie`. Using `@movie.reviews.new` to initialize the review ensures that the review's `movie_id` foreign key is automatically set to point to the movie. After you've initialized the review, the rest of the action should be very familiar.

   Answer:

   ```
   def create
     @movie = Movie.find(params[:movie_id])
     @review = @movie.reviews.new(review_params)

     if @review.save
       redirect_to movie_reviews_path(@movie),
                   notice: "Thanks for your review!"
     else
       render :new, status: :unprocessable_entity
     end
   end

   private

   def review_params
     params.require(:review).permit(:name, :comment, :stars)
   end
   ```

3. Back in your browser, give it a whirl by creating a new (valid) review for a movie. You should end up getting redirected to the listing of reviews for that movie, with a flash message thanking you for the review. And you should see the review you just posted!

   If for some reason it doesn't work as you expect, check the log in the command prompt window where your app is running. Make sure the parameters are being passed as you'd expect when the form is posted.

Fantastic! Now we have a web interface for creating reviews and associating them with their respective movie.

4. Display Validation Errors

Whoopsie! We followed the happy path, but forgot about the unhappy path.

1. Create an invalid review by entering a name, but leave the rest of the form blank. Submit it, and you should see the form redisplayed. The valid name you entered should be populated in the name field. The other required fields should be suspiciously highlighted, but the validation error messages aren't being displayed at the top of the form.

2. As you'll recall, we already have a partial that displays validation errors. It's in the `app/views/shared/_errors.html.erb` file.

   Hey, we can just use that partial to display review errors!

3. To do that, change the `new.html.erb` template in the `app/views/reviews` directory to render the `_errors` partial at the top of the form. Since the `_errors` partial assumes that a local variable named `object` references an ActiveRecord model object, you'll need assign `@review` to that variable when rendering the partial.

   Hint:

   Remember, when calling a partial, we assign local variables using a hash. Each hash key will be the name of a local variable and the value will be that variable's value in the partial. So in this case the key needs to be `object` and the value needs to be the `@review` object.

   Answer:

   ```
   <%= form_with(model: [@movie, @review]) do |f| %>
     <%= render "shared/errors", object: @review %>
   ```

```
    . . .
<% end %>
```

4. Now try creating an invalid review and you should see helpful error messages at the top of the form.

And that takes care of the unhappy path. :-)

5. Refactor

We're almost done! But before we cross this feature off the to-do list, let's take a moment to clean up some code duplication. Look back in your `ReviewsController`. See any duplicated code? Since a review only makes sense in the context of an movie, the first thing every action does is look up the movie using identical code:

```
@movie = Movie.find(params[:movie_id])
```

It's easy to imagine if we add new actions we'll end up duplicating this line of code more times. So let's clean up the code duplication before it becomes more of a problem.

1. Start by writing a `private` method named `set_movie` that retrieves the movie and sets the `@movie` instance variable.

   Answer:

   ```
   def set_movie
     @movie = Movie.find(params[:movie_id])
   end
   ```

   Note that methods that are `public` (the default access control) are callable as actions. Our `set_movie` method is only going to be called within the controller, so it's good form to make the method `private`. That way it can't be called as an action.

2. We always want to run the `set_movie` method *before* running any action's code. To do that, use `before_action` to make sure the method is called before any action's code is executed.

   Hint:

   Call the `before_action` method and pass it the *name* of the method you want to run (as a symbol).

   Answer:

   ```
   class ReviewsController < ApplicationController
     before_action :set_movie

     # existing code
   ```

3. Then remove the duplicated line of code from each action.

4. Check your handiwork in the browser and everything should flow just as before. But don't you feel better knowing your code is nice and tidy?

Using a `before_action` is an idiomatic way to remove duplication in a controller, but you can also use `before_action` to share common functionality across controllers. For example, if you had a `before_action` in the `ApplicationController` class it would get applied to *every* action in your application. Alternatively, you could define a common method in the `ApplicationController` and then selectively apply it to specific controllers by adding the `before_action` line to each participating controller. Finally, you can use the `:only` and `:except` options to apply the `before_action` to specific actions within a controller.

Solution

The full solution for this exercise is in the `one-to-many-forms` directory of the code bundle.

Bonus Round

What, that wasn't enough for ya? OK, eager beaver. Here are a few ideas for additional features you might want to try if you're up for a challenge. You do **not** need to finish these before moving on to the next exercise. Consider them self-study extra credit!

1. The review form is currently using a drop-down to select the number of stars to award a movie. Given that there are only five options and we don't anticipate adding more, radio buttons are probably more user friendly. Change the review form to use radio buttons for selecting the stars.

   Answer:

   ```
   <%= f.label :stars %>
   <% Review::STARS.each do |star| %>
     <span class="star">
       <%= f.radio_button :stars, star %>
       <%= label_tag star %>
     </span>
   <% end %>
   ```

2. Suppose you wanted a user to be able to write a review directly on a movie's show page. Try putting the review form at the bottom of that page. You'll need to make sure the `show` action initializes a review object for the form to use. It might also be a good idea to make a form partial so you can use it on the movie's `show` page and the review's `new` page.

3. Add functionality to delete reviews. It ends up being very similiar to the flow for deleting movies, but you'll need to use a nested route helper method to generate the "Delete" link. Then in the `destroy` action, you'll need to use the `reviews` association to find the existing review that's associated with the movie specified in the URL.

4. To take things up a notch, add functionality to edit existing reviews. Again, it's similar to editing movies, but involves using the nested routes. To avoid duplication, use a form partial to reuse the existing form for creating reviews.

Wrap Up

Way to hang in there! We accomplished a lot over the last couple exercises. Even though the concept of *one* movie having *many* reviews isn't necessarily complicated, there can be quite a bit of mental overhead involved when it comes to actually implementing this association in the user interface. You always have to keep not one, but *two* objects in your head as you work through each component of the MVC triad.

In the next section, we're going to have some fun with the review data we now have nestled in our database.

One-to-Many: Logic

Exercises

Objective

Now that we have a web interface for writing movie reviews, let's have a little fun with the data we're collecting. For example, movie buffs would like to see the average number of stars for each movie. It turns out that Active Record makes it easy to run these sorts of calculations without having to write a bunch of complicated SQL. So let's get right to it!

1. Calculate a Movie's Average Stars

The inefficient way to calculate the average number of stars for a movie would be to use the `movie.reviews` association to load the array of reviews, loop through each review accumulating star values, then calculate the average at the end. If the data set is small enough, loading all the reviews isn't a big deal.

But it's more efficient to use native in-database calculation functions. That way we're not loading a bunch of reviews into memory and then throwing them away. Each database supports calculations slightly differently, but Active Record hides all those details behind a friendly set of calculation methods.

So, before launching straight into the task at hand, let's spend a few seconds in the `console` just getting the hang of the built-in `average` calculation method.

1. To calculate the average value of a given model attribute, you use the `average` method. For example, here's how to calculate the average number of stars for all the reviews in the database:

   ```
   >> Review.average(:stars)
   ```

   The result is a `BigDecimal`, so to print it in a friendly format in the console you'll need to use `to_s`:

   ```
   >> Review.average(:stars).to_s
   ```

   Notice that the generated SQL statement uses SQLite's `AVG` function to perform the entire calculation in one database query. Pretty cool!

2. Ok, so we called the `average` method on the `Review` class. And that ran the average calculation on all the rows in the `reviews` database table. That's interesting, but it's not really what we want. We want to calculate the average number of stars for reviews written for a *particular* movie.

   How do we do that? Well, here's something really cool: we can also call the `average` method on an *association*. In particular, we can call the `average` method on the `reviews` association of a specific movie.

   For example, here's how to calculate the average number of stars for "Captain Marvel":

   ```
   >> movie = Movie.find_by(title: "Captain Marvel")

   >> movie.reviews.average(:stars).to_s
   ```

   Notice that the SQL statement is now querying the rows in the `reviews` database table, and in particular only those rows that belong to the movie "Captain Marvel". And again, it's just *one* database query!

Nifty! Now let's apply what we learned back in the app...

2. Display a Movie's Average Stars

Returning to our original task, we want to display the average number of review stars for each movie. We know how to run the calculation, but it's always important to think about where the code will live. Calculations of this kind are a type of business logic. As such, they belong in the model. We'll start by encapsulating the calculation in an `average_stars` method of the `Movie` model, then work our way back out to the web interface.

1. In the `Movie` class, define an instance method named `average_stars`. Implement the method so that it calculates and returns the average number of stars for the movie. There's a corner case to be aware of. If the movie doesn't have any reviews then the calculation will return `nil`. In that case, return a value of 0.0 as the average.

   Hint:

   You'll call the same calculation method you used earlier in the `console`, but because you're inside of an instance method you don't need to use the `movie` object. The current object (`self`) will already be set to a movie object and it becomes the implicit receiver of the call to `reviews`. For bonus style points, use the Ruby `||` operator to return 0.0 if the calculation returns nil.

   Answer:

   ```
   class Movie < ApplicationRecord
     # existing code

     def average_stars
       reviews.average(:stars) || 0.0
     end
   end
   ```

2. Next we want to display the average number of stars on a movie's `show` page. That's easy enough, but there's a catch: if the movie doesn't have any reviews then the `average_stars` method will return 0.0. In that case, rather than just displaying a zero value we instead want to display "No reviews".

Hmm... so to do that we'll need some conditional view logic. And that always belongs in a view helper. Let's start from the outside-in and assume we have a `average_stars` helper that takes a `movie` object and applies the conditional logic. Even though we don't have that helper method yet, go ahead and update the movie `show` template with the following:

```
<div class="reviews">
  <%= average_stars(@movie) %>
  <%= link_to pluralize(@movie.reviews.size, "review"),
                movie_reviews_path(@movie) %>
</div>
```

3. Now implement that view helper method. If the given movie has an average stars value of zero, the helper should return the string "No reviews" (in bold). Otherwise, return the average number of stars as the string "2 stars", for example.

   Hint:

   Define the `average_stars` helper method in the `movies_helper.rb` file. It needs to take a `movie` object as a parameter. The helper will turn around and call the `movie.average_stars` method (defined in the model) to get the average number of stars for that movie. Then it needs to check if the returned value is zero.

   Answer:

   ```
   def average_stars(movie)
     if movie.average_stars.zero?
       content_tag(:strong, "No reviews")
     else
       pluralize(movie.average_stars, "star")
     end
   end
   ```

4. Here's another finishing move: Since the `movie.average_stars` method returns a `BigDecimal`, you can end up with fairly precise numbers such as 3.333333333333333. That's great for science, but overkill for movie review averages. To convert it into something more manageable such as 3.3, use the built-in number_with_precision helper to format the result with a precision of 1.

   Answer:

   ```
   def average_stars(movie)
     if movie.average_stars.zero?
       content_tag(:strong, "No reviews")
     else
       pluralize(number_with_precision(movie.average_stars, precision: 1) , "star")
     end
   end
   ```

5. Finally, back in your browser, go to a movie `show` page and you should see the average number of stars!

6. Since you now have a helper that makes it easy to display a movie's average stars, feel free to update the movie `index` page as well if you fancy seeing averages in the movie listing. To pick up some CSS styling, put the call to the `average_stars` helper in a `span` with a class of `stars`. This will display the average stars to the right of the movie title.

   Answer:

   ```
   <div class="summary">
     <h2>
       <%= link_to movie.title, movie_path(movie) %>
     </h2>
     <h3>
       <%= total_gross(movie) %>
     </h3>
     <span class="stars">
       <%= average_stars(movie) %>
     </span>
     <p>
       <%= truncate(movie.description, length: 150, separator: ' ') %>
     </p>
   </div>
   ```

Nicely done—5 stars!

Solution

The full solution for this exercise is in the `one-to-many-logic` directory of the code bundle.

Bonus Round

Asterisk Stars

Change the `average_stars` helper method to display the average number of stars as a number of asterisks (*). For example, if the average rating is 4, show four asterisks.

Hint:

Here's a Ruby trick you may not know: You can multiply a string (e.g. "*") by a number `x`. The result is a new string with the original string repeated `x` times. Remember though that the `average_stars` method returns a `BigDecimal` such as 3.66666. Multiply "*" by 3.66666 and you'll get three asterisks. If you want to round up (so you get four asterisks), you can use the `round` method.

Answer:

```
def average_stars(movie)
  if movie.average_stars.zero?
    content_tag(:strong, "No reviews")
  else
```

```
    "*" * movie.average_stars.round
  end
end
```

★ Stars

If you completed a previous bonus exercise where you displayed a review's star rating (1 - 5) as a series of partially filled in stars, then it makes sense to also display a movie's average stars the same way!

1. First you need a way to convert a movie's average number of stars to a percentage. That responsibility clearly falls to the `Movie` model, and it already has an `average_stars` method.

   So start in the `Movie` model by defining an `average_stars_as_percent` instance method that defers to the existing `average_stars` method to calculate the average stars and returns the average stars as a percentage.

   Answer:

   ```
   class Movie < ApplicationRecord
     # existing code

     def average_stars_as_percent
       (self.average_stars / 5.0) * 100
     end
   end
   ```

2. Then on the movie `index` page, change the line that calls the `average_stars` helper method to instead render the existing `shared/stars` partial, making sure to assign the required `percent` variable to the result of calling the `movie.average_stars_as_percent`.

   Answer:

   ```
   <span class="stars">
     <%= render "shared/stars", percent: movie.average_stars_as_percent %>
   </span>
   ```

3. In the same way, on the movie `show` page, change the line that calls the `average_stars` helper method to instead render the same `shared/stars` partial and assign the percent.

   Answer:

   ```
   <div class="reviews">
     <%= render "shared/stars", percent: @movie.average_stars_as_percent %>
     <%= link_to pluralize(@movie.reviews.size, "review"),
                 movie_reviews_path(@movie) %>
   </div>
   ```

4. If you want, you can now remove the `average_stars` helper method defined in the `MoviesHelper` module since that method is no longer being used.

Now both the movie listing page and the movie detail page should show a series of partially-filled in stars reflecting the average reviews!

More Calculation Methods

In addition to `average`, the calculation API includes other handy calculation methods. Look through the possibilities, and then give the following scenarios a try in the `console`:

1. Start by calculating the minimum total gross value of all the movies in the database.

   Answer:

   ```
   >> Movie.minimum(:total_gross).to_s
   ```

2. Now calculate the **maximum** total gross value of all the movies.

   Answer:

   ```
   >> Movie.maximum(:total_gross).to_s
   ```

3. What if you wanted to calculate the minimum (or maximum) total gross of only the movies that have a PG-13 rating? It turns out you can chain together calls such as `where` and `minimum`, for example. Give it a try before looking at the answer.

   Answer:

   ```
   >> Movie.where(rating: "PG-13").minimum(:total_gross).to_s
   ```

4. The accountants think it would be interesting to tally up the total gross values for all the movies in our database. Use a calculation to do that.

   Answer:

   ```
   >> Movie.sum(:total_gross).to_s
   ```

5. The statisticians think it's more interesting to calculate the **average** total gross of all our movies. Show 'em how it's done.

   Answer:

   ```
   >> Movie.average(:total_gross).to_s
   ```

6. The marketing gurus think it would be interesting to tally up the total number of review stars earned by a movie. We have no idea what they'll do with this data, but the future of movie advertising may be at your fingertips!

   Answer:

```
>> movie = Movie.find_by(title: "Captain Marvel")
>> movie.reviews.sum(:stars)
```

7. Now calculate the **maximum** number of stars in a review for the same movie.

   Answer:

   ```
   >> movie.reviews.maximum(:stars)
   ```

Calculations are fun stuff, even if you aren't an accountant, statistician, or marketing guru!

Cult Movies

We all know of a movie that didn't necessarily gross a ton at the box office, but the few people who did watch the movie *really* loved it! We call those movies *cult classics*. And to a specific group of fans they are blockbusters.

Change the definition of the `flop?` method so that cult classics aren't included. For example, if a movie has more than 50 reviews and the average review is 4 stars or better, then the movie shouldn't be a flop regardless of the total gross.

Here's a hint: Because the logic for determining whether a movie is a flop is tucked inside the `Movie` model, you can make this change in one place. When you can do that, you know you're on the right design path!

Answer:

```
def flop?
  unless (reviews.count > 50 && average_stars >= 4)
    (total_gross.blank? || total_gross < 225_000_000)
  end
end
```

Wrap Up

Once you start creating model associations, looking at the data, and talking with users, you'll likely come up with all sorts of interesting business logic. The secret is to always try to push as much business logic as you can back into the model. That makes it easier to use the logic in different parts of your application.

Also, although it may be tempting to perform calculations in Ruby, you'll get better performance if you instead use Active Record's calculation methods.

Dive Deeper

To learn more about Active Record calculations, refer to the "Calculations" section in the Rails Guide: Active Record Query Interface.

User Account Model

Exercises

Objective

Over the next few exercises we'll incrementally create a basic user account and authentication system from scratch. For the purposes of this exercise, we'll focus on getting the `User` model in good shape. Doing that involves the following tasks:

- Generate a `User` resource that securely stores user passwords in a `password_digest` column
- Install the `bcrypt` gem
- Declare reasonable validations in the `User` model
- Create a couple example users in the database using the Rails console

We hope we don't need to belabor the point that **passwords should never, ever be stored in the database as plain text**. But just for good measure, go ahead and raise your right hand now and repeat aloud after us:

> "I, (state your name), being a professional in the craft of software development, do hereby promise to take responsibility for securing the passwords entrusted to me by users of my application. I will never, under any circumstance including project deadline pressures, store the plain-text version of a user's password in my database or other storage medium. Furthermore, I will not hold Mike and Nicole liable for any wrongdoing on my part. And I won't use totally obvious passwords such as '123456' when creating my own account."

OK, then. We're ready to make good on that promise!

1. Generate the User Resource

First we need to create a new (third) resource to represent a user with a name, e-mail address, and secure password. We'll need a `users` database table, a `User` model, and ultimately a `UsersController` and conventional routes for interacting with users via the web. We'll use the *resource generator* to make quick work of this.

1. Start by generating a resource named `user` (singular) with `name` and `email` attributes of type `string`, and a `password` attribute of type `digest`.

   Hint:

   For a refresher on the resource generator options, run the `rails g resource` command.

   Answer:

   ```
   rails g resource user name:string email:string password:digest
   ```

2. We'll look at everything that got generated as we go along. For now, open the generated migration file and you should see the following:

```
class CreateUsers < ActiveRecord::Migration[7.0]
  def change
    create_table :users do |t|
      t.string :name
      t.string :email
      t.string :password_digest

      t.timestamps
    end
  end
end
```

Notice that the generator translated the declaration `password:digest` into a `password_digest` column of type `string` in the migration file. That's pretty smart!

3. To actually create the `users` database table, you need to run the migration.

   Answer:

   ```
   rails db:migrate
   ```

4. The generator also gave us a `User` model in the `app/models/user.rb` file, so crack that file open and you should see the following:

   ```
   class User < ApplicationRecord
     has_secure_password
   end
   ```

   Again, the generator is pretty clever! Notice that it added the `has_secure_password` line. It knew we wanted to store passwords securely because we used `password:digest` when running the generator.

   As you'll recall from the video, the built-in `has_secure_password` method in Rails helps us do the right thing when it comes to securely storing passwords. And when you read it out loud it makes sense: A user has a secure password.

   In the `Movie` and `Review` models, we used similar declarations—`belongs_to` and `has_many`—to get functionality for managing model relationships. Using `has_secure_password` in a model adds functionality for securely storing passwords.

5. The `has_secure_password` method depends on the bcrypt gem which implements a state-of-the-art secure hash algorithm used to encrypt passwords. Not all Rails applications need this gem, so it's commented out in the default `Gemfile`.

   So you need to uncomment the following line in your `Gemfile`:

   ```
   gem 'bcrypt', '~> 3.1.7'
   ```

6. Then install the gem using

   ```
   bundle install
   ```

And that's all there is to it! With these changes in place, Rails has everything it needs to securely store passwords in the `password_digest` column of the `users` database table.

2. Declare User Validations

Next we need to add some reasonable validations to the generated `User` model to ensure that invalid user records can't be stored in the database. The `has_secure_password` line automatically adds password-related validations, but we also need validations for the user's name and email. Use built-in validations to enforce the following validation rules:

1. A name must be present.

   Answer:

   ```
   validates :name, presence: true
   ```

2. An email must be present and formatted so that it has one or more non-whitespace characters on the left and right side of an `@` sign.

   Hint:

   Remember that the `format` validation uses a Ruby regular expression to match a field's value. The regular expression for emails can be a bit tricky, so go ahead and use `/\S+@\S+/`.

   Answer:

   ```
   validates :email, format: { with: /\S+@\S+/ }
   ```

3. We don't want two users in the database to have the same e-mail address. So make sure emails are unique regardless of whether they use upper or lower case characters.

   Hint:

   The `uniqueness` validation performs a simple check to ensure that no two rows in database table have the same field value. You can also pass the `:case_sensitive` option to the `uniqueness` validation. By default, `:case_sensitive` is set to `true` which means it will check for an exact match. However, email addresses are case-insensitive, so you'll need to explicitly set `case_sensitive` to `false`.

   Answer:

   ```
   validates :email, presence: true,
                     format: { with: /\S+@\S+/ },
                     uniqueness: { case_sensitive: false }
   ```

3. Create Users in the Console

Now that we have a `users` database table and a `User` model with validations, let's try creating some users in the database using the Rails console and see what `has_secure_password` gives us:

1. Fire up a Rails `console` session.

   ```
   rails c
   ```

2. Then instantiate a new `User` object without a name, email, or password.

   Answer:

   ```
   >> user = User.new
   ```

3. Now try to save the invalid `User` object to the database.

   Answer:

   ```
   >> user.save
   ```

   The result should be `false`. Remember, when you try to save (or create) a model object, its validations are automatically run. If a validation fails, a corresponding message is added to the model's `errors` collection. And if the `errors` collection contains any messages, then the save is abandoned and `false` is returned. In short, the failed validations prevent the user from being saved to the database, which is exactly what we want!

4. To see which validations are failing, inspect the validation error messages by accessing the `errors` collection. To dig down into the actual error messages, tack on a call to `full_messages` to get an array of error messages.

   Answer:

   ```
   >> user.errors.full_messages
   ```

   You should get the following:

   ```
   => ["Password can't be blank",
    "Name can't be blank",
    "Email can't be blank",
    "Email is invalid"]
   ```

   Notice that `has_secure_password` added a validation to ensure a password is present when creating a new user, in addition to the name and email validations we declared. Nice!

5. Go ahead and assign just a name and email for the user.

   Answer:

   ```
   >> user.name = "Larry"
   >> user.email = "larry@example.com"
   ```

   These attributes map directly to the `name` and `email` database columns.

6. Next, set a password for the user by assigning a value to the virtual `password` attribute.

   Answer:

   ```
   >> user.password = "abracadabra"
   ```

   Remember, the `password` attribute is a *virtual attribute* that was dynamically defined by the `has_secure_password` method. Unlike a typical attribute, when we assign a value to the `password` attribute it doesn't try to store the value in a corresponding `password` database column. That's good because we don't have a `password` column and we don't want plain-text passwords stored in our database!

   Instead, assigning a value to the `password` attribute causes the plain-text version of the password to be encrypted and the encrypted version is then stored in the `password_digest` column. It appears to be a clever sleight of hand, but now you know it's not actually magic. It works because assigning a value to the `password` attribute calls the special `password=` method defined by `has_secure_password`. And that method turns around and does the encryption for us.

7. Now for the big reveal: Print the value of the `password_digest` attribute.

   ```
   >> user.password_digest
   ```

   You should get a string of what looks like gibberish, such as:

   ```
   "$2a$10$wTBwLqnYrXffr.ainX60qOVB6hWeF4T1rU3RMHTL2olZ.erAmJS7O"
   ```

   That string, typically referred to as an irreversible *digest*, is the result of running the plain-text password through the one-way hash algorithm in the `bcrypt` gem.

8. Now set a password confirmation that doesn't match the password and try to save the user record again.

   Answer:

   ```
   >> user.password_confirmation = "alakazam"

   >> user.save
   => false
   ```

   Think of a typical sign-up form that prompts for a password and makes you re-enter it to confirm that the passwords match. That's pretty common, and `has_secure_password` has you covered. It added a `password_confirmation` attribute and a validation that requires a password confirmation to be present, as well.

9. Check the validation error messages and you should get the following:

   ```
   >> user.errors.full_messages
   => ["Password confirmation doesn't match Password"]
   ```

Note that the `password_confirmation` attribute is also a *virtual attribute*. It doesn't map to a database column. Instead, when you assign a value to `password_confirmation`, the value is simply stored temporarily in an instance variable. Behind the scenes, `has_secure_password` runs the validations against that instance variable.

10. Then assign a matching password confirmation so that you can finally (successfully) save the user to the database.

    Answer:

    ```
    >> user.password_confirmation = "abracadabra"

    >> user.save
    => true
    ```

11. Next, create a second user by calling the `new` method and passing it a hash of attribute names and values: `name`, `email`, `password`, and `password_confirmation`. Save it and make sure it successfully saves without any validation errors.

    Answer:

    ```
    >> user = User.new(name: "Daisy", email: "daisy@example.com", password: "open-sesame", password_confirmation: "open-sesame")
    >> user.save
    => true
    ```

12. To demonstrate that the plain-text password is only temporarily stored in memory, find the user you just created by their email address like so:

    ```
    >> user = User.find_by(email: "daisy@example.com")
    ```

    Then print the value of the `password_digest` attribute and you should get the encrypted-version of the password (gibberish):

    ```
    >> user.password_digest
    => "$2a$12$k9ohzPl5snCOuKvzgqXyMuwxj7YkQtbhopihmQy1OyrcgiGgVFqo."
    ```

    We assigned a plain-text password when we initially created the user, but is it still there? To check, print the value of the `password` attribute:

    ```
    >> user.password
    => nil
    ```

    Ah, it has a value of `nil`! Remember, we fetched the user from the database. Doing that populated all the real (non-virtual) attributes with the values corresponding to each column. But there is no `password` column, so the `password` virtual attribute has a value of `nil`. And that's exactly what we want: there's no trace of the original plain-text password!

13. Finally, since we required that a user's email be unique, try creating a third user with the same email address as a previously-created user.

    Answer:

    ```
    >> user = User.new(name: "Daisee", email: "daisy@example.com", password: "secret", password_confirmation: "secret")

    >> user.save
    => false
    ```

    It should return `false` and the database transaction should get rolled back. Notice that the `uniqueness` validation ran a SQL `SELECT` query to check if a user with the same email address already existed in the database.

    Print the validation error messages.

    Answer:

    ```
    >> user.errors.full_messages
    => ["Email has already been taken"]
    ```

    Indeed, a user already exists with the same email address so the new (duplicate) user wasn't created.

14. Remember how to check that you now have *two* unique users in the database?

    Answer:

    ```
    >> User.count
    ```

Hey, we got a lot for free! By following the tried-and-true convention, `has_secure_password` takes care of validating that a user has a (confirmed) password and securely stores it for us.

Solution

The full solution for this exercise is in the `user-account-model` directory of the code bundle.

Bonus Round

We've seen that `has_secure_password` automatically gives us validations for the presence and confirmation of a user password. That's a really good start, but of course you can add more validations as necessary.

For example, suppose you wanted to require passwords to be at least 10 characters in length. To do that, add the following to your `User` model:

```
validates :password, length: { minimum: 10, allow_blank: true }
```

By setting the `allow_blank` option to `true`, the length validation won't run if the password field is blank. That's important because a password isn't required when a user updates his name and/or email. So if the password field is left blank when editing the user account, the length validation is skipped.

What About a Gem?

You might be wondering why we just don't use an off-the-shelf authentication gem such as Devise. It's not because we have anything against gems. But gems tend to come and go. So rather than investing a lot of time in learning a particular gem, we believe that learning how things work at a fundamental level is a better long-term investment.

Every Rails developer should know at a basic level how an authentication system works. And the best way to gain that understanding is to build one yourself. That way, should you decide down the road to slip a third-party gem such as Devise in your app, you'll be in a much better position to understand, customize, and troubleshoot if you run into any problems.

Wrap Up

Excellent! Our `User` model looks to be in good shape, so we're ready for the next exercise where we'll tackling the web interface for user accounts. Abracadabra...

Dive Deeper

- If you're curious how `has_secure_password` works under the hood, and don't mind stretching your Ruby skills, spend a few minutes reviewing the has_secure_password source. It's well-documented and a lot less code than you might expect. And it's always good to at least have a high-level understanding of how something works, especially if you're entrusting it to do something as important as securing user passwords.
- You might also be curious as to how the bcrypt gem is implemented. It's actually just a Ruby binding for the OpenBSD bcrypt password hashing algorithm. It's important to note that it's a *one-way, irreversible* hashing algorithm that uses a salt to guarantee a unique output (the hashed password) even when the inputs are the same.
- If you want to be more strict about e-mail address validation, you might consider using the email_validator gem.

User Signup

Exercises

Objective

Now that the `User` model is in good shape, we need a web interface that supports the following account management functions:

- List all the user accounts
- Show a user's profile page that displays their account information
- Allow new users to create an account using a sign-up form

Creating the web interface for user accounts is very similar to creating the interface for movies and reviews. So most of this will be review, and good practice!

1. List All Users

To get things rolling, let's start by displaying a list of users currently in the database. (We'll link each user to their profile page which we'll create shortly.) In terms of controller actions and templates, what will you need?

Answer:

an `index` action and corresponding `app/views/users/index.html.erb` view template that displays a list of users

If you're feeling confident, by all means give this exercise a try on your own first before following the steps below.

1. To get your bearings, check out the defined routes and you'll notice we already have the following routes for interacting with users:

   ```
   Helper         HTTP Verb        Path                  Controller#Action
   users_path       GET        /users(.:format)           users#index
                    POST       /users(.:format)           users#create
   new_user_path    GET        /users/new(.:format)       users#new
   edit_user_path   GET        /users/:id/edit(.:format)  users#edit
   user_path        GET        /users/:id(.:format)       users#show
                    PUT        /users/:id(.:format)       users#update
                    PATCH      /users/:id(.:format)       users#update
                    DELETE     /users/:id(.:format)       users#destroy
   ```

   These conventional resource routes exist because the resource generator we ran earlier added the following line to the `config/routes.rb` file:

   ```
   resources :users
   ```

   So we know the URL to use to get a list of users...

2. Browse to http://localhost:3000/users to list the existing users and you'll get an all-too-familiar error:

   ```
   Unknown action

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

3. Following the error, open the empty `UsersController` that was also generated by the resource generator and define an `index` action that fetches all the users from the database.

   Answer:

   ```
   def index
     @users = User.all
   end
   ```

4. Then create a corresponding `app/views/users/index.html.erb` view template. In the template, generate a list of user names with each name linked to the user's `show` page. To style the list according to our CSS rules (totally optional), make sure that the `ul` tag has the CSS class `users`.

   For extra credit, also display how long ago each user account was created and the number of users being listed.

   Hint:

   You'll need to iterate through each (wink, wink) user and then use a `link_to` to hyperlink the user's name. Use the `time_ago_in_words` built-in helper to calculate how long ago the record was created. And use the built-in `pluralize` helper to generate an appropriately pluralized count of users.

   Answer:

   ```
   <h1><%= pluralize(@users.size, "User") %></h1>
   <ul class="users">
     <% @users.each do |user| %>
       <li>
         <%= link_to user.name, user %>
         created
         <%= time_ago_in_words(user.created_at) %> ago
       </li>
     <% end %>
   </ul>
   ```

5. Now reload the `index` page back in your browser and you should see the two users we created in the console in the previous exercise.

2. Show a User's Profile Page

When you click on a user's name in the user listing, we want to show their profile page. For now the profile page will simply display the user's name and email. We'll fill in more profile information in upcoming exercises.

In terms of controller actions and templates, what will you need?

Answer:

a `show` action and corresponding `app/views/users/show.html.erb` view template that displays the user's profile page

Give it a go on your own first!

1. Browse to http://localhost:3000/users and try clicking on one of the user names. It should come as no surprise that you get this error:

   ```
   Unknown action

   The action 'show' could not be found for UsersController
   ```

2. Following the error, define a `show` action that finds the requested user.

   Answer:

   ```
   def show
     @user = User.find(params[:id])
   end
   ```

3. Then create the corresponding `app/views/users/show.html.erb` template that displays the user's name and email. As a bonus, use the `mail_to` helper to link the user's email so that clicking it pops open your favorite email program with a new message addressed to the user.

   Answer:

   ```
   <section class="user">
     <h1><%= @user.name %></h1>
     <h2><%= mail_to(@user.email) %></h2>
   </section>
   ```

4. Now navigate from the user listing page to each user's profile page as a quick visual check. If you want to pick up our CSS styles, you'll need to use the same HTML markup as found in the solution above.

3. Create New User Accounts

Now that we have a user profile page to land on, we're ready to let users create new accounts using a sign-up form. To get to that form, they'll either click a "Sign Up" link in the header or browse to http://localhost:3000/signup. To do that, we'll need to:

- Add a route to support the custom URL http://localhost:3000/signup.
- Generate a convenient "Sign Up" link at the top of every page.
- Define a `new` action that renders the `new.html.erb` view template to display a sign-up form with fields for the user's name, email, password, and password confirmation.
- Define a `create` action that accepts the form data and uses it to create a new user in the database, but only if the user valid.

You've got this, so give it an honest try!

1. According to the existing routes, the URL for displaying the form to create a new user is http://localhost:3000/users/new. That request gets sent to the `new` action in the `UsersController`. We need to make that work, but we'd also like to support the more descriptive URL http://localhost:3000/signup. We actually want both URLs to show the sign-up form.

   In the `config/routes.rb` file, add a route that maps a `GET` request for `/signup` to the `new` action of the `UsersController`.

Answer:

```
get "signup" => "users#new"
```

2. Now check out the defined routes and you should see all the conventional resource routes for users **and** the custom `/signup` route (the last one below):

```
Helper          HTTP Verb        Path                  Controller#Action
users_path        GET        /users(.:format)           users#index
                  POST       /users(.:format)           users#create
new_user_path     GET        /users/new(.:format)       users#new
edit_user_path    GET        /users/:id/edit(.:format)  users#edit
user_path         GET        /users/:id(.:format)       users#show
                  PUT        /users/:id(.:format)       users#update
                  PATCH      /users/:id(.:format)       users#update
                  DELETE     /users/:id(.:format)       users#destroy
signup_path       GET        /signup(.:format)          users#new
```

Because we know you're already thinking ahead, identify the route helper method you'll need to generate a "Sign Up" link.

3. We want the "Sign Up" link to show up in the header at the top of *every* page, which means the link needs to get generated by the existing `app/views/layouts/_header.html.erb` partial file

Start by using the route helper method to generate the "Sign Up" link anywhere inside the `nav` section of the header, just to get a quick win.

Answer:

```
<%= link_to "Sign Up", signup_path, class: "button" %>
```

4. Then to position the link as we did in the video, first change the existing `ul` tag to have a class of `left`, like so:

```
<ul class="left">
  <li>
    <%= link_to "All Movies", movies_path %>
  </li>
</ul>
```

Then below that `ul` tag add another `ul` tag that has a class of `right` with a single `li`, like so:

```
<ul class="left">
  ...
</ul>
<ul class="right">
  <li>
  </li>
</ul>
```

Then move the "Sign Up" link into the right `li`.

Answer:

```
<ul class="left">
  ...
</ul>
<ul class="right">
  <li>
    <%= link_to "Sign Up", signup_path, class: "button" %>
  </li>
</ul>
```

5. Now reload and click the "Sign Up" link. The URL in your browser's address field should be http://localhost:3000/signup. Of course, we get an error because we don't yet have a `new` action, but we know how to fix that.

6. Define a `new` action that instantiates a new `User` object for the sign-up form to use:

Answer:

```
def new
  @user = User.new
end
```

7. Next, create the corresponding `app/views/users/new.html.erb` view template that starts with this:

```
<h1>Sign Up</h1>
```

Then it needs to generate a sign-up form with the following elements:

- a text field to enter the user's name
- an email field to enter the user's e-mail address
- a password field to enter the user's super-secret password
- another password field to confirm the user's password
- a submit button
- also use the existing `app/views/shared/_errors.html.erb` partial to display any validation errors

We'll want to use this same form when editing a user's account (in the next exercise) so go ahead and put all the sign-up form code in an `app/views/users/_form.html.erb` partial file that uses a local variable named `user`.

Answer:

```
<%= form_with(model: user) do |f| %>
  <%= render "shared/errors", object: user %>

  <%= f.label :name %>
```

```
    <%= f.text_field :name, autofocus: true %>

    <%= f.label :email %>
    <%= f.email_field :email %>

    <%= f.label :password %>
    <%= f.password_field :password %>

    <%= f.label :password_confirmation, "Confirm Password" %>
    <%= f.password_field :password_confirmation %>

    <%= f.submit %>
  <% end %>
```

It's worth noting that you could use a standard `text` field for entering the email address, but using the HTML 5 `email` field gives a better user experience on some mobile devices. For example, iOS devices display a keyboard with the `@` symbol on the primary screen.

8. Then, back in the `new.html.erb` template, render the `form` partial and pass it a local variable named `user` that has the value of `@user`. Remember, you don't use the underscore in the name when rendering the partial.

   Answer:

   ```
   <h1>Sign Up</h1>
   <%= render "form", user: @user %>
   ```

9. Reload to make sure the form shows up as you'd expect before moving on.

10. Next, define a `create` action that uses the submitted form data to create a new user record in the database. If the user is successfully created, redirect to their profile page and show a cheery flash message to let the user know their account was created. Otherwise, if the user is invalid, redisplay the sign-up form.

    Hint:

    Remember that you'll also need to define a private `user_params` method that returns the hash of parameters that are permitted to be mass-assigned by form data: `name`, `email`, `password`, and `password_confirmation`. Otherwise, the values for the fields will get ignored, which is a really tricky bug to hunt down!

    Answer:

    ```
    def create
      @user = User.new(user_params)
      if @user.save
        redirect_to @user, notice: "Thanks for signing up!"
      else
        render :new, status: :unprocessable_entity
      end
    end

    private

    def user_params
      params.require(:user).
        permit(:name, :email, :password, :password_confirmation)
    end
    ```

11. Now use the sign-up form to sign up a new user. Try the following combinations for posterity:

    - Leave the password and confirmation fields empty
    - Fill in a password, but leave the confirmation blank
    - Fill in a password and confirmation that do not match
    - Fill in a password and matching confirmation

    Make sure you end up creating a new user, landing up on their profile page. Home sweet home!

Solution

The full solution for this exercise is in the `user-signup` directory of the code bundle.

Bonus Round

Show "Member Since" Date

As a nice touch, on the user's profile page show the month and year that the user became a member (created an account) on our site. Format the "member since" date as "January 2019", for example.

Hint:

Remember that the date and time at which a user record was created in the database is accessible in the `created_at` attribute. Use the Ruby `DateTime.strftime` method to format it properly.

Answer:

```
<h3>Member Since</h3>
<p>
  <%= @user.created_at.strftime("%B %Y") %>
</p>
```

Add a Profile Image

If you want to give the user profile page a bit more personality, you might consider adding a profile image for each user. A popular way to do that is by integrating with the free Gravatar service. Gravatar lets you upload your preferred profile image (called a *global avatar* image) to the Gravatar site and associate that image with a particular email address. Then when you create a user account on another site using that same email address, the site can use the Gravatar service to show your preferred profile image. It's really convenient because you can register your profile image with Gravatar once and the image automatically follows you to any Gravatar-enabled site.

It's relatively easy to Gravatar-enable an app, and we'll get you started...

1. First, to access a user's profile image, we need to generate an MD5 hash of the user's email address. To do that, add the following method to your `User` model:

   ```
   def gravatar_id
     Digest::MD5::hexdigest(email.downcase)
   end
   ```

   That method simply returns a string that represents the hashed value for the email address. For example, for Mike's email address we'd get back the string `58add23fa01eae6d736adce86e07ae00`. For every unique email address, the method will return a consistent and unique hashed value. Think of it as your unique Gravatar id, which is why we named the method as such.

2. Then to request the associated profile image that's stored on Gravatar's site, we use a URL with the form http://secure.gravatar.com/avatar/gravatar-id where the `gravatar-id` part is replaced with a particular user's gravatar id. For example, Mike's profile image is at http://secure.gravatar.com/avatar/58add23fa01eae6d736adce86e07ae00. Open that URL in a new browser window and you get Mike's mugshot. Now, to actually show Mike's profile image on his profile page, we need to generate an image tag for the image that lives on Gravatar's site.

   To do that, write a custom view helper named `profile_image` that takes a user object as a parameter. It needs to generate a string that represents the URL for that user's Gravatar image and then use that URL to generate and return an image tag for the image. Put the helper method in the `users_helper.rb` file.

   Answer:

   ```
   def profile_image(user)
     url = "https://secure.gravatar.com/avatar/#{user.gravatar_id}"
     image_tag(url, alt: user.name)
   end
   ```

3. Next, update the user profile page to call the `profile_image` helper so that the user's profile image is shown on the page.

   Answer:

   ```
   <section class="user">
     <%= profile_image(@user) %>
     <h1><%= @user.name %></h1>
     <h2><%= mail_to(@user.email) %></h2>
   </section>
   ```

4. Reload a user profile page and you should either see the default Gravatar image (a blue square) or an actual profile image if the user's email has already been associated with a Gravatar image. Now might be a good time to create your own Gravatar image!

Check the Log File

We've been diligent about not storing plain-text passwords in our database, but we also need to be careful not to expose them in other areas of our application.

For example, every time a form is submitted Rails automatically records the submitted form data in the log file. In development mode, everything is logged in the `log/development.log` file which is also displayed in the command window where your app is running.

Check out the log file and scroll back to the part where you signed up a new user. You should see something like this:

```
Parameters: {"authenticity_token"=>"[FILTERED]", "user"=>{"name"=>"Larry", "email"=>"larry@example.com", "password"=>"[FILTERED]", "password_c
```

In particular, notice that the submitted `password` and `password_confirmation` form parameters were masked as `[FILTERED]` so that the actual values aren't displayed in the log file. This is another example of the Rails defaults trying to help us do the right thing. It works because of the following code in the `config/initializers/filter_parameter_logging.rb` file:

```
Rails.application.config.filter_parameters += [
  :passw, :secret, :token, :_key, :crypt, :salt, :certificate, :otp, :ssn
]
```

That line simply appends the key `:passw` (and others) to the `filter_parameters` array. Then, before the `params` hash gets logged, the values for all the keys matching the regular expression `/passw/` get replaced with the string `[FILTERED]`. It's as if someone used a big black marker to hide all the classified information!

Passwords are the most common parameter that need to be filtered from the log files, so Rails takes care of that for you. But you'll want to consider filtering other sensitive parameters specific to your application.

Wrap Up

Nicely done! Folks are now able to create user accounts **complete with super-secret passwords** and view a user's profile page. Next up, we'll allow users to edit their account information and even delete their account.

Edit User Account

Exercises

Objective

Now that new users can sign up, they probably also want to be able to edit their account information, and (dare we say) perhaps even delete their account.

Since this is mostly review, feel free to just go for it! There's nothing surprising here. The goal is just to finish building out the UI for user accounts. This will set the stage for more account-related features we'll implement in upcoming exercises.

1. Edit User Account Information

What controller actions and templates will you need to edit a user's account information?

Answer:

an `edit` action that renders an edit form pre-filled with the user's account info (it's the same form used to create a user account) and an `update` action that accepts the form data and uses it to update a user in the database if the user is valid

1. First we need to put an "Edit Account" link on the user profile page. Use a route helper method to generate that link on the `show` page (which we're calling the user profile page). To apply our CSS styling rules, put it in a `div` with a class of `actions`.

   Answer:

   ```
   <section class="user">
     <h1><%= @user.name %></h1>
     <h2><%= mail_to(@user.email) %></h2>

     <div class="actions">
       <%= link_to "Edit Account", edit_user_path(@user),
                     class: "button edit" %>
     </div>
   </section>
   ```

2. Then define the `edit` action which finds the user matching the `id` parameter in the URL so we can populate the edit form with the user's existing information.

   Answer:

   ```
   def edit
     @user = User.find(params[:id])
   end
   ```

3. Now that we have the user we want to edit, create the corresponding `app/views/users/edit.html.erb` view template. It needs to display the same form we used to create a new user, which we already have in a partial that expects a `user` local variable. (Don't you love it when a plan comes together?) So use that partial to generate the edit form.

   Answer:

   ```
   <h1>Edit Account</h1>
   <%= render "form", user: @user %>
   ```

4. Back in your browser, revel in your work by clicking the "Edit Account" link for a user. You should see a form pre-filled with the user's name and email, but the password fields are blank... as they should be!

5. Next, define the `update` action to use the submitted form data to update the user in the database. If the user is successfully updated, redirect to their profile page with a flash message confirming that their account was successfully updated. Otherwise, if the form data is invalid, redisplay the edit form so the user can give it another try.

   Answer:

   ```
   def update
     @user = User.find(params[:id])
     if @user.update(user_params)
       redirect_to @user, notice: "Account successfully updated!"
     else
       render :edit, status: :unprocessable_entity
     end
   end
   ```

6. Then, back in your browser, change the user's name and/or email, but leave the password fields blank. You should get redirected to the user's profile page and see the updated information.

7. Now edit the user account again, and try typing something in the password field. Submit the form and you should get password validation errors. If you type *anything* into the password fields, then both fields are required and must match. In other words, when updating a user the password-related validations only run if you try to change the password. That's another nice touch that comes courtesy of using `has_secure_password`.

8. Now for a bit of customization. You may have noticed on "Edit Account" form that the default submit button is labeled "Update User". If you then look at the "Sign Up" form, the submit button is labeled "Create User". The form builder is smart enough to label the button based on whether the `user` object represents an existing user already stored in the database (we're editing it) or a new user not yet stored in the database (we're creating it). If `user` is a new record, the button is generated with the label "Create User"; otherwise, the label is "Update User".

   The default button name is convenient, but suppose we want to use account vernacular and have the button say "Create Account" and "Update Account", respectively. To do that, simply locate the following line in the form partial:

   ```
   <%= f.submit %>
   ```

   Then replace that line with the following lines:

   ```
   <% if user.new_record? %>
     <%= f.submit "Create Account" %>
   <% else %>
     <%= f.submit "Update Account" %>
   <% end %>
   ```

   All we do here is call the `new_record?` method that's available on all ActiveRecord objects. It returns `true` if the object hasn't already been saved to the database and `false` otherwise. Then depending on the answer, we pass a string label to the `submit` method. It's a small thing, but users tend to think in terms of their *account* and

using words that align with their thinking is reassuring.

2. Delete User Accounts

We hope it doesn't happen often, but users may want to delete their account. What do you need to make that possible?

Answer:

a "Delete Account" link and a `destroy` action that deletes the user from the database and redirects to the user somewhere, such as the movie listing page

1. Start by putting a "Delete Account" link on the user profile page in the `div` with a class of `actions`.

   Hint:

   Remember, each controller action is triggered by a unique combination of the HTTP verb and the path. According to the routes, to call the `destroy` action the HTTP verb needs to be DELETE. But browsers can't send DELETE requests natively, so you'll need to generate a link that explicitly sets the `turbo_method` attribute to `:delete`. It's also a good idea to use the `turbo_confirm` attribute to show a confirmation dialog before permanently deleting the user's account!

   Answer:

   ```
   <div class="actions">
     <%= link_to "Edit Account", edit_user_path(@user), class: "button edit" %>
     <%= link_to 'Delete Account', @user, class: "button delete",
                 data: { turbo_method: :delete, turbo_confirm: "Permanently delete your account!?" } %>
   </div>
   ```

2. Then define the `destroy` action. Once the user's record has been destroyed, redirect to the home page and flash an alert message confirming that the account was successfully deleted. Make sure to set the redirection `status` to `:see_other` so the redirect to the movies page will be a `GET` request, rather than a `DELETE` request.

   Answer:

   ```
   def destroy
     @user = User.find(params[:id])
     @user.destroy
     redirect_to movies_url, status: :see_other,
       alert: "Account successfully deleted!"
   end
   ```

3. Now go back to the browser and click the "Delete Account" link. You should end up on the application home page. Go to the user listing page and the user you deleted should **not** be displayed in the listing.

4. Finally, you might want to use your new account management web interface to re-create the user you just deleted. :-)

Solution

The full solution for this exercise is in the `edit-user-account` directory of the code bundle.

Bonus Round

Add a Username Field

In addition to a full name and email, some sites also allow users to set a unique username. It's your online nickname or screenname. For example, Mike's Twitter username is "clarkware".

Add a `username` field to the `users` database table, and allow users to specify a username when creating (and editing) their account. Usernames must be present and only consist of letters and numbers (alphanumeric characters) without spaces. Also, no two users can have the same username in the database. Treat usernames as being case-insensitive.

Try it on your own first (after all, we're in the bonus round!) and then follow the steps below if you need a hand.

1. Generate a migration to add a `username` column to the `users` table.

   Answer:

   ```
   rails g migration AddUsernameToUsers username:string
   ```

2. Declare appropriate validations in the `User` model.

   Hint:

   The regular expression for the `format` validation can be a bit tricky, so go ahead and use `/\A[A-Z0-9]+\z/i` which matches alphanumeric characters only.

   Answer:

   ```
   validates :username, presence: true,
                        format: { with: /\A[A-Z0-9]+\z/i },
                        uniqueness: { case_sensitive: false }
   ```

3. Update the form partial to include a text field for entering the username.

   Answer:

   ```
   <%= f.label :username %>
   <%= f.text_field :username, placeholder: "Alphanumeric characters only!" %>
   ```

4. Don't forget to add the `username` field to the list of permitted parameters so that the username can be assigned from form data.

Answer:

```
def user_params
  params.require(:user).
    permit(:name, :email, :password, :password_confirmation, :username)
end
```

5. Update the `users/show.html.erb` view template to display the user's username.

   Answer:

```
<%= @user.username %>
```

Wrap Up

And with that, we have a complete user interface for managing user accounts! Working incrementally through it allowed us to review the following Rails fundamentals:

- Creating a new resource using the generator
- Applying a migration file
- Declaring validations in the model
- Using the Rails console to create records in our database
- Defining actions to support the conventional resource routes: `index`, `show`, `new`, `create`, `edit`, `update`, and `destroy`
- Linking pages together
- Using partials to reduce view-level code duplication
- Routing custom URLs and using route helper methods
- And providing feedback with flash messages

With all this now in place, in the next section we'll allow users to actually sign in to their account. Open sesame...

Sign In

Exercises

Objective

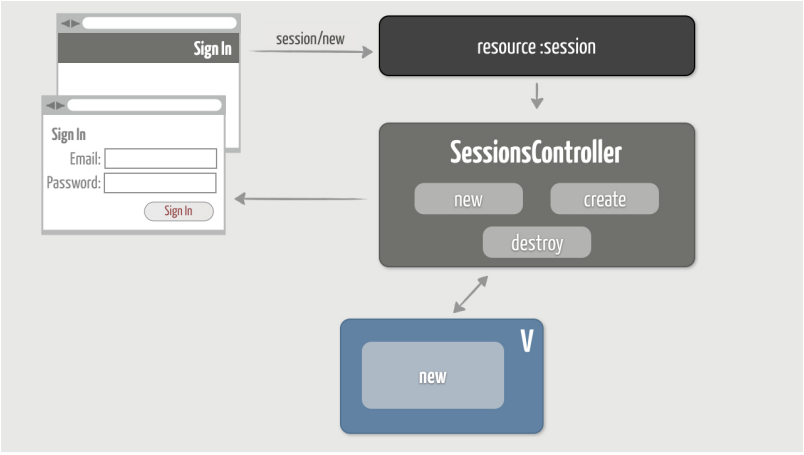
Now that users can create an account, the next logical step is to let registered users sign in using their email and password. Once signed in, we'll then need to identify them as they browse from page to page, until they eventually sign out.

If you've spent any time on the web, you've gone through the typical sign-in process time and again. From the user's perspective it seems like such a trivial thing, but there's actually a lot going on behind the scenes. Indeed, getting the entire sign-in process working correctly involves arranging a number of moving parts. As is our development style, we'll take an incremental approach.

Our first step is simply to get to the point of displaying the sign-in form. To do that, we'll need to:

1. Create a `SessionsController` to manage the session resource
2. Add specific routes for a singular `session` resource
3. Generate a "Sign In" link in the header of every page
4. Define a `new` action in the `SessionsController` that renders a sign-in form with email and password fields
5. Define a bare-bones `create` action in the `SessionsController` that, for now, simply captures the submitted form data
6. While we're at it, also define an empty `destroy` action which we'll implement in a future exercise

Here's visually what we want:



1. Generate the SessionsController

Start by creating a `SessionsController` with empty `new`, `create`, and `destroy` actions.

Hint:

You can use the controller generator as we did in the video by using `rails g controller sessions`, but it's almost as easy (and good practice) to create the `app/controllers/sessions_controller.rb` file yourself and then define the empty actions.

Answer:

```
class SessionsController < ApplicationController
  def new
  end

  def create
  end

  def destroy
  end
end
```

2. Add Session Routes

Then we need to add routes for our new `session` resource. These routes will be slightly different than the routes for our other resources.

The **first difference** is we only need routes for the `new`, `create`, and `destroy` actions.

The **second difference** is the session-related URLs won't include an `:id` placeholder when referencing an individual session. For example, to delete a session we don't need to include the session's id (its database primary key) in the URL. That's because, unlike the other resources in our app, the session resource won't be stored in the database. Instead, the session data will be stored in a cookie that automatically gets sent back to our app with every request. So we don't need a `sessions` database table or a `Session` model. And if sessions aren't in the database, then it doesn't make sense to include a session id in the session-related URLs. For any particular request, there's only *one* session. Rails calls this a *singular* resource.

1. In the `config/routes.rb` file, define the specific routes for a singular `session` resource.

   Answer:

   ```
   resource :session, only: [:new, :create, :destroy]
   ```

2. Go ahead and check out the defined routes to see what that gives you. You should see the following *three* session-related routes:

   ```
   Helper              HTTP Verb        Path                  Controller#Action
   new_session_path       GET        /session/new(.:format)     sessions#new
   session_path           DELETE     /session(.:format)         sessions#destroy
                          POST       /session(.:format)         sessions#create
   ```

   Notice that none of the URLs require an `:id` parameter as we're used to seeing with other resource routes. You'll also notice that the other conventional routes (such as listing sessions) weren't generated since we used the `only` option to specify specific routes.

3. Before moving on, pay careful attention to the naming in the generated routes. The route helper methods and URLs use the singular form (`session`). However, the controller name is the plural form (`sessions`). Rails always keeps you on your toes when it comes to singular and plural naming!

3. Generate a "Sign In" Link

You expect a convenient "Sign In" link at the top of every web page, so the next step is to put one there...

1. Use a route helper method to generate a "Sign In" link to the left of the existing "Sign Up" link.

   Hint:

   You want the "Sign In" link to run the `new` action. The route helper method named `new_session_path` generates a URL to that action. Use that route helper method to generate the link in the `app/views/layouts/_header.html.erb` partial.

   Answer:

   ```
   <ul class="right">
     <li>
       <%= link_to "Sign In", new_session_path, class: "button" %>
     </li>
     <li>
       <%= link_to "Sign Up", signup_path, class: "button" %>
     </li>
   </ul>
   ```

2. Over in your browser, reload any page and you should see a "Sign In" link. Click it and the URL should change to http://localhost:3000/session/new.

So the route is recognized and runs the `new` action of the `SessionsController`, but of course we don't yet have a view template for that action.

4. Create the Sign-In Form

Next up we need to create the sign-in form with the following elements:

- an email label and field to enter the user's name
- a password label and field to enter the user's password
- a submit button appropriately named "Sign In"

1. Start by creating the `app/views/sessions/new.html.erb` view template that simply displays "Sign In" in an `h1` tag.

   Answer:

   ```
   <h1>Sign In</h1>
   ```

2. Now create the form with just an email label and field.

Remember, we don't have a model for sessions so this form won't be bound to a model object. So when calling the `form_with` method to generate the form, you'll need to use the `url` option rather than the `model` option we've used previously. The value of the `url` option needs to be the URL where the form will POST the form data. Use a route helper method to generate that URL.

Answer:

```
<h1>Sign In</h1>

<%= form_with(url: session_path) do |f| %>
  <%= f.label :email %>
  <%= f.email_field :email, autofocus: true %>
<% end %>
```

3. Now add a password label and field.

   Answer:

```
<h1>Sign In</h1>

<%= form_with(url: session_path) do |f| %>
  <%= f.label :email %>
  <%= f.email_field :email, autofocus: true %>

  <%= f.label :password %>
  <%= f.password_field :password %>
<% end %>
```

4. Finally, add a submit button.

   Answer:

```
<h1>Sign In</h1>

<%= form_with(url: session_path) do |f| %>
  <%= f.label :email %>
  <%= f.email_field :email, autofocus: true %>

  <%= f.label :password %>
  <%= f.password_field :password %>

  <%= f.submit "Sign In" %>
<% end %>
```

5. Capture Submitted Form Data

In this exercise, we just want the `create` action to capture the submitted form data. We'll finish it off in the next exercise.

1. Just to see the form data that's submitted, add a `fail` line to the `create` action so that the submitted form data is dumped onto an error page:

```
def create
  fail
end
```

2. Then submit the form with data and you should get an error (or a debugging page, depending on how you look at it). Check out the stuff under the "Request" heading and you should see something like this:

```
{"authenticity_token"=>"[FILTERED]",
 "email"=>"lucy@example.com",
 "password"=>"[FILTERED]",
 "commit"=>"Sign In"}
```

   Notice that the email and password form field values are captured in the corresponding `email` and `password` parameters. The password value is automatically filtered when displayed on the error page, but rest assured that the submitted password is actually in the `password` parameter.

   So at this point the `params` hash contains everything we need to authenticate users by email and password. In the next exercise, we'll do exactly that! For now, you're good to go if you see values for the `email` and `password` parameters on the error page.

Solution

The full solution for this exercise is in the `sign-in` directory of the code bundle.

Bonus Round

Define a Custom Sign In Route

Currently, when you click the "Sign In" link the URL in the browser's address bar is http://localhost:3000/session/new. That's fairly descriptive if you understand sessions, but you might also want to support the more-friendly URL http://localhost:3000/signin.

1. Add a route that dispatches the URL http://localhost:3000/signin to the Sign In form.

   Answer:

```
get "signin" => "sessions#new"
```

2. Then update the "Sign In" link in the header to use a route helper method to generate the "Sign In" link as `/signin` rather than `/session/new`.

   Answer:

```
<%= link_to "Sign In", signin_path, class: "button" %>
```

Add Another "Sign Up" Link

What happens if a person lands up on the Sign In form but they don't already have an account? Well, we'd be foolish not to encourage them to create an account. So add a friendly link on the Sign In form that lets them hop right over to the Sign Up form.

Hint:

You already have a route helper method that generates the URL to sign up, so use it to generate the link!

Answer:

```
<h1>Sign In</h1>
<p>
  No account yet? <%= link_to "Sign up!", signup_path %>
</p>

<%= form_with(url: session_path) do |f| %>
  ...
<% end %>
```

Require Sign-In Fields

If you want to get fancy with HTML 5, you can use the new `required` attribute on input fields. Browsers that support HTML 5 won't allow a form to be submitted until all the fields marked as being required have a value. Those browsers also highlight the fields with missing values.

That's kinda convenient in the case of a Sign In form since we don't have a corresponding model with validations. It also saves a trip back to the app if a field is left blank.

So mark the email and passwords fields as being required by adding the `required: true` option when generating the fields.

Answer:

```
<%= f.email_field :email, autofocus: true, required: true %>

<%= f.password_field :password, required: true %>
```

Wrap Up

That's a great start! Now that we have a familiar sign-in form, next we need to try to authenticate the user when they submit the form...

Dive Deeper

You might find it helpful to refer to the form_with documentation which explains each of the options with handy examples.

Authentication

Exercises

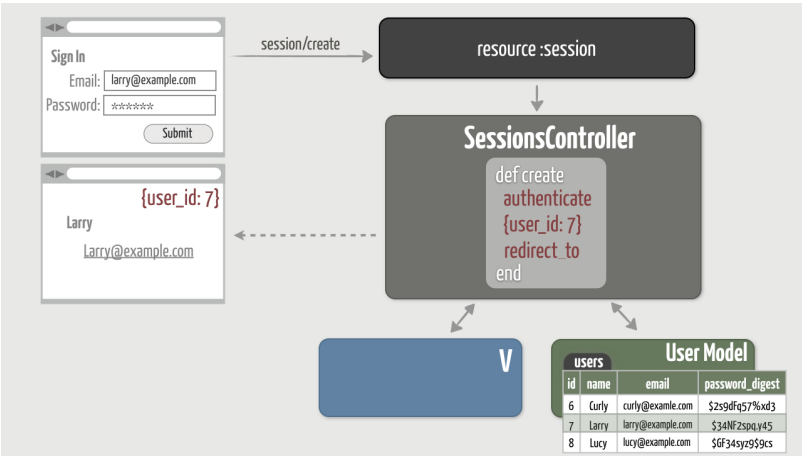
Objective

So far we've displayed the sign-in form where a user can enter their credentials, but we haven't actually *authenticated* the user yet. So here's where the rubber meets the road! When a user submits their email and password, we need to sign them in.

To finish the sign-in process, we'll need the `create` action to do the following:

1. Find a user in the database with the submitted e-mail address
2. Verify that the submitted password is correct for that user
3. Store the authenticated user's id in the session
4. Redirect to the user's profile page
5. If the submitted email and/or password don't match a user in the database, we'll redisplay the sign-in form

Here's visually what we want:



It's a fairly straightforward exercise that follows what we did in the video, so let's jump right into it!

1. Authenticate in the Console

Before we start filling in the `create` action, let's practice the first two authentication steps in the console using example email and password values...

1. First, find a user you created previously by their e-mail address and assign that user to a `user` variable.

   Answer:

   ```
   >> user = User.find_by(email: "lucy@example.com")
   ```

2. Next, verify a given password is correct for that user. As you saw in the video, the `has_secure_password` line we added to our `User` model earlier was kind enough to give us an `authenticate` method that makes this trivial. Remember, it's an *instance method* that you can call on any `User` object.

   First try calling `authenticate` with an incorrect password.

   Answer:

   ```
   >> user.authenticate("guess")
   => false
   ```

   It should return `false`, indicating that the password was incorrect.

3. Then try authenticating with a correct password.

   Answer:

   ```
   >> user.authenticate("abracadabra")
   => #<User:0x00000001063f6928
    id: 2,
    name: "Lucy",
    email: "lucy@example.com",
    ...>
   ```

   It should return the `User` object, indicating that the password was correct.

Hey, that's pretty easy! Now let's bring this full circle. When this user created an account, they gave us a plain-text password which got encrypted (hashed) and saved in the `password_digest` column. Now when that user sign ins, she gives us the same plain-text password which we pass to the `authenticate` method. It then encrypts the password and compares the result to what's in the `password_digest` column. If the passwords don't match, then `authenticate` returns `false`. Otherwise, if the passwords do match, the `User` object is returned.

So by comparing encrypted passwords instead of plain-text passwords, we're able to authenticate users without ever storing plain-text passwords. And that's critically important!

2. Implement the Create Action

Now we're ready to step back into the `create` action and finish it off. First, it needs to perform the same authentication steps we just did in the console, only this time using the submitted email and password that came in from the sign-in form. If the user is successfully authenticated, we then need to store their user id in the session and redirect to their profile page. Otherwise, if authentication fails, we need to redisplay the sign-in form.

1. If you still have that `fail` line hanging around in `create`, go ahead and zap it.

2. With a clean slate, start by finding the user that matches the submitted email address. Assign that user to a variable called `user`.

   Hint:

   Use `params[:email]` to access the email value submitted by the form.

   Answer:

   ```
   def create
     user = User.find_by(email: params[:email])
   end
   ```

3. Then verify that the submitted password is correct for that user, just like you did in the console.

   Hint:

   Use `params[:password]` to access the password value submitted by the form, and pass that value to the `authenticate` method.

   Answer:

   ```
   def create
     user = User.find_by(email: params[:email])
     user.authenticate(params[:password])
   end
   ```

4. Next, set up an `if/else` conditional to check whether the user was authenticated or not. Keep in mind:

   - Before calling `authenticate`, you need to make sure the `User` object returned by `find_by` is not `nil`. Otherwise, calling the `authenticate` method will cause an error.
   - The `authenticate` method will return `false` if the password isn't correct.

   Hint:

Remember that a user is successfully authenticated only if a user exists with the given email **and** that user has the given password. So you need the `if` to check two things: the `user` variable must reference a non-nil value *and* the `authenticate` method must return a non-nil value. To do that, use the Ruby `&&` (logical *and*) operator. Remember that in Ruby, anything other than `nil` and `false` is `true`.

Answer:

```
def create
  user = User.find_by(email: params[:email])
  if user && user.authenticate(params[:password])
  else
  end
end
```

5. If the user is authenticated, record that fact in the session by assigning the user's `id` to the `:user_id` session key. We'll use the presence of that key/value pair in the session to indicate that the user is signed in. Then redirect to that user's profile page with a cheery flash notice such as "Welcome back, Lucy!".

   Answer:

```
def create
   user = User.find_by(email: params[:email])
   if user && user.authenticate(params[:password])
     session[:user_id] = user.id
     redirect_to user, notice: "Welcome back, #{user.name}!"
   else
   end
 end
```

6. Finally, handle the case where the user is not authenticated by redisplaying the sign-in form with a flash alert to prompt the user to try again. Don't forget to set the response `status` to `:unprocessable_entity`.

   Hint:

   You'll need to use `flash.now` in order for the flash message to show up immediately when the sign-in form is redisplayed. When we used a flash in previous situations, it came before a redirect. Then when the redirect happened, it picked up the flash in the subsequent request. But in this case, we're re-rendering the form which doesn't send a separate request. Using `flash.now` makes the flash message available in the same request rather than waiting for the next request.

   Answer:

```
def create
  user = User.find_by(email: params[:email])
  if user && user.authenticate(params[:password])
    session[:user_id] = user.id
    redirect_to user, notice: "Welcome back, #{user.name}!"
  else
    flash.now[:alert] = "Invalid email/password combination!"
    render :new, status: :unprocessable_entity
  end
end
```

7. Finally, hop back into a browser and give this a whirl:

   - Try signing in with an **invalid** email/password combination, and you should see the red flash message. Bzzt... wrong answer!
   - Then sign in an **existing user** and you should end up on their profile page with a green flash message confirming that you're signed in. Welcome back!

Solution

The full solution for this exercise is in the `authentication` directory of the code bundle.

Bonus Round

Quick Session Facts

Knowledge is power, so here are a few quick facts about sessions to keep in mind:

- The session cookie expires when the browser is closed.
- The session cookie is limited to 4kb in size. So avoid storing large objects in the session, and instead store an object's id. Then use that id to look up the object in the database, as we did with the `User` object.
- The session cookie can't be forged by a hacker. It's cryptographically signed to make it tamper-proof, and Rails will reject the cookie if it has been altered in any way. The session cookie is also encrypted, so you can't even read what's inside the session cookie. That prevents a malicious person from trying to impersonate a signed-in user by changing the user id in the session, for example.
- That being said, you should *never* store sensitive information such as a password in a session!

Use Alternate Credentials

In a previous bonus exercise, you may have added a `username` field to the `users` database table. Arrange things to allow a user to sign in using either their e-mail address *or* their unique username, and their password of course.

1. First, on the Sign In form you'll need to change the `email_field` to `text_field` so the user can either enter their email or their username. That field is currently named `email`, so you'll also want to rename it to something like `email_or_username` since its value can be either. Finally, change the name of the label so folks know they can use either their email or username.

   Answer:

```
<%= f.label :email_or_username %>
<%= f.text_field :email_or_username, autofocus: true, required: true %>
```

2. Then change the `create` action to attempt to find the user by either their provided email or username. The Ruby `||` operator comes in really handy here!

   Answer:

```
def create
  user = User.find_by(email: params[:email_or_username]) ||
         User.find_by(username: params[:email_or_username])
  if user && user.authenticate(params[:password])
    ...
  else
    ...
  end
end
```

Wrap Up

We're making good progress! We're able to sign in users and track them in the session. However, it's not very evident who's currently signed in. It would feel a lot more welcoming if we displayed the current user's name in the header. And while we're at it, we can remove those pesky "Sign Up" and "Sign In" links once a user is signed in. We'll do exactly that in the next section...

Current User

Exercises

Objective

So you've signed in, but how do you really know you're still signed in when you browse from page to page? There's currently no reassuring feedback indicating that the app knows who you are.

So let's fix that. If a user is signed in, we'll display their name in the header of every page with a link to their profile page. We'll also remove those "Sign Up" and "Sign In" links, since they're kinda meaningless if you're already signed in.

1. Display the Current User

First we want to show the user's name in the header, so let's start simple and build up from there...

1. To set the stage, begin by signing in so you have a user id in the session. You'll end up on their profile page. Then go to the application home page, for example, and you'll notice there's absolutely no indication that you're signed in.

2. Get some practice reading session data by simply showing the currently signed-in user's id in the header (`app/views/layouts/_header.html.erb`).

   Hint:

   Remember, we used the session key `:user_id` to store the user's id. So to read the user's id, you use `session[:user_id]`.

   Answer:

```
<ul class="right">
  <li>
    <%= session[:user_id] %>
  </li>
  <li>
    <%= link_to "Sign In", signin_path, class: "button" %>
  </li>
  <li>
    <%= link_to "Sign Up", signup_path, class: "button" %>
  </li>
</ul>
```

3. Now reload and you should see the user's id displayed at the top of *every* page.

4. But of course we really want to display the user's *name* rather than their id. For now, go ahead and add the code to do that directly inside the view template.

   Hint:

   Find the user record in the database that has the `id` matching the `:user_id` in the `session` hash. Then read that user's `name` attribute.

   Answer:

```
<ul class="right">
  <li>
    <%= User.find(session[:user_id]).name %>
  </li>
  <li>
    <%= link_to "Sign In", signin_path, class: "button" %>
  </li>
  <li>
    <%= link_to "Sign Up", signup_path, class: "button" %>
  </li>
</ul>
```

5. Finally, reload again and you should now see the user's name displayed at the top of every page.

2. Write a Custom View Helper

It's working, but we're never satisfied with that. To get it to work we put query code in a view template, and that's always a red flag. Looking forward, other parts of our app will also need to be able to get the currently signed in user. And as we learned earlier in the course, a great way to reuse bits of view logic is to write custom view helpers. In this case, we want to define a helper named `current_user` that returns the currently signed-in user.

1. Start by replacing the offending line in the header with what we really want it to look like—using a yet-to-be-defined `current_user` helper:

   ```
   <ul class="right">
     <li>
       <%= current_user.name %>
     </li>
     <li>
       <%= link_to "Sign In", signin_path, class: "button" %>
     </li>
     <li>
       <%= link_to "Sign Up", signup_path, class: "button" %>
     </li>
   </ul>
   ```

   There, that's more expressive and neatly encapsulates the query.

2. Next, define that helper method. As it's an application-wide helper, and not specific to any one controller, this helper belongs in the `ApplicationHelper` module found in the `app/helpers/application_helper.rb` file.

   Hint:

   The `find` method will raise an exception if passed a value of `nil`. So it's important to check that the session actually has a user id before calling `find`. Doing this also ensures that the user id in the session corresponds to an *actual* user record in the database.

   Answer:

   ```
   module ApplicationHelper
     def current_user
       User.find(session[:user_id]) if session[:user_id]
     end
   end
   ```

3. Refresh any page and you should still see the user's name. But now your code is better organized, and that makes you feel better, too.

4. Having a helper method that returns the current user makes it easy to link the user's name to their profile page. So go ahead and do that!

   Answer:

   ```
   <ul class="right">
     <li>
       <%= link_to current_user.name, current_user %>
     </li>
     <li>
       <%= link_to "Sign In", signin_path, class: "button" %>
     </li>
     <li>
       <%= link_to "Sign Up", signup_path, class: "button" %>
     </li>
   </ul>
   ```

5. Now on every page you should be able to click the signed-in user's name to go directly to their profile page.

3. Toggle the Header

Hey, what about those "Sign In" and "Sign Up" links? If a user is already signed in, it doesn't make any sense to show those in the header.

Arrange things so that the header changes based on the sign-in status. When a user is signed in, they should only see their name in the header. When a user is not signed in, they should see the "Sign In" and "Sign Out" links in the header.

Hint:

You know a user is signed in if the `current_user` method returns a non-nil value.

Answer:

```
<ul class="right">
  <% if current_user %>
    <li>
      <%= link_to current_user.name, current_user %>
    </li>
  <% else %>
    <li>
      <%= link_to "Sign In", signin_path, class: "button" %>
    </li>
    <li>
      <%= link_to "Sign Up", signup_path, class: "button" %>
    </li>
  <% end %>
</ul>
```

Before moving on, check that the header changes as you'd expect based on whether a user is signed in or signed out. We don't yet have a way to sign out a user (that's coming up next!), so for now you'll need to restart your browser which expires the session cookie.

4. Automatically Sign In After Sign Up

As things stand, when the user signs up for a new account they aren't automatically signed in. They end up on their profile page, but their name doesn't show up in the header. Instead, they have to sign in using the same information they just used to sign up and that's pretty annoying!

Fix that by automatically signing in the user when they create an account.

Hint:

Change the `create` action in the `UsersController` so that it puts the user's id in the session once the user record has successfully been created.

Answer:

```
def create
  @user = User.new(user_params)
  if @user.save
    session[:user_id] = @user.id
    redirect_to @user, notice: "Thanks for signing up!"
  else
    render :new, status: :unprocessable_entity
  end
end
```

Since you already have the header changing based on the sign-in status, when you create a new user you should immediately get feedback that they're signed in!

5. Performance Tip

When you think you've finished a feature, it's always a good idea see what's going on in the log file when that feature is exercised. In particular, it's wise to keep a check on the number of database queries required to show any particular page.

So let's see what's happening in query land for this feature...

1. Make sure you're signed in, and then go the application home page (or anywhere except the user profile page).

2. Then scroll back in the log file to the part where the `GET` request was issued for that page. As part of fulfilling that request, you should notice three SQL queries for the same user record:

   ```
   User Load (0.1ms)  SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
   CACHE User Load (0.0ms)  SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
   CACHE User Load (0.1ms)  SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
   ```

   Now, Rails is pretty good about caching query results in the same request. In this case, since these three queries fetch the same user record, Rails caches the result of the first SQL query. That cached result is then returned for the second and third queries. Thus, those two queries are marked as `CACHE`.

   So it's no big deal since the result is cached, but we can avoid those second two queries entirely.

3. If you look back in the header (`app/views/layouts/_header.html.erb`), you'll notice that we're currently calling our `current_user` method three times:

   ```
   <% if current_user %>
     <%= link_to current_user.name, current_user %>
   ```

   And we know that `current_user` performs a `find` query:

   ```
   def current_user
     User.find(session[:user_id]) if session[:user_id]
   end
   ```

   So that's why we see three queries in the log file!

4. We can optimize the `current_user` method by storing the result of calling `User.find` in an instance variable, and only running the query again if that instance variable doesn't already have a value. To do that, change the `current_user` method to set a `@current_user` instance variable using the `||=` operator, like so:

   ```
   def current_user
     @current_user ||= User.find(session[:user_id]) if session[:user_id]
   end
   ```

   This is a Ruby idiom that you'll often see used in Rails applications. Here's what happens in this case: The first time the `current_user` method is called, the `@current_user` instance variable doesn't have a value. So the code on the right-hand side of the `||=` operator (the `User.find` query) gets run and the result is assigned to the `@current_user` instance variable. Then the method implicitly returns the value of that instance variable. So far, so good.

   Here's where things get interesting: If the `current_user` method is called again during the **same request**, then it doesn't bother running the query on the right-hand side of `||=` because `@current_user` already has a value. Instead, it simply returns that value without running the database query.

   It's important to note that you must assign the result of calling `User.find` to an *instance* variable, not a *local* variable. A local variable (a variable without the `@` symbol) would go out of scope when the `current_user` method returned, which would mean the query result wouldn't be stored for use the next time `current_user` was called.

   You'll often hear this idiom referred to as *memoization*. It's a fancy word for a simple concept. The upshot is that `User.find` will only be called once per request, the first time the `current_user` method is called.

5. To observe its effect, reload the page and check out the log file again:

   ```
   User Load (0.1ms)  SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
   ```

Now you should see one query rather than three! All the `CACHE` lines are gone since the `current_user` method now only calls the `User.find` query once per request.

Solution

The full solution for this exercise is in the `current-user` directory of the code bundle.

Bonus Round

Show an "Account Settings" Link

When a user is signed-in, it might be convenient to also put an "Account Settings" link in the header next to the user's name. That way the user can jump directly to the form for editing their account information. And it's really easy to do now that you have a `current_user` method!

Answer:

```
<ul class="right">
  <% if current_user %>
    <li>
      <%= link_to current_user.name, current_user %>
    </li>
    <li>
      <%= link_to "Account Settings", edit_user_path(current_user) %>
    </li>
  <% else %>
    <li>
      <%= link_to "Sign In", signin_path, class: "button" %>
    </li>
    <li>
      <%= link_to "Sign Up", signup_path, class: "button" %>
    </li>
  <% end %>
</ul>
```

Wrap Up

Way to hang in there! Looking back, you learned a lot about sessions in the last three modules.

Sessions are used to uniquely track signed-in users across multiple requests:

- When a user successfully signs in, their unique user id (the primary key in the `users` database table) is stored in a hash called `session`.
- The session hash with the user's id is shuttled back and forth between the app and the browser in a cookie.
- The user's id is removed from the session when they sign out (details coming up in the next section).

Session data is stored in a cookie. As such, this means:

- We don't look up session data in a database table.
- We don't need a Session model.
- We use a singular resource route (`resource :session`).
- We access the contents of the cookie by reading values from the `session` hash.

It's the presence of the user id (or lack thereof) in the `session` hash that allows us to:

- Determine if a user is signed in.
- Look up the user in the database.
- Display that user's name in the header.
- Show "Sign In" and "Sign Up" links as appropriate.

The `SessionsController` has three actions:

- The `new` action renders a sign-in form.
- The `create` action stores an authenticated user's unique id in the `session` hash.
- The `destroy` action removes the user's id from the session.

So next time you visit your favorite site and enter your information in their sign-in form, you'll better appreciate the complexity that's going on behind the scenes!

Up next, we'll address what happens when you sign out, which is quick and easy as compared to signing in.

Sign Out

Exercises

Objective

Last, but by no means least, we need a way for signed-in users to sign out! To do that, we need to:

- Generate a "Sign Out" link in the header
- Implement the `destroy` action in the `SessionsController` so that it removes the user's id from the session

Notice that we don't need to add a new route because remember we already have the singular resource routes. So this turns out to be a short exercise!

1. Generate a "Sign Out" Link

First, let's put a familiar "Sign Out" link in the header...

1. We'll need to use a route helper method to generate that link, so you may want to start by reviewing the defined session routes and identifying the correct route helper method.

2. Using that route helper method, generate a "Sign Out" link next to the signed-in user's name in the header. The link should only show up if a user is currently signed in.

Hint:

You want the "Sign Out" link to run the `destroy` action. According to the routes, to generate that link you'll need to use the `session_path` route helper method **and** set the `turbo_method` data attribute to `:delete`. Alternatively, you could make it a button by using the `button_to` helper and setting the `method` option to `:delete`.

Answer:

```
<% if current_user %>
  <li>
    <%= link_to current_user.name, current_user %>
  </li>
  <li>
    <%= link_to "Sign Out", session_path,
                data: { turbo_method: :delete },
                class: "button" %>
    # or make it a button
    <%= button_to "Sign Out", session_path, method: :delete %>
  </li>
  <% else %>
  ...
<% end %>
```

3. Over in your browser, make sure you're signed in and then reload any page. You should see a "Sign Out" link at the top of every page.

2. Implement the Destroy Action

To sign out the user, all we have to do is remove their user id from the session...

1. Implement the `destroy` action in the `SessionsController`. Once the user is signed out, redirect to the application home page with a flash message that confirms the user is no longer signed in. Don't forget to set the redirection `status` to `:see_other` so that the redirect is a `GET` request.

   Hint:

   To remove something from the session, simply set the key's value to be `nil`. And in this case, the key is `:user_id`.

   Answer:

   ```
   def destroy
     session[:user_id] = nil
     redirect_to movies_url, status: :see_other,
       notice: "You're now signed out!"
   end
   ```

2. Then go back to the browser and click the "Sign Out" link again. You should end up on the movies page with the header status changed to indicate that you're no longer signed in. That's a nice reward for all the groundwork we did in previous exercises!

3. Sign Out When Destroying Accounts

Before we declare victory and move on, we have a small bit of house-keeping. If a signed-in user deletes their account, things go haywire because the session will still contain that user's id. To avoid this messy situation, when a user deletes their account we need to automatically sign them out. That way an invalid user id isn't left hanging around in the session to wreak havoc.

1. Update the `destroy` action in the `UsersController` to sign out the user after destroying it.

   Answer:

   ```
   def destroy
     @user = User.find(params[:id])
     @user.destroy
     session[:user_id] = nil
     redirect_to movies_url, status: :see_other,
                 alert: "Account successfully deleted!"
   end
   ```

2. Then make sure deleting a user account doesn't raise an exception!

Solution

The full solution for this exercise is in the `sign-out` directory of the code bundle.

Wrap Up

A hearty round of applause is in order, we'd say! You now have an effective authentication solution in place that you built step-by-step from scratch by putting all the following components together:

- Creating a user resource complete with a `users` database table, a `User` model with validations, and a `UsersController`.
- Defining actions (and creating respective views) to support the conventional resource routes for interacting with users via the web: `index`, `show`, `new`, `create`, `edit`, `update`, and `destroy`.
- Validating that a user has a confirmed password and storing it securely using Rails conventions: a `password_digest` column to the `users` database table, `has_secure_password` in the `User` model, and installing the `bcrypt` gem.
- Adding routes for a singular `session` resource, creating a `SessionsController` to manage the session resource, and ultimately using the session to uniquely track signed-in users.
- Generating a sign-in form using the `form_with` helper and the `url` option, since the session resource doesn't map to a model object.

- Authenticating a user with the super-easy, super-powerful `authenticate` method.
- Displaying the user's name (and appropriate "Sign In", "Sign Up", or "Sign Out" links) in the header, courtesy of a `current_user` method.
- Implementing a `destroy` action in the `SessionsController` that removes the user's id from the session to effectively complete the authentication process.

Whew, that **is** a lot of progress! If you haven't already, now's a great time to take a break, stretch your legs, and let all this new knowledge settle in your brain.

In the next module, we'll build on this foundation by restricting access to certain areas of the app. For starters, only Daisy should be able to edit her account information. Not Lucy, Larry, or any other user. We'll fix that after the break!

Dive Deeper

- To learn more about sessions, refer to *5 Session* in the Action Controller Overview and *2 Sessions* in the Rails Guides: Ruby On Rails Security Guide.
- If you need an authentication system with a lot more bells and whistles, you might consider using Devise. It supports a number of advanced features out of the box, and hides all the nitty-gritty details from you. Unfortunately, that means Devise feels very magical if you don't have a solid understanding of the basics of authentication. That's why the creators of Devise recommend that you *not* use Devise if you're building your first Rails app. Instead, they advise you start by creating a simple authentication system from scratch. And now you've done exactly that!

Authorization: Part 1

Exercises

Objective

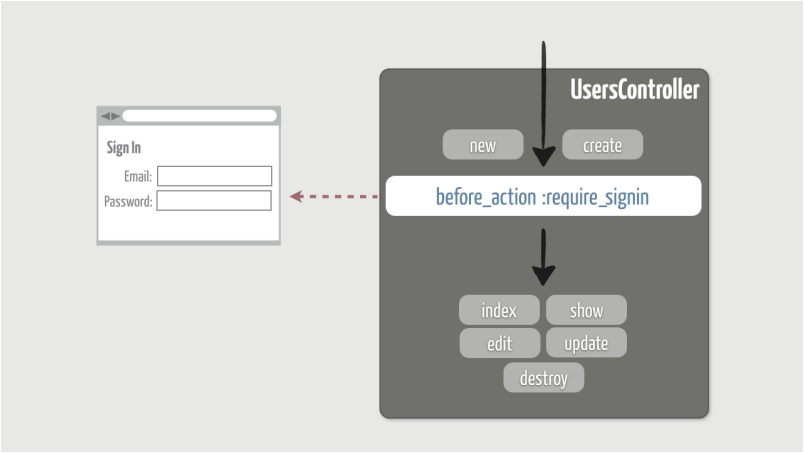
Let's face it, our app is a bit too trusting. As it stands, **anybody**—whether they're signed in or not!—can do some serious damage: delete movies, change information in user accounts, and even permanently delete all the user accounts! But that's all about to change...

First we need to make sure that a user is signed in before allowing them to list users or view a user's profile page. In fact, with the exception of the `new` and `create` actions in the `UsersController`, all user-level actions should be restricted to signed-in users.

This process is commonly referred to as *authorization*. Now, authorization rules tend to vary widely depending on the nature of the application. The good news is once you understand the basic technique, you can apply it as you see fit. Enforcing authorization rules turns out to be fairly straightforward thanks to the `before_action` method in Rails. By defining a `before_action` we can intercept calls to any controller action and automatically run code to determine, given the circumstances, whether the action can indeed be called.

For the purposes of this exercise, we'll define a `require_signin` method that runs before restricted user-level actions to check whether a user is signed in. If a user isn't currently signed in, we'll redirect to the sign-in form and the original action won't run. Otherwise, if the user is signed in, we'll go ahead and let the original action run.

Here's a high-level visual of our goal:



It's like putting a big, bad security guard at the door of every restricted action. He either nods and lets you in, or he furrows his brow and points you to the security desk to collect your badge.

Restrict Access to Users

Anyone should be able to create an account using the `new` and `create` actions in the `UsersController`. However, only signed-in users should be allowed to run the other actions in that controller. To enforce that, we'll start by setting up a gatekeeper...

1. In the `UsersController`, use a `before_action` to call a method named `require_signin` (which we'll implement next) before running any action except `new` or `create`.

   Hint:

   Call the `before_action` method and pass it the name of the method you want to run (as a symbol). By default, a `before_action` is run before *every* action in the controller where it's defined. You can use the `:only` and `:except` options to apply the `before_action` to specific actions within a controller. In the case of authorization, it's better to use `:except` to specify the actions for which the `before_action` should *not* run. That way, if you happen to define new actions, they're protected by default.

Answer:

```
class UsersController < ApplicationController
  before_action :require_signin, except: [:new, :create]

  # existing actions
end
```

2. Next, define the `require_signin` method so that it redirects to the sign-in form if a user isn't currently signed in.

   Where should the `require_signin` method be defined? Well, the immediate need is to be able to call that method from the `UsersController`. But later on we'll need to restrict access to actions in other controllers as well. So to make the `require_signin` method accessible to all controllers, you'll need to define it in `ApplicationController` (in `app/controllers/application_controller.rb`).

   All Rails controllers inherit from `ApplicationController` which means any methods you define there are automatically accessible in all the other controllers. And since the `require_signin` method is only going to be called by `before_action` within a controller, it's a good practice to make it a `private` method so it's not exposed as a controller action.

   For now, assume you can call the existing `current_user` method to determine if a user is signed in.

   Hint:

   The `require_signin` method simply needs to check the result of calling the existing `current_user` method. If it returns `nil` (a false value in Ruby) indicating that a user isn't currently signed in, then redirect to the sign-in form with a flash alert message.

   Answer:

```
class ApplicationController < ActionController::Base

private

  def require_signin
    unless current_user
      redirect_to new_session_url, alert: "Please sign in first!"
    end
  end

end
```

3. There's one small problem: The `require_signin` method calls the `current_user` method which is currently a view helper method defined in the `ApplicationHelper`. And that won't work because helper methods can't be called from inside controllers!

   To fix that, move the `current_user` method from the `ApplicationHelper` module to the `ApplicationController` class as another private method. That makes the method available to controllers, **but** it can no longer be called as a helper method in views. So you'll also need to declare `current_user` as a helper method so that it's available in all views.

   Hint:

   To declare a method in a controller as also being a helper method, call the `helper_method` method and pass it the name of the method (as a symbol) that you want to make accessible as a helper method.

   Answer:

```
class ApplicationController < ActionController::Base

private

  def require_signin
    unless current_user
      redirect_to new_session_url, alert: "Please sign in first!"
    end
  end

  def current_user
    @current_user ||= User.find(session[:user_id]) if session[:user_id]
  end

  helper_method :current_user
end
```

4. Finally, to check your work, sign out and then try to list the users and view a particular user's profile page. Regardless of what you do, the `before_action` should intercept your request and redirect you to the sign-in form. Because the `before_action` caused a redirect, the original action (`index` or `show` in this case) never got run. And that's exactly as it should be!

Solution

The full solution for this exercise is in the `authorization-1` directory of the code bundle.

Wrap Up

That's a great start! Now that you have some basic authorization working, we'll take things up a notch in the next exercise by further restricting access based on *who* is signed in.

Authorization: Part 2

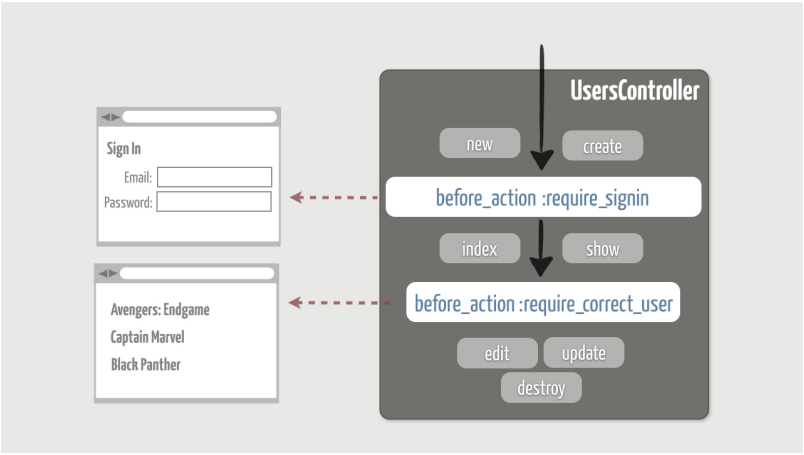
Exercises

Objective

Requiring users to be signed in puts us on the right path, but it's not enough. We still have a gaping security hole: any signed-in user can edit or delete any other user's account! Obviously, users should only be allowed to edit or delete their *own* account.

To enforce that, we'll define a `require_correct_user` method to check whether the signed-in user is the same as the user being edited or deleted. And we'll declare a second `before_action` to run the `require_correct_user` method before the `edit`, `update`, and `destroy` actions. We'll also hide the "Edit Account" and "Delete Account" links if you're not authorized to perform those actions.

Here's a high-level visual of our goal, adding a second level of authorization:



1. Ensure the Correct User Is Signed In

First up, a signed-in user should only be allowed to edit or delete their *own* account.

1. In the `UsersController` beneath the existing `before_action`, add another `before_action` that calls a method named `require_correct_user` before running the `edit`, `update`, or `destroy` actions.

   Hint:

   In this situation, it's best to use the `only:` option so that the `before_action` is only run before those three specific actions.

   Answer:

   ```
   class UsersController < ApplicationController
     before_action :require_signin, except: [:new, :create]
     before_action :require_correct_user, only: [:edit, :update, :destroy]

     # existing actions
   end
   ```

   The order in which you declare `before_action` methods is crucial. Before an action is run, the `before_action` methods are executed in turn starting at the top and working down. If at any point along the way one of the `before_action` methods causes a redirect (or returns `false`), then execution terminates. So by putting the `require_correct_user` check *after* the `require_signin` check, we're assured that the user is signed in before checking that they're the correct user.

2. Then define the `require_correct_user` method. It will only be called by the `UsersController`, so define the method in the `private` section of that class. The `require_correct_user` method needs to find the user being accessed and compare it to the currently signed-in user (the result of calling the `current_user` method). If the user being accessed is not the signed-in user, then deny access by redirecting to the application home page.

   Hint:

   Use the == operator to compare the result of calling the `current_user` method with the value in the `@user` instance variable.

   Answer:

   ```
   class UsersController < ApplicationController
     before_action :require_signin, except: [:new, :create]
     before_action :require_correct_user, only: [:edit, :update, :destroy]

     # existing actions

     private

     def require_correct_user
       @user = User.find(params[:id])
       unless current_user == @user
         redirect_to root_url, status: :see_other
       end
     end
   end
   ```

3. Now you're in a position to remove a small bit of duplication. Remember that `require_correct_user` will be called *before* the `edit`, `update`, and `destroy` actions. And if you peek at those actions, you'll notice that the first thing they do is find the user being accessed and assign it to an `@user` instance variable. You can safely remove those lines since the `require_correct_user` method has already set up an `@user` instance variable.

4. Check your work by signing in, listing the users, and showing the profile page for a user you aren't signed in as. Then try clicking the "Edit Account" link. Since it's not your account, you should get redirected to the home page.

   What about deleting someone else's account? You should get turned away just the same.

5. Finally, make sure you can in fact edit your own account!

2. Hide Unauthorized Links

We've successfully restricted access to the `edit` and `destroy` actions, but the "Edit Account" and "Delete Account" links are still being displayed on every user's profile page. Let's tidy that up. We only want to display those links if you're viewing your own profile page. In other words, you need to be the correct user to even *see* those links!

1. Start by wrapping the "Edit Account" and "Delete Account" links on the profile page with a simple conditional. Notice that the `show.html.erb` template already has an `@user` instance variable referencing the user being shown. So you only want to generate the links if that user is the same as the currently signed-in user (the result of calling the `current_user` method).

   Answer:

   ```
   <% if current_user == @user %>
     <div class="actions">
       <%= link_to "Edit Account", ... %>
       <%= link_to "Delete Account", ... %>
     </div>
   <% end %>
   ```

2. As a quick sanity check, go to a profile page for a user you aren't signed in as. You shouldn't see the "Edit Account" and "Delete Account" links. However, the links should still be displayed on your own profile page.

3. OK, that works, but it presents an opportunity to refactor. You now have the following comparison happening in both the `show` template and the `require_correct_user` method:

   ```
   current_user == @user
   ```

   What we're essentially trying to do here is determine whether the `@user` is the currently signed-in user. We can make that more expressive, and eliminate possible duplication by encapsulating that comparison in a method called `current_user?`. Here's an example of how we want to call that method:

   ```
   current_user?(@user)
   ```

   Remember, methods in Ruby that end with a question mark (?) generally return a boolean result. Calling them is like asking a yes or no question. In this case, we're asking the `current_user?` method whether the given `@user` is the current user. And we can use the result to make a decision in a conditional expression. It's a trivial one-line method, but it expresses our intent better.

4. With that in mind, go ahead and define the `current_user?` method. You want to be able to call it from any controller and also as a helper method from any view. So it needs to be defined in the `ApplicationController`.

   Hint:

   The question mark (?) is part of the method name. Also, don't forget to declare it as a helper method using `helper_method`.

   Answer:

   ```
   class ApplicationController < ActionController::Base

   private
     # existing methods

     def current_user?(user)
       current_user == user
     end

     helper_method :current_user?
   end
   ```

5. Then change the user `show` template to call your new `current_user?` method.

   Answer:

   ```
   <% if current_user?(@user) %>
     <div class="actions">
       <%= link_to "Edit Account", ... %>
       <%= link_to "Delete Account", ... %>
     </div>
   <% end %>
   ```

6. As well, you can now use the `current_user?` method over in the `require_correct_user` method of the `UsersController`. For style points, put the redirection and conditional all on one line.

   Answer:

   ```
   def require_correct_user
     @user = User.find(params[:id])
     redirect_to root_url, status: :see_other unless current_user?(@user)
   end
   ```

7. Finally, make sure everything still works!

3. Go Back To The Intended URL

Ready for a finishing move? Here's a fun one that gives you more practice with the session...

You probably noticed that if you're not signed in and try to access a restricted page, after signing in you don't end up on the intended page. Instead, after successfully signing in, you *always* get redirected to your profile page. It's as if the application forgot where you intended to go in the first place. A smarter application would remember your intended destination and then immediately redirect you back to that URL after you've signed in.

And now that we understand sessions, we can do that...

1. To demonstrate the problem, first sign out and then try to visit the user listing page. We've set it up to only be viewable by signed-in users, so you'll get redirected to the sign-in form. Go ahead and take the bait by signing in. You should then get redirected to your profile page. But wait, the whole point of signing in was to see that user listing page!

2. First, you need to store the URL of the requested page in the session *before* redirecting to the sign-in form. Remember, that redirection happens in the `require_signin` method of the `ApplicationController`. You can get the URL of the requested page using `request.url`. Simply assign it to the `:intended_url` session key (the name of the key is arbitrary).

   Answer:

   ```
   def require_signin
     unless current_user
       session[:intended_url] = request.url
       redirect_to new_session_url, alert: "Please sign in first!"
     end
   end
   ```

3. Then, when a user has successfully signed in, redirect them back to the URL stored in the session if one exists. If there isn't a URL to return to, default to redirecting to the user's profile page as before. Remember that after redirecting you'll need to remove the URL from the session so it's not stuck in there.

   Hint:

   Use the Ruby `||` operator to redirect to the URL in `session[:intended_url]` if it's not `nil`. Otherwise, redirect to the user's profile page.

   Answer:

   ```
   class SessionsController < ApplicationController
     def create
       user = User.find_by(email: params[:email])
       if user && user.authenticate(params[:password])
         session[:user_id] = user.id
         redirect_to (session[:intended_url] || user),
                     notice: "Welcome back, #{user.name}!"
         session[:intended_url] = nil
       else
         flash.now[:alert] = "Invalid email/password combination!"
         render :new, status: :unprocessable_entity
       end
     end
   end
   ```

4. Finally, give it a whirl by signing out and going to a page that requires sign in. After signing in, you should end up on the page you were aiming for.

It's just a little icing on the cake that makes life easier for our users!

Solution

The full solution for this exercise is in the `authorization-2` directory of the code bundle.

Wrap Up

You now have "gatekeepers" in place for all the restricted user actions. This means your app is more secure when it comes to only allowing certain users to do certain things.

Can you guess where else we could use a gatekeeper in the app? Hint: What might a malicious user do if he doesn't like a particular movie?

Dive Deeper

To learn more about filters such as `before_action`, refer to *8 Filters* in the Rails Guide: Action Controller Overview.

Admin Users

Exercises

Objective

In the previous exercise we focused on authorization to restrict access to the actions in the `UsersController`. Using the same technique, we can layer in authorization in other areas of our app. In this exercise we'll apply what we learned toward restricting access to another resource. And this time we'll throw in a twist...

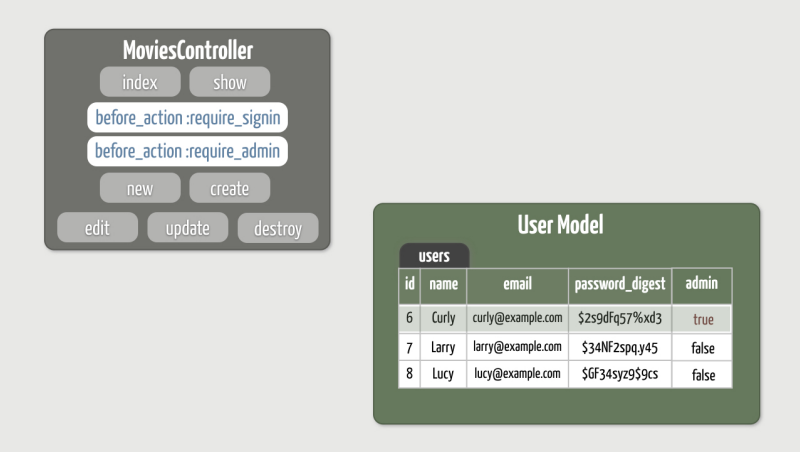
Way, way back in the course we designed the interface for creating, editing, and even destroying movies. At the time we weren't concerned with *who* could perform those actions. So as it stands, that interface is wide open to anyone who happens to stumble into our app. Obviously that's just asking for trouble. We at least need to restrict that part of the interface to signed-in users only. And we already know how to do that.

But what type of user is allowed to create, edit, and destroy movies? To make things interesting, let's say that only special "admin" users are allowed to perform those actions. Think of admins as the folks with the insider Hollywood connections. They're our trusted authority when it comes to movie information.

So here's what we need to do:

- Generate a new migration file that adds an `admin` boolean field to the `users` database table
- Turn an existing user into an admin user (with that special Hollywood glow!)
- Use a `before_action` to ensure only admin users can create, edit, and destroy movies
- Hide all the links to admin functions unless an admin user is signed in

Here's a quick visual of our goal:



Most of this is just a slight variation of what we've already done, so let's break a leg...

1. Add an Admin Column to Users

Granting special access to admin users hinges on the ability to identify admin users apart from regular users. To do that, we'll add a boolean `admin` attribute to the `User` model. Then we'll flag a user as being an admin by setting their `admin` attribute to `true`.

1. Use the migration generator to generate a migration named `AddAdminToUsers` that adds an `admin` column of type `boolean` to the `users` table.

   Answer:

   ```
   rails g migration AddAdminToUsers admin:boolean
   ```

2. Next we need to make sure that new users will not be flagged as admins by default. To enforce that, update the `change` method in the generated migration file so that the `admin` column defaults to a value of `false`.

   Answer:

   ```
   add_column :users, :admin, :boolean, default: false
   ```

   Technically this isn't necessary because boolean columns automatically default to `nil` which is considered false in Ruby. However, setting the default to `false` is more explicit and clear.

3. Finally, apply the new migration.

   Answer:

   ```
   rails db:migrate
   ```

2. Turn a Regular User Into an Admin User

Now we're ready to transform a regular user into an all-powerful admin user. We'll do that in the console...

1. First, find an existing user you want to endow with admin privileges.

   Answer:

   ```
   >> user = User.find_by(name: "Daisy")
   ```

2. Then turn that user into an admin user.

   Hint:

   Remember, adding a database column to the `users` table automatically gives you an `admin` attribute in the `User` model. To flag a user as an admin, simply assign `true` to the `admin` attribute.

   Answer:

   ```
   >> user.admin = true
   ```

3. Now check that the user is indeed an admin user.

   Hint:

You can either read the value of the `admin` attribute, or use the more expressive `admin?` method to test the admin status. Either way, the result should be `true`.

Answer:

```
>> user.admin?
=> true
```

4. Finally, don't forget to save the admin user before moving on!

   Answer:

   ```
   >> user.save
   ```

3. Only Allow Admins to Create, Edit, and Delete Movies

Next up, we need to ensure that only signed-in *admin* users can create, edit, and delete movies. Thankfully, we already have some experience restricting access to controller actions.

1. Starting in the `MoviesController`, first use a `before_action` to require a signed-in user before running any action except `index` or `show`.

   Hint:

   Call the `require_signin` method you defined in the previous exercise and use the `except` option.

   Answer:

   ```
   class MoviesController < ApplicationController
     before_action :require_signin, except: [:index, :show]
   end
   ```

2. Then add a second `before_action` that calls a yet-to-be-defined `require_admin` method before running any action except `index` or `show`.

   Answer:

   ```
   class MoviesController < ApplicationController
     before_action :require_signin, except: [:index, :show]
     before_action :require_admin, except: [:index, :show]
   end
   ```

3. Now define the `require_admin` method. It needs to check that user is currently signed in and has admin status. If that's not the case, deny access by redirecting to the application home page with a flash alert message.

   To neatly encapsulate the code that checks whether the current user is an admin, go ahead and define another method named `current_user_admin?`. It should return `true` if there's a current user **and** that user is an admin. Then call the `current_user_admin?` method from the `require_admin` method.

   Both of these methods should be accessible to all controllers, so put them in the `ApplicationController`.

   Answer:

   ```
   def require_admin
     unless current_user_admin?
       redirect_to root_url, alert: "Unauthorized access!"
     end
   end

   def current_user_admin?
     current_user && current_user.admin?
   end
   ```

4. Check your work by signing in as a non-admin user and clicking the "Add New Movie" link in the sidebar. Since you're not an administrator, you should get bounced over to the home page. The same should hold true if you try to edit or delete a movie.

5. Then sign out and sign in as a bonafide admin user. Now you should be able to create a new movie, as well as edit and delete an existing movie. How's that for rolling out the red carpet?

4. Hide Unauthorized Links

Finally, only admin users should see the "Edit" and "Delete" links on the movie show page and the "Add New Movie" link on the movie index page. No sense tempting everyone else.

1. We'll obviously need to wrap the links in a conditional. The conditional needs to check whether the current user is also an admin user. And lo and behold, we've already encapsulated that logic in the `current_user_admin?` method. But to call it from a view, you'll first need to declare the method as a helper. Remember how to do that?

   Answer:

   ```
   helper_method :current_user_admin?
   ```

2. Then over on the movie show page, use that helper method to only generate the "Edit" and "Delete" links if the current user is an admin.

   Answer:

   ```
   <% if current_user_admin? %>
     <section class="admin">
       <%= link_to "Edit", ... %>
       <%= link_to "Delete", ... %>
     </section>
   <% end %>
   ```

3. In the same way, hide the "Add New Movie" link on the movie index page unless the current user is an admin.

   Answer:

```
<% if current_user_admin? %>
  <section class="admin">
    <%= link_to "Add New Movie", new_movie_path, class: "button" %>
  </section>
<% end %>
```

4. To visually check your work, first make sure you're not signed in as an admin. Then go to a movie show page and you shouldn't see the "Edit" and "Delete" links. You also shouldn't see the "Add New Movie" link at the bottom of the movie index page.

Solution

The full solution for this exercise is in the `admin-users` directory of the code bundle.

Bonus Round

Only Allow Admins to Delete Accounts

We now have three `before_action` methods that give us fairly fine-grained control when it comes to authorization: `require_signin`, `require_correct_user`, `require_admin`. Depending on the situation, you can mix and match these methods (or define new ones) to dial in the proper amount of control.

As an example, suppose that you only wanted to allow admins to delete user accounts. That is, a user isn't allowed to delete his or her own account. How might you re-arrange the code to support that?

1. Restrict access with `before_action` methods.

   Hint:

   In the `UsersController`, change the `require_correct_user` method to only run for the `edit` and `update` actions. In other words, remove the `destroy` action from the `only` option. Then add a third `before_action` that runs the `require_admin` method before the `destroy` action.

   Answer:

```
class UsersController < ApplicationController
  before_action :require_signin, except: [:new, :create]
  before_action :require_correct_user, only: [:edit, :update]
  before_action :require_admin, only: [:destroy]

  ...
end
```

2. You'll also need to change the `destroy` action to find the user to destroy since the `require_correct_user` filter is no longer being run for this action. Also, after destroying the user you don't want to automatically sign out the current user. Otherwise, the admin user will get signed out when they delete someone's account!

   Answer:

```
def destroy
  @user = User.find(params[:id])
  @user.destroy
  redirect_to root_url, status: :see_other,
    alert: "Account successfully deleted!"
end
```

3. On the user profile page, only show the "Delete Account" link if the current user is an admin.

   Answer:

```
<div class="actions">
  <% if current_user?(@user) %>
    <%= link_to "Edit Account", ... %>
  <% end %>
  <% if current_user_admin? %>
    <%= link_to "Delete Account", ... %>
  <% end %>
</div>
```

Wrap Up

You now have a built-from-scratch authorization and authentication system in place that:

- Uses `before_action` methods to intercept calls to controller actions and determine if, based on that particular user, an action can indeed be called.
- Defines commonly-used code in the `ApplicationController` to avoid duplication.
- Adds the user's intended destination to the session and redirects the user back to that URL after she has signed in.
- Distinguishes admin users from regular users by way of a boolean `admin` attribute in the `User` model, and grants admin users special access.
- Displays only certain links (such as "Edit Account" and "Edit Movie") to authorized users.

Nicely done! Now might be the perfect time to enjoy a well-deserved break. ☕ 🍷 🍩 🍕

Coming up next, since we have users and basic authorization in place, we'll use it to streamline the movie review process. If you're already signed in, there's no need to ask for your name when you write a review. Furthermore, in the database we'll associate your user account with the reviews you've written!

Dive Deeper

If your application has more complex authorization requirements, you might consider using a gem such as CanCan or Pundit. They give you a flexible way to assign specific permissions to users and check those permissions during authorization. They rely on the existence of a `current_user` method, which you could now write in your sleep. Indeed, now that you've created a simple authentication and authorization system from scratch, you have a better understanding of how it all works!

Many-to-Many Associations: Part 1

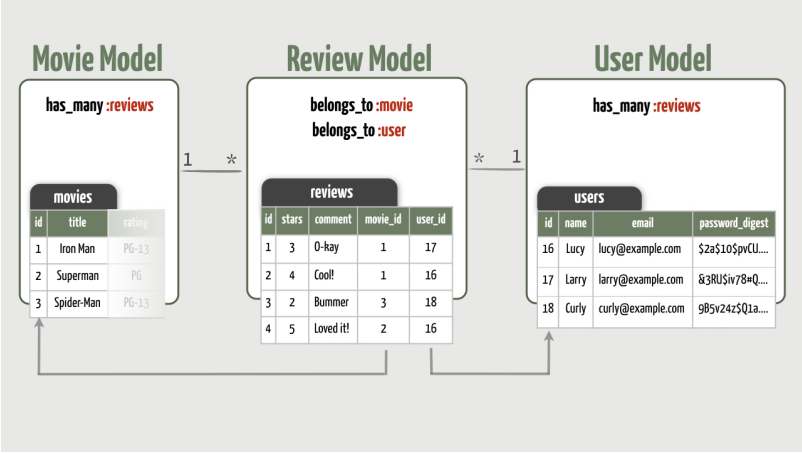
Exercises

Objective

So let's take stock of where our app stands. We added support for user accounts and we have an effective authentication solution allowing users to sign up, sign in, and sign out. We also put in place a simple authorization system that only allows certain users to do certain things.

Building off that, we now want to arrange things so that only signed-in users can write a movie review. Given that restriction, we'll always know who's writing the review, so we can remove the name field from the review form. Then when a review is created, we'll associate it with the user who reviewed the movie.

To do that, we'll first need to create a one-to-many association between users and reviews. The `Review` model will become a *join model* that connects movies and users (reviewers). Visually, here's what the relationships will look like in the database:



Per Rails conventions, a join model has two foreign keys. In our `reviews` table, the `movie_id` key references a row in the `movies` table and the `user_id` key references a row in the `users` table.

To create this one-to-many association between users and reviews, we'll need to do the following:

- Change the `reviews` database table to have a `user_id` foreign key, and remove the `name` column.
- Remove the name validation from the `Review` model.
- Declare a one-to-many association between the `User` and `Review` models.

Of course, we'll also need to change the rest of the app to accommodate this new model association. But let's not get ahead of ourselves. Rather than tackling all this at once, we'll break it down into manageable tasks. We'll start at the model layer in this exercise, and then build upon it in the next exercise.

1. Change the Reviews Table

First we need to change the `reviews` table so that it joins a movie and a user. To do that, we just need to remove the `name` column and add a `user_id` foreign key column. (The `reviews` table doesn't have an `email` column like the `registrations` table did in the video.)

1. Start by generating a new (empty) migration file named `MakeReviewsAJoinTable`.

   Answer:

   ```
   rails g migration MakeReviewsAJoinTable
   ```

2. Then inside the migration file you'll find an empty `change` method. In the `change` method, first remove the `name` column from the `reviews` table using the handy `remove_column` method.

   Then add a `user_id` column of type `integer` to the `reviews` table using the `add_column` method.

   And since this schema change will cause all the existing reviews to be invalid, go ahead and delete all the existing reviews.

   Hint:

   The `remove_column` method takes three parameters: the name of the table, the name of the column to remove, and the column type. The `add_column` method also takes three parameters: the name of the table, the name of the column, and the column type. Remember that by convention, the name of a foreign key must be the name of the related entity (`user` in this case), followed by `_id`.

   Answer:

   ```
   class MakeReviewsAJoinTable < ActiveRecord::Migration[7.0]
     def change
   ```

```
      remove_column :reviews, :name, :string
      add_column :reviews, :user_id, :integer
      Review.delete_all
    end
end
```

3. Finally, apply the migration.

   Answer:

   ```
   rails db:migrate
   ```

It's worth pointing out that applying this migration causes us to potentially lose some data. By removing the `name` column, any names that were in that column are now gone forever. On top of that, we deleted all the reviews anyway since they aren't attached to users (their `user_id` column would have been `nil`). As we're in development mode and using example data, losing this data isn't a big deal.

However, running this migration file on the production database could be *disastrous* if it contained legitimate reviews you wanted to preserve. In that case, you might consider a fancier solution whereby you try to find the user with the given name and then programmatically (using Ruby code) set the `user_id` for existing reviews.

2. Declare the Model Relationships

Applying the migration gave us a `user_id` foreign key column in the `reviews` table, but we still need to tell Rails what kind of relationship we want between the `Review` and `User` models. In particular, a user *has many* reviews and a review *belongs to* a user.

1. Let's start with the `Review` model. It has an existing `belongs_to` association with the `Movie` model. Update the `Review` model to also have a `belongs_to` association with the `User` model.

   Answer:

   ```
   class Review < ApplicationRecord
     belongs_to :movie
     belongs_to :user

     ...
   end
   ```

2. While you're in that model, remove the validation for the name since it's no longer a review attribute. Leave the validations for `comment` and `stars` attributes intact, however.

   Answer:

   ```
   class Review < ApplicationRecord
     belongs_to :movie
     belongs_to :user

     validates :comment, length: { minimum: 4 }

     STARS = [1, 2, 3, 4, 5]

     validates :stars, inclusion: {
       in: STARS,
       message: "must be between 1 and 5"
     }

     # existing code
   end
   ```

3. Next, look in the `User` model and you'll notice that it doesn't have a reciprocal association to its reviews. Bi-directional relationship aren't required; you only need to define relationships in the direction you intend to use them. In our case, we certainly want to be able to find all of the associated reviews given a `User` object.

   To do that, update the `User` model to declare a `has_many` association with reviews (the plural form). Make sure that the association automatically destroys all of a user's reviews when the user itself is destroyed.

   Hint:

   Using `dependent: :destroy` indicates that the existence of `reviews` is dependent on the existence of the user. If we destroy the user (deleting it from the database) we also want any reviews associated with that user to be automatically destroyed.

   Answer:

   ```
   class User < ApplicationRecord
     has_many :reviews, dependent: :destroy

     # existing code
   end
   ```

   Remember, this declaration tells Rails to expect a `user_id` foreign key column in the table wrapped by the `Review` model, which by convention will be the `reviews` table. As we'll see, Rails also dynamically defines methods for accessing a user's reviews.

And with those two declarations, we now have a bi-directional association between a user and their reviews!

3. Experiment with the Association

Before we attempt to change the web interface, let's spend a minute in the `console` just making sure we can create a proper review that's attached to both a movie and a user.

1. First, initialize a new `Review` object in memory and go ahead and set the number of stars and a brief comment. (Don't worry about the movie or user just yet.) Assign the resulting review to a `review` variable.

Answer:

```
>> review = Review.new(stars: 5, comment: "Two thumbs up!")
```

2. Then ask the review for its associated movie.

   Hint:

   When you declared `belongs_to :movie` in the `Review` model, Rails dynamically defined a `movie` attribute in the `Review` model. Reading the `movie` attribute from a `Review` object returns the associated `Movie` object.

   Answer:

   ```
   >> review.movie
   ```

   You should get `nil` because a movie hasn't yet been attached to the review.

3. So attach a movie to the review by first finding a `Movie` object and then assigning it to the review's `movie` attribute.

   Answer:

   ```
   >> movie = Movie.find_by(title: "Captain Marvel")
   ```

   ```
   >> review.movie = movie
   ```

4. To verify that the movie and review are now linked together, ask the review for its associated movie.

   Answer:

   ```
   >> review.movie
   ```

   You should get back the `Movie` object you assigned to the review. That works because the `movie_id` foreign key was automatically set by virtue of using the `movie` object to initialize the `review`. So now when we read the review's `movie` attribute, it fetches the movie that has an `id` matching the `movie_id` value for this particular review.

   The review is now attached to the movie. So far, so good.

5. Now let's turn our attention to the one-to-many association between `Review` and `User` that we declared earlier in this exercise.

   Start by asking the review for its associated user.

   Hint:

   When you declared `belongs_to :user` in the `Review` model, Rails dynamically defined a `user` attribute in the `Review` model. Reading the `user` attribute from a `Review` object returns the associated `User` object.

   Answer:

   ```
   >> review.user
   ```

   You should get `nil` because a user hasn't yet been attached to the review.

6. So find and attach a user to the review.

   Answer:

   ```
   >> user = User.find_by(name: "Daisy")
   ```

   ```
   >> review.user = user
   ```

7. To verify that the review and user are now linked together, ask the review for its associated user again.

   Answer:

   ```
   >> review.user
   ```

   This time you should get back the `User` object you assigned to the review. When we assigned a `User` object to the `user` attribute, the user's primary key (`id`) was automatically assigned to the review's `user_id` foreign key. Now when we read the review's `user` attribute, it queries the database for the user that has an `id` matching the `user_id` value for this particular review.

   Excellent—now a review has everything it needs: a movie, a user, some stars, and a comment!

8. We haven't yet saved the review in the database, so go ahead and do that next.

   Answer:

   ```
   >> review.save
   ```

9. Now imagine you want to get all the reviews for this user. How would you do that?

   Hint:

   We declared that a user `has_many` reviews, so calling the `reviews` method on a `User` object returns an array of reviews that are associated with that user.

   Answer:

   ```
   >> user.reviews
   ```

   You should see the database being queried and get back an array that contains the one `Review` object that's associated with the user.

10. Just for practice, count the number of reviews for this user.

    Answer:

```
>> user.reviews.size
```

Cool! Now we have **two** one-to-many associations. A movie has many reviews and a user has many reviews. And sitting in the middle, a review belongs to one movie and one user. That effectively means we have a many-to-many relationship!

Solution

The full solution for this exercise is in the `many-to-many-1` directory of the code bundle.

Wrap Up

So now we have the associations working in the models. In the next section we'll look at what else we need to change in our app so that only signed in users can write a review and we no longer need to ask for the user's name on the review form.

This will actually be familiar territory. Before watching the next video, take a couple minutes to think through the MVC flow for reviews. Can you pinpoint some of the areas we'll need to change? As you become a more practiced Rails developer it will become increasingly important for you to be able to identify how changes in one part of your app warrant changes in other parts of your app.

Dive Deeper

To learn more about associations, refer to the Rails Guide: Active Record Associations. You might also want to review the documentation for the belongs_to and has_many methods.

Many-to-Many Associations: Part 2

Exercises

Objective

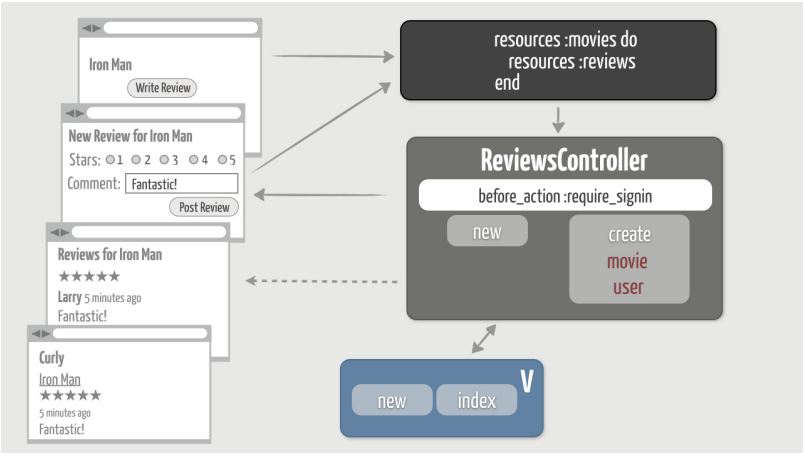
Now that we have the new review-user association wired up, we're ready for the next steps toward our overall objective:

- Only allow signed-in users to write a review
- Remove the name field from the review form since it's no longer required
- When a review is created, associate it with the signed-in user
- List a user's reviews on their profile page

To pull this off, we'll make changes across the familiar MVC triad:

- Use a `before_action` to require a user to sign in before writing a movie review.
- Change the review form to remove the unnecessary `name` field.
- Update the `create` action so that it associates the signed-in user with the review before it's saved.
- Fix the review index page so it no longer expects a review to have a `name` attribute.
- Spruce up the user profile page to list the reviews the user has written.

Visually, here's what we're looking for:



1. Require Signin to Write a Review

Currently, when someone writes a review using our web interface, the review is created but it's not associated with any particular user. In other words, we end up with a row in the `reviews` table with the `user_id` foreign key set to `nil`.

We need to arrange things so that every review gets tied to a particular user. And to do that, we need to require that a user is signed in before allowing them to write a review. Here's another opportunity to use that `require_signin` method we wrote earlier...

1. At the top of the `ReviewsController` class definition, use a `before_action` to run the `require_signin` method before every action.

Hint:

Remember, we can call the `require_signin` method from the `ReviewsController` because it inherits from `ApplicationController` where the `require_signin` method is defined.

Answer:

```
class ReviewsController < ApplicationController
  before_action :require_signin
  before_action :set_movie

  # existing code
end
```

2. Then sign out and try to review a movie. It should kick you back to the sign-in form. Sign in and you should end up back on the review form.

2. Update the Review Form

Since a user must now be signed in to write a review, we don't need to ask for their name on the review form. In fact, if you view the form now you'll get an error because the `Review` object no longer has a `name` attribute.

So go ahead and remove the `name` field from the form (`app/views/reviews/new.html.erb`), leaving the fields for the stars and comment.

Answer:

```
<%= form_with(model: [@movie, @review]) do |f| %>
  <%= render "shared/errors", object: @review %>

  <%= f.label :stars %>
  <%= f.select :stars, Review::STARS, prompt: "Pick one" %>

  <%= f.label :comment %>
  <%= f.text_area :comment, placeholder: "What did you think?" %>

  <%= f.submit "Post Review" %>
<% end %>
```

3. Change the Create Action

We know the review form will post to the `create` action in the `ReviewsController`. Currently that action creates a review that's attached to a movie, but the review isn't attached to a user yet. We need to fix that...

1. Change the `create` action in the `ReviewsController` so that it associates the signed-in user with the review before it's saved.

   Hint:

   Call the `current_user` method and assign the user it returns to the review's `user` attribute. Remember, `current_user` is defined in `ApplicationController`, so we can call it from the `ReviewsController`. And we know `current_user` will always return the signed-in user because we used a `before_action` to ensure that the `require_signin` method is always run before the `create` action.

   Answer:

   ```
   def create
     @review = @movie.reviews.new(review_params)
     @review.user = current_user
     if @review.save
       redirect_to movie_reviews_path(@movie), notice: "Thanks for your review!"
     else
       render :new, status: :unprocessable_entity
     end
   end
   ```

2. You'll also want to update the `review_params` method so that it no longer permits the `name` parameter, just the `comment` and `stars` parameters. Technically this isn't necessary since reviews no longer have a `name` attribute, but it's always good to remove unnecessary code.

   Answer:

   ```
   def review_params
     params.require(:review).permit(:comment, :stars)
   end
   ```

3. As a quick sanity check, reload the review form to make sure it no longer prompts for the name. But don't try to submit it just yet...

4. Change the Reviews Index Page

When a movie review is successfully created, the `create` action redirects to the page that lists all the reviews for that particular movie. It turns out that removing the `name` attribute from `Review` also affects this page. We need to make one small tweak to get it back in step with the code...

1. Start by looking in `app/views/reviews/index.html.erb` and you'll notice that it's displaying the reviewer's name like so:

   ```
   <%= review.name %>
   ```

   That will no longer work because we removed the `name` attribute from the `Review` model.

2. Change that line to instead display the name of the user who created the review.

   Hint:

Remember that the `belongs_to :user` line in the `Review` model lets us access the user who created the review by calling `review.user`. You can then get the reviewer's name by reading the user's `name` attribute.

Answer:

```
<%= review.user.name %>
```

3. For style points, instead of just displaying the reviewer's name, link their name to their user profile page.

   Answer:

   ```
   <%= link_to review.user.name, review.user %>
   ```

5. Write a Review Already!

Finally, with all these changes in place, it's time to give it a whirl...

1. First, make sure you're signed out.
2. Then go to a movie page.
3. Click "Write Review" and you should get redirected to the sign-in form.
4. Sign in and you should get redirected to the new review form. Perfect!
5. Write a review that reflects your true opinion of the movie and submit it.
6. You should end up seeing your review (with your linked name) in the list of reviews!

6. List Reviews on the User Profile Page

That leaves us with one final task...

Click on a reviewer's name to go to that user's profile page, and you'll notice that the profile page is still fairly sparse. The lights are on, but nobody appears to be home. To liven things up a bit, let's show all the reviews this user has written.

1. First you need to access all of the user's reviews. Remember that it's a good practice to have a controller action set up the data for its view. So first identify the action that renders the profile page and then in that action define an `@reviews` instance variable that references the user's reviews.

   Hint:

   The profile page is rendered by the `show` action in the `UsersController`. The `has_many :reviews` line in the `User` model lets us access the user's reviews by calling `@user.reviews`. Assign the resulting array to a `@reviews` instance variable.

   Answer:

   ```
   def show
     @user = User.find(params[:id])
     @reviews = @user.reviews
   end
   ```

2. Now that you have the review data, update the `show` view template to display the user's reviews if any are present. For each review, display the following:

   - title of the reviewed movie, linked back to the movie page
   - number of stars (use the `shared/stars` partial)
   - how long ago the review was created
   - the review comment

   Don't worry about the HTML structure and styling. Once you get all the data displayed you can peek at the answer to structure and style the reviews according to our CSS rules (which is always totally optional).

   Answer:

   ```
   <section class="user">
     # existing code

     <% if @reviews.present? %>
       <h3>Reviews</h3>
       <% @reviews.each do |review| %>
         <div class="review">
           <div class="details">
             <span class="title">
               <%= link_to review.movie.title, review.movie %>
             </span>
             <%= pluralize(review.stars, 'star') %>
             <span class="date">
               <%= time_ago_in_words(review.created_at) %> ago
             </span>
             <p>
               <%= review.comment %>
             </p>
           </div>
         </div>
       <% end %>
     <% end %>
   </section>
   ```

3. Finally, reload the user's profile page and you should see some recent activity!

Solution

The full solution for this exercise is in the `many-to-many-2` directory of the code bundle.

Bonus Round

Show Each Reviewer's Profile Image

Want to give the list of reviewers a bit more personality? Consider displaying a small version of each reviewer's profile image next to their name in the list. And if you added a `username` to each user in a previous bonus exercise, consider showing that as well.

To do that, you'll need to modify the `profile_image` helper you created in a previous bonus exercise to allow for a size to be specified. Here's one way to do that with `size` parameter that defaults to a size of 80:

```
def profile_image(user, size=80)
  url = "https://secure.gravatar.com/avatar/#{user.gravatar_id}?s=#{size}"
  image_tag(url, alt: user.name)
end
```

Then you can call that helper to display a small version of each review's profile image, say 35 pixels.

Answer:

```
<%= profile_image(review.user, 35) %>
```

Wrap Up

Way to hang in there! We accomplished a *lot* over the last couple exercises. Even though the concept of *one* movie having *many* reviews isn't necessarily complicated, there can be quite a bit of mental overhead involved when it comes to actually implementing this association in the user interface. You have to keep not one, but *two* objects in your head as you think through the parts of MVC that are affected.

In the next section, we'll use another join model to create yet another many-to-many association. This go-around we'll allow users to "fave" movies, which will give us practice creating a many-to-many association from scratch...

Another Many-to-Many Association

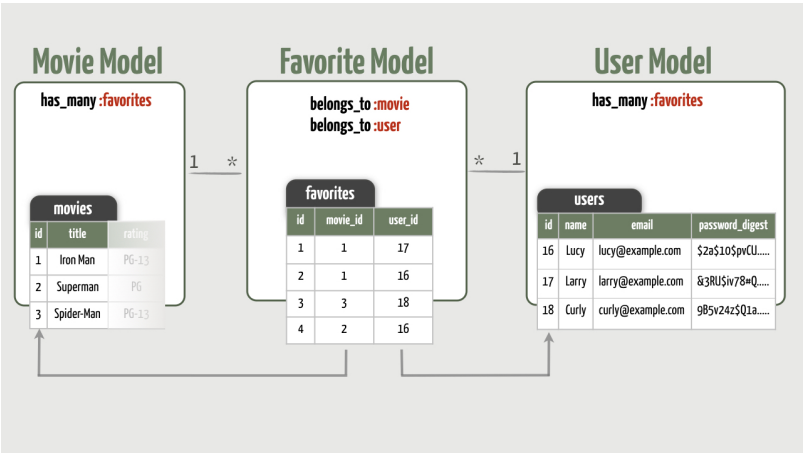
Exercises

Objective

Got any favorite movies? We want to let signed-in users "fave" their favorite movies. A user can have many favorite movies. And a movie could be a favorite of many users.

To do this, we'll need a many-to-many association between movies and users. Since many-to-many associations can't be modeled with just two database tables, we'll create a brand new join table named `favorites` between them. Then we'll declare a many-to-many association between the `Movie` and `User` models that's joined through the `Favorite` model. This will give us practice creating a many-to-many association from scratch.

Here's a snapshot of the models and tables we want:



1. Generate the Favorites Resource

First we need a new resource to represent a favorite, which simply joins a movie and a user. We'll need a `favorites` database table, a `Favorite` model, a `FavoritesController`, and the conventional resource routes. To make quick work of this, we'll use the resource generator to churn out all the files we need.

1. Start by generating a resource named `favorite` that references both a `movie` and a `user`.

   Answer:

   ```
   rails g resource favorite movie:references user:references
   ```

2. Crack open the generated migration file and you should see the following:

```
class CreateFavorites < ActiveRecord::Migration[7.0]
  def change
    create_table :favorites do |t|
      t.references :movie, null: false, foreign_key: true
      t.references :user, null: false, foreign_key: true

      t.timestamps
    end
  end
end
```

Remember that whenever you see `references` in a migration file, it's simply a shortcut for adding a foreign key column. For example, the line `t.references :movie` adds a column called `movie_id` and `t.references :user` adds a column called `user_id`. The generator also tacks on the `null: false` and `foreign_key: true` options to both columns so null values aren't allowed and appropriate foreign key constraints are added.

So this migration will end up creating a join table with two foreign keys, which is exactly what we want.

3. Finish up as always by applying the migration, which in this case creates the `favorites` database table.

Answer:

```
rails db:migrate
```

2. Declare the Model Associations

Applying the migration created the `favorites` database table with `movie_id` and `user_id` foreign key columns. So far, so good. The next step is to declare the specific associations we want in the models. Coming back to our objective, we want a many-to-many association with the following direct model associations:

- A favorite belongs to both a movie and a user
- A movie has many favorites
- A user also has many favorites

Let's start with the `Favorite` model...

1. Open the generated `favorite.rb` file and you should see that it's ready to go:

```
class Favorite < ApplicationRecord
  belongs_to :movie
  belongs_to :user
end
```

Fresh from the generator, a `Favorite` already `belongs_to` both a movie and a user. Those two declarations were generated because we used `movie:references` and `user:references` when generating the `favorite` resource. One model down, two to go.

2. Next, look in the `Movie` model and you'll discover that it doesn't yet have an association with its favorites. So update the `Movie` model to declare a `has_many` association with `favorites`. If a movie is destroyed, make sure all of its favorites are also automatically destroyed.

Hint:

Remember that setting the `dependent` option to `destroy` will cause the `destroy` method to be called on each associated favorite if/when the movie itself is destroyed.

Answer:

```
class Movie < ApplicationRecord
  has_many :reviews, dependent: :destroy
  has_many :favorites, dependent: :destroy

  # existing code
end
```

Remember, this declaration tells Rails to expect a `movie_id` foreign key column in the table wrapped by the `Favorite` model (the `favorites` table).

Two models down, one to go.

3. Now, over in the `User` model, we want to declare the reciprocal association. Declare a `has_many` association with `favorites`. If a user is destroyed, all of its favorites should also be destroyed.

Answer:

```
class User < ApplicationRecord
  has_many :reviews, dependent: :destroy
  has_many :favorites, dependent: :destroy

  # existing code
end
```

This declaration tells Rails to expect a `user_id` foreign key column in the `favorites` table.

We now have a bi-directional, many-to-many association between a movie and the users who fave it.

3. Experiment with the Associations

Now let's jump into a `console` session and create some favorites, just to get a feel for how these associations work...

1. First, find an existing `Movie` and assign it to a `movie` variable.

Answer:

```
>> movie = Movie.find_by(title: "Black Panther")
```

2. Then find an existing `User` and assign it to a `user` variable.

   Answer:

   ```
   >> user = User.find_by(name: "Daisy")
   ```

3. Then initialize a new `Favorite` object and associate it with the movie and user.

   Hint:

   Create the favorite by calling `Favorite.new`. Associate the movie to the favorite by assigning the movie to the favorite's `movie` attribute. And associate the user to the favorite by assigning the user to the favorite's `user` attribute.

   Answer:

   ```
   >> favorite = Favorite.new
   >> favorite.movie = movie
   >> favorite.user = user
   ```

   Then if you print the `Favorite` object, the `movie_id` and `user_id` foreign keys should be set to reference the respective `movie` and `user` objects.

4. To verify that the movie and favorite are now linked together, ask the favorite for its associated movie.

   Answer:

   ```
   >> favorite.movie
   ```

   You should get back the `Movie` object you assigned to the favorite.

5. Also verify that the favorite and user are linked together by asking the favorite for its associated user.

   Answer:

   ```
   >> favorite.user
   ```

   You should get back the `User` object you assigned to the favorite.

6. We haven't yet saved the favorite in the database, so go ahead and do that next.

   Answer:

   ```
   >> favorite.save
   ```

   You should see a SQL `INSERT` statement get executed to create a new row in the `favorites` database table. Notice that the `movie_id` and `user_id` foreign keys are set to reference the respective rows in the `movies` and `users` tables. So any particular row in the `favorites` table effectively joins a movie and a user.

7. Now, just for practice, make this same movie a favorite of another user. But instead of initializing a new favorite by calling `Favorite.new`, this time use the `movie.favorites` association to initialize the new favorite so that it's automatically associated with the same `movie` object. Then associate the resulting `Favorite` object with another user.

   Answer:

   ```
   >> favorite = movie.favorites.new
   >> user = User.find_by(name: "Larry")
   >> favorite.user = user
   ```

8. Then verify that the favorite belongs to both the movie and the user.

   Answer:

   ```
   >> favorite.movie
   >> favorite.user
   ```

9. Don't forget to save it in the database.

   Answer:

   ```
   >> favorite.save
   ```

10. At this point, the movie should have two favorites. How would you verify that?

    Hint:

    We declared that a movie `has_many` favorites, so calling the `favorites` method on a `Movie` object returns an array of favorites that are associated with that movie. And you can check the number of favorites by tacking on a call to the `size` method.

    Answer:

    ```
    >> movie.favorites
    >> movie.favorites.size
    ```

11. Finally, create one more favorite for a different movie and the same user, this time associating the movie and user and saving the favorite in one step.

    Hint:

Create the favorite by calling `movie.favorites.create!` and passing it a hash with the key `user` whose value is the `user` object. Using the `movie.favorites` association automatically sets the favorite's `movie_id` foreign key to point to the `movie` object. Passing in the `user` object automatically sets the favorite's `user_id` foreign key to point to the `user` object. The `create!` method will raise an exception if for some reason the favorite couldn't be created.

Answer:

```
>> movie = Movie.find_by(title: "Avengers: Endgame")
>> movie.favorites.create!(user: user)
```

You should see a SQL `INSERT` statement get immediately executed to create a new `favorites` row in the database with the `movie_id` and `user_id` foreign keys automatically set to reference the respective `movie` and `user` objects.

12. At this point, the user should have two favorites for different movies. How would you verify that?

    Hint:

    We declared that a user `has_many` favorites, so calling the `favorites` method on a `User` object returns an array of favorites that are associated with that user. And again, you can count the number of favorites by chaining a call to the `size` method.

    Answer:

```
>> user.favorites
>> user.favorites.size
```

Excellent! That gives us confidence that our models are in good shape.

Solution

The full solution for this exercise is in the `another-many-to-many` directory of the code bundle.

Wrap Up

So this gave us more practice with many-to-many associations. But often when you have these types of associations, the join model gets in the way. For example, we can ask a user for her favorites, but what if we want to get the actual movies that she considers her favorites? Likewise, what if we want to get all the users who have favorited a particular movie? Well, as things stand right now, to do that we'd have to write custom code that traverses across three models: `Movie`, `Favorite`, and `User`. And doing that is both tedious and inefficient.

Thankfully, Rails offers another type of association—the `through` association—that offers a convenient and efficient way to traverse across the models. We'll explore `through` associations in the next section...

Through Associations: Part 1

Exercises

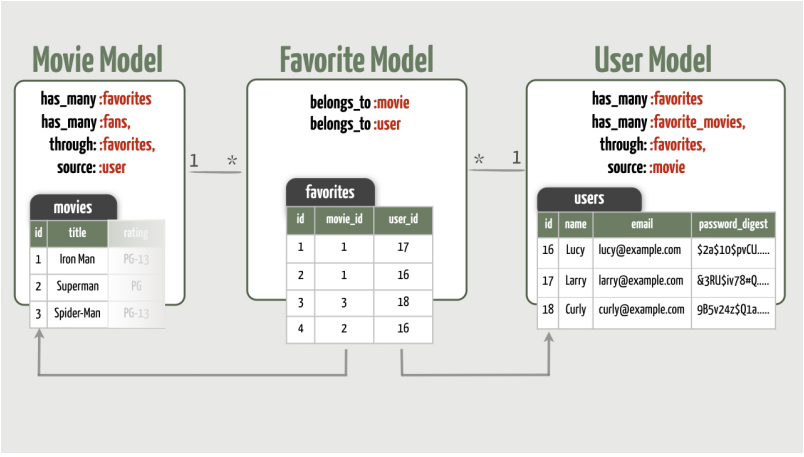
Objective

Now we want to take the many-to-many association a step further by creating a `through` association so we can navigate directly from a user to her favorite movies, and vice versa. Like many things in Rails, these associations can feel magical at first. So give yourself permission to take your time through this exercise so you can fully appreciate and understand what's going on.

Here are our objectives:

- Declare through associations between movies and users (which we'll call "fans")
- Display a movie's fans on the movie's show page
- Display a user's favorite movies on the user's profile page

Here's an updated snapshot of the associations we want:



1. Declare the Through Associations

We currently have the following direct associations between the `Favorite`, `Movie`, and `User` models:

- A favorite belongs to both a movie and a user
- A movie has many favorites
- A user also has many favorites

We also want the following indirect associations going *through* favorites:

- A movie has many fans (users)
- A user has many favorite movies

1. Start by looking at the `Movie` file and you'll see it currently has many favorites:

   ```
   class Movie < ApplicationRecord
     has_many :reviews, dependent: :destroy
     has_many :favorites, dependent: :destroy

     # existing code
   end
   ```

   Given this `has_many` declaration, we can ask a movie for its favorites using `movie.favorites`. That returns an array of `Favorite` objects. Then if we wanted to get the users who favorited the movie, we'd need to iterate through each of the `Favorite` objects and fetch the associated user. But that's the long way around the barn (it takes 1+n database queries). Instead of hopping from a movie to its favorites and then to the associated users, we want to go directly from a movie to its users, which we'll call "fans".

   To do that, in the `Movie` model create another `has_many` association named `fans` that goes *through* the existing `favorites` association.

   Hint:

   The `through` option takes the name of the association through which to perform the query, which is `favorites` in this case. It must be declared *after* the `has_many :favorites` association. Since Rails can't infer that a fan is actually a user (how could it?), you also need to use the `source` option to specify that the source is `:user`. Rails then knows to use the `belongs_to :user` association in the `Favorite` model to query the users.

   Answer:

   ```
   class Movie < ApplicationRecord
     has_many :reviews, dependent: :destroy
     has_many :favorites, dependent: :destroy
     has_many :fans, through: :favorites, source: :user

     # existing code
   end
   ```

2. Now, look in the `User` model and you'll recall that it currently also has many favorites:

   ```
   class User < ApplicationRecord
     has_many :reviews, dependent: :destroy
     has_many :favorites, dependent: :destroy

     # existing code
   end
   ```

   Given a user, we can ask for its favorites by calling `user.favorites`. But we'd also like to be able to ask for the user's favorite movies, and fetch them from the database in one query.

   To support that, in the `User` model create another `has_many` association named `favorite_movies` that goes *through* the existing `favorites` association.

   Hint:

   The source of the relationship is `:movie` and it goes `through` the `favorites` association.

   Answer:

   ```
   class User < ApplicationRecord
     has_many :reviews, dependent: :destroy
     has_many :favorites, dependent: :destroy
     has_many :favorite_movies, through: :favorites, source: :movie

     # existing code
   end
   ```

And that's all there is to it! We can now efficiently traverse from one side of the relationship to the other. Not too shabby for 2 lines of code!

2. Experiment with the Associations

Let's jump into a `console` session and see what these 2 lines of code let us do...

1. First, find a `Movie` that you created favorites for in the previous exercise, assign it to a `movie` variable.

   Answer:

   ```
   >> movie = Movie.find_by(title: "Black Panther")
   ```

2. To check that the `through` association works, ask the movie for its fans.

   Hint:

   When you declared `has_many :fans` in the `Movie` model, a `fans` instance method was dynamically defined. Because the association goes through `favorites` with `user` as the source, the `fans` method returns an array of `User` objects representing the users who favorited the movie.

Answer:

```
>> movie.fans
```

You should get back an array that contains the users who fave'd that movie.

Make sure to appreciate the SQL statement that was generated to query the database. It used an `INNER JOIN` to select all the users in the `users` table that have an `id` matching the `user_id` in the `favorites` table, but only for those rows in `favorites` where the `movie_id` also matches the `id` of the `movie` object. In short, the query joins together the movie and the users who favorited that movie by going through the `favorites` table. And it does all this in just **one query**! Pretty cool.

3. Then find a `User` that you created favorites for in the previous exercise, and assign it to a `user` variable.

   Answer:

   ```
   >> user = User.find_by(name: "Daisy")
   ```

4. Ask the user for her favorite movies, as a quick check of the `through` association traversing from the user side of things.

   Hint:

   When you declared `has_many :favorite_movies` in the `User` model, a `favorite_movies` instance method was dynamically defined. Because the association goes through `favorites` with `movie` as the source, the `favorite_movies` method returns an array of `Movie` objects representing that user's favorite movies.

   Answer:

   ```
   >> user.favorite_movies
   ```

   You should get back an array that contains the movies that the user fave'd. Again, it joins together the user and their favorite movies in just one query!

3. List a Movie's Fans

Now that we have some example fans in the database and we can access them using the `through` association, let's display the fans on the movie `show` page.

1. To set up the new data for the movie's show page, fetch the fans for the movie and assign the result to a `@fans` instance variable.

   Hint:

   Where would you do that? The controller is responsible for setting up the data required by a view. So all the data for the movie `show` page should be set up in instance variables by the `show` action of the `MoviesController`. In that action, call the `fans` method on the movie and assign the result to a `@fans` instance variable.

   Answer:

   ```
   def show
     @movie = Movie.find(params[:id])
     @fans = @movie.fans
   end
   ```

2. Then update the movie `show` template to generate a list of the movie's fans with each fan's name linking to their profile page, if any fans are present. To pull in some styling, list the fans in an `aside` tag directly under the `div` with a class of `details`.

   Answer:

   ```
   <aside>
     <% if @fans.present? %>
       <h4>Fans</h4>
       <ul>
         <% @fans.each do |user| %>
           <li>
             <%= link_to user.name, user %>
           </li>
         <% end %>
       </ul>
     <% end %>
   </aside>
   ```

3. Over in your browser, go to the show page for a movie that already has fans and you should see links for the users who are fans of the movie. Fan-tastic!

If you then click on a fan (you'll need to be signed in), you'll notice that their profile page doesn't list their favorite movies. So let's fix that next...

4. List a User's Favorite Movies

Clearly a user's profile page should list that user's favorite movies. Sound easy enough? There's nothing new here, so you should totally be able to do this with one hand tied behind your back.

1. Set up the new data for the user's profile page.

   Hint:

   In the `show` action of the `UsersController`, call the `favorite_movies` method on the user and assign the result to a `@favorite_movies` instance variable.

   Answer:

   ```
   def show
     @user = User.find(params[:id])
     @reviews = @user.reviews
     @favorite_movies = @user.favorite_movies
   end
   ```

2. Then in the user `show` template generate a list of all the favorite movies underneath the list of reviews. Link each favorite movie image back to the movie page.

   Answer:

```
<% if @favorite_movies.present? %>
  <h3>Favorite Movies</h3>
  <div class="favorite-movies">
    <% @favorite_movies.each do |movie| %>
      <%= link_to image_tag(movie.image_file_name), movie %>
    <% end %>
  </div>
<% end %>
```

3. Check your handiwork in the browser!

Solution

The full solution for this exercise is in the `through-1` directory of the code bundle.

Bonus Round

Alternate Way to Create Fans

In the previous exercise you created a favorite by calling the `create!` method on the `movie.favorites` association, like so:

```
>> movie.favorites.create!(user: user)
```

You might be interested to learn that another way to do this is by adding `User` objects to the `fans` association using the << operator. For example, suppose you had three `User` objects referenced by the variables `sally`, `jose`, and `susan` respectively. You could make them fans of a movie like so:

```
>> movie.fans << sally
>> movie.fans << jose
>> movie.fans << susan
```

You can also call the `include?` method on the `fans` association to check whether a particular user is already a fan:

```
>> movie.fans.include?(sally)
=> true
```

Mix these together, and you get a one-liner for adding a fan if they aren't already a fan:

```
>> movie.fans << sally unless movie.fans.include?(sally)
```

Show Each Fan's Profile Image

The list of fans on the movie `show` page could also use a bit more personality! Consider displaying a small version of each fan's profile image next to their name in the list. And if you added a `username` to each user in a previous bonus exercise, consider showing that as well.

Answer:

```
<li>
  <%= profile_image(user, size: 35) %>
  ...
</li>
```

Query for Movie Critics

Suppose you wanted to be able to ask a movie for all of the users who have reviewed it (call them "critics") using just one query. For example, calling `movie.critics` should return all the users that wrote a review for that movie. How would you do that?

Hint:

Declare a `has_many` association between movies and users called `critics` that goes *through* the `reviews` association. You'll need to set the `source` option to be `:user` so that Rails knows that a critic is really a user.

Answer:

```
class Movie < ApplicationRecord
  has_many :reviews, dependent: :destroy
  has_many :critics, through: :reviews, source: :user

  has_many :favorites, dependent: :destroy
  has_many :fans, through: :favorites, source: :user

  # existing code
end
```

Wrap Up

As we've seen, one of the most powerful features of Active Record is the ability to declare a variety of associations between different models: one-to-many, many-to-many, and through associations, for example. Traditionally, you'd be saddled with writing custom SQL queries to link everything together in the database and create corresponding Ruby objects. But by following Rails conventions, a few lines of declarative Ruby code takes care of all that for you. A through association is a perfect example of the true power of Active Record. And once you start using them, you'll begin to see common usage patterns. In fact, we'll use another through association a bit later.

In the meantime, our app is crying out for a "Fave" button so users can quickly and easily show some love for their top picks!

Through Associations: Part 2

Exercises

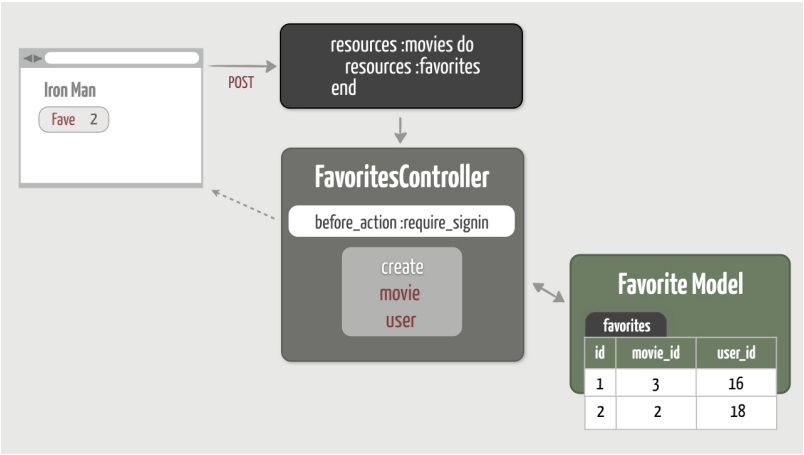
Objective

It's pretty obvious that our app needs a "Fave" button! And if fan changes their mind, we need an "Unfave" button, too. We're in a good position to make this work since our models, tables, and associations are already set up. So in this exercise, we'll focus on the UI side for creating (and deleting) a favorite.

To do that, we need to:

1. Nest the `favorites` resource inside of the `movies` resource so that requests to the `FavoritesController` always include a movie ID in the URL.
2. Generate a "Fave" button on the movie `show` page that runs the `create` action in the `FavoritesController`.
3. Define a `create` action in the `FavoritesController` that creates a new favorite in the database which is associated to both the movie specified in the URL and the signed-in user.
4. If the signed-in user has already fave'd the movie, generate an "Unfave" button on the movie `show` page that runs the `destroy` action in the `FavoritesController`.
5. Define a `destroy` action in the `FavoritesController` that deletes the appropriate favorite from the database.
6. Write a custom view helper method that determines whether to display the "Fave" or "Unfave" button.

Here's a snapshot of our objective for creating a favorite with a button:



1. Nest the Favorite Resource

Before we get too far along, we know that we always want to create favorites for a specific movie. So the URL for creating a favorite will always need to specify the movie that's being favorited. As we learned earlier in the course, we can require all favorite-related URLs to include a movie ID by *nesting* the `favorites` resource inside of the `movies` resource.

So let's go ahead and do that first...

1. Pop open the `config/routes.rb` file and you'll notice that the resource generator for `favorites` added this line:

   ```
   resources :favorites
   ```

   Move that line so that `favorites` are a nested resource inside of the `movies` resource. And since we only need routes for the `create` and `destroy` actions, use the `only` option to only generate routes for those actions. (In the video we generated all the like-related routes, but sometimes it's nice to only generate what's needed.)

   Answer:

   ```
   resources :movies do
     resources :reviews
     resources :favorites, only: [:create, :destroy]
   end
   ```

2. As a refresher of what that gives you in terms of routes, check out the defined routes. You should see the following *two* favorite-related routes (we've left out the other routes):

   ```
   Helper                  HTTP Verb              Path                                  Controller#Action
   movie_favorites_path      POST       /movies/:movie_id/favorites(.:format)         favorites#create
   movie_favorite_path       DELETE     /movies/:movie_id/favorites/:id(.:format)     favorites#destroy
   ```

   Notice that *every* path to the `FavoritesController` now requires a `movie_id` parameter in the URL, which is exactly what we need. Also notice that with nested routes the names of the route helper methods include the singular name of the parent, `movie` in this case.

3. We're going to need to generate a "Fave" button that creates a new favorite. Can you guess which route helper method we'll use to generate the URL that the button will POST to? What about the "Unfave" button?

2. Generate a "Fave" Button

Next we need to generate a "Fave" button on the movie `show` page. Why a button rather than a link? Well, we expect clicking a link to simply take us to a new page and not actually change anything. Pressing a button, on the other hand, comes with the expectation that something changes. In our case, hitting the "Fave" button should immediately create a new favorite in the database. So a button is the way to go here.

1. First, did you identify the name of the route helper method that generates a URL to create a new favorite? In this case, we want the "Fave" button to run the `create` action of the `FavoritesController` so here's the route:

```
movie_favorites_path        POST        /movies/:movie_id/favorites(.:format)          favorites#create
```

2. On the movie `show` page, inside the `div` with a class of `image` directly after the `img` tag, generate a "Fave" button using the route helper method. Make sure it's only displayed if a user is currently signed-in. Put it inside a `div` with a class of `faves` to pull in our CSS styles.

   Hint:

   The `button_to` helper takes two parameters: the name to be displayed on the button and the URL where the button will POST. To generate the URL, use the `movie_favorites_path` route helper method. The route has a `:movie_id` placeholder, so you need to fill it in by passing `@movie` as a parameter to `movie_favorites_path`.

   Answer:

```
<% if current_user %>
  <div class="faves">
    <%= button_to "❤️ Fave", movie_favorites_path(@movie) %>
  </div>
<% end %>
```

3. While you're here, display the total number of faves directly under the button. Put the count inside a `div` with a class of `count` to take advantage of our CSS styles.

   Hint:

   The number of faves is the same as the number of fans, so use the `@fans` variable that's already accessible in this template.

   Answer:

```
<% if current_user %>
  <div class="faves">
    <%= button_to "❤️ Fave", movie_favorites_path(@movie) %>
    <div class="count">
      <%= @fans.size %>
    </div>
  </div>
<% end %>
```

4. Then, back in your browser, navigate to a movie show page. If you're signed in you should see an enticing "Fave" button and the number of faves for that movie.

5. View the source of the generated page, and you'll discover that `button_to` generated a tidy form that POSTs to `/movies/1/favorites`, for example. Unlike links which issue a `GET` request, a button generated with `button_to` will issue a `POST` request by default. And that's exactly the HTTP verb required for our route to match!

3. Implement the Create Action

In the `create` action of the `FavoritesController` we need to create a new favorite in the database, making sure it's associated to a movie and a user. Doing this should feel familiar as you've already created a favorite in the console. The only difference here is the favorite needs to be associated with the movie *specified in the URL* and the *signed-in* user.

1. Implement the `create` action so that it creates an appropriate favorite and redirects to the movie `show` page.

   Hint:

   First, find the movie being favorited using the `:movie_id` parameter. Then use `@movie.favorites.create!` just like you did in the console earlier. This time, however, associate the favorite with the signed-in user which you can get by calling `current_user`.

   Answer:

```
def create
  @movie = Movie.find(params[:movie_id])
  @movie.favorites.create!(user: current_user)

  # or append to the through association
  # @movie.fans << current_user

  redirect_to @movie
end
```

2. For this to work, you need to require that a user is always signed in before running the `create` action, or any other action in the `FavoritesController` for that matter. Remember how to do that?

   Answer:

```
class FavoritesController < ApplicationController
  before_action :require_signin

  # existing code
end
```

3. Back in your browser, give it a whirl by hitting the "Fave" button a few times. Each time, you should see the favorite count increment and the signed-in user's name added to the list of fans. Also, if you navigate over to a fan's profile page, the movie title should be added to their list of favorite movies.

High-fives all around!

4. Only Allow One Fave Per User

That works good... perhaps a little *too* good. Overzealous users can fave a movie multiple times, unfairly driving up the fave count. If the user has already fave'd the movie, we should remove the "Fave" button and instead show an "Unfave" button.

1. To do that, you first need a way to check whether the current user has already faved the movie. The `console` is a great place to figure out queries like this, so hop into `console` session and we'll work through a solution.

   Start by finding the user you're currently signed-in as and the movie you just faved while you were in the browser:

   ```
   >> user = User.find_by(name: "Lucy")

   >> movie = Movie.find_by(title: "Captain Marvel")
   ```

   Now, you need to run a query that checks if the `favorites` database table already has a row that joins that particular movie and user. To do that, you could try to find a favorite that has the matching user and movie IDs, like so:

   ```
   >> favorite = Favorite.find_by(user_id: user.id, movie_id: movie.id)
   ```

   That certainly works, but here's the preferred way to do the same thing:

   ```
   >> favorite = user.favorites.find_by(movie_id: movie.id)
   ```

   The difference is we're using the `user.favorites` association in this case. The beauty of using the association is that it automatically scopes the query to only those favorites with a `user_id` foreign key matching the ID of the `user` object. We then tack on a `find_by` which further scopes the query to only those favorites with a `movie_id` foreign key matching the ID of the `movie` object. In this way, the query is automatically scoped to a specific user.

   And when you execute that line you'll see that it performs just one database query, which looks something like this:

   ```
    SELECT "favorites".* FROM "favorites" WHERE "favorites"."user_id" = ? AND "favorites"."movie_id" = 3 LIMIT 1  [["user_id", 1]]
   ```

   So even though it would appear that the tacked-on `find_by` would generate a second query, Rails is smart enough to combine it all into one query.

2. Now that you've isolated the code to run the query, it's time to use it in the app. Over in the `show` action of the `MoviesController`, use that same query to check if the signed-in user has already faved the movie being shown. Assign the resulting favorite to a `@favorite` instance variable. We'll use the existence of `@favorite` as a flag to indicate that the user has already faved the movie.

   Hint:

   The difference from the console is that inside the action you need to use `current_user` as the user and `@movie` as the movie. You also need to check that a user is signed in since the `show` action doesn't require a signed-in user (and that's as it should be).

   Answer:

   ```
   def show
     @movie = Movie.find(params[:id])
     @fans = @movie.fans
     if current_user
       @favorite = current_user.favorites.find_by(movie_id: @movie.id)
     end
   end
   ```

3. Then change the `show` template to hide the "Fave" button if the movie has already been faved. To do that, you'll need a conditional that checks if `@favorite` has a value. In other words, you only want to show the "Fave" button if there is no current favorite. Go ahead and show a placeholder "Unfave" button (use `'#'` as the URL for now) if a current favorite already exists.

   Hint:

   For now, we're simply using the `@favorite` variable as a flag to indicate that a favorite already exists. A bit later we'll actually use the value of the favorite object.

   Answer:

   ```
   <% if current_user %>
     <div class="faves">
       <% if @favorite %>
         <%= button_to "♡ Unfave", "#" %>
       <% else %>
         <%= button_to "❤️ Fave", movie_favorites_path(@movie) %>
       <% end %>
       <div class="count">
         <%= @fans.size %>
       </div>
     </div>
   <% end %>
   ```

4. Finally, visit a show page for a movie you already faved (your browser is probably already on that page). The "Fave" button should be replaced by an "Unfave" button. Great - that means a user can only fave a movie once!

But of course the "Unfave" button doesn't do anything, yet...

5. Make the "Unfave" Button Work

If a user hits that sad "Unfave" button, we need to delete the favorite that joins the signed-in user with the movie being shown. By convention, that should happen in the `destroy` action of the `FavoritesController`. So we need to configure the "Unfave" button to run the `destroy` action. To do that, we'll use a different route helper method.

1. Start by reviewing all the favorite-related routes. Here's the route to the `destroy` action we're aiming for:

   ```
   movie_favorite_path        DELETE     /movies/:movie_id/favorites/:id(.:format)     favorites#destroy
   ```

Because a favorite is nested under a specific movie, when accessing a favorite such as to delete a favorite, the route requires *two* parameters in the URL: `:movie_id` and `:id`. The `:movie_id` parameter identifies the ID of the parent movie and the `:id` identifies the ID of the favorite to delete. Notice also that this route will only be triggered by a `DELETE` request.

2. Now use that route helper method to generate the URL for the "Unfave" button. This is where that `@favorite` variable comes in handy, because the URL needs to include the id of the favorite you intend to delete. And lo and behold, the favorite you want to delete is represented by the `@favorite` variable.

   Hint:

   The route has two placeholders, so you have to pass the `movie_favorite_path` method two parameters: the `@movie` object and the `@favorite` object. You also need to override the HTTP method so it's `DELETE` rather than `POST` in order to trigger the delete route. When using `link_to`, you do that by setting the `turbo_method` data attribute. But when using `button_to`, you can simply set the `method` option to `:delete`.

   Answer:

   ```
   <%= button_to "♡ Unfave", movie_favorite_path(@movie, @favorite),
         method: :delete %>
   ```

3. As a quick sanity check that the button is wired up correctly, hit the "Unfave" button and it should try (unsuccessfully as you'll see in the server log) to run the `destroy` action of the `FavoritesController`.

4. Following that cue, implement the `destroy` action. It needs to destroy the favorite that has the id embedded in the URL. Once the favorite has been destroyed, redirect to the show page for the movie that's also identified in the URL.

   Here's the catch: It's very important that you only destroy the favorite if it was created by the signed-in user. Otherwise it's possible for a malicious signed-in user to generate URLs that end up deleting favorites created by other users.

   Hint:

   To find the favorite to delete, use the `current_user.favorites` association which automatically scopes the query to only those favorites created by the current user. Then further scope the query by calling `find` with the favorite id specified in `params[:id]`. Then call `destroy` on the resulting `Favorite` object. You can then use `params[:movie_id]` to find the specified movie and redirect to its show page. You can use instance variables for the favorite and the movie, but it's not necessary since they aren't displayed on a template as a result of running the `destroy` action. Local variables work just as well. Finally, you do not need to set the redirection `status` to `:see_other` since `destroy` was triggered by a button rather than a link.

   Answer:

   ```
   def destroy
     favorite = current_user.favorites.find(params[:id])
     favorite.destroy

     movie = Movie.find(params[:movie_id])
     redirect_to movie
   end
   ```

5. Finally, go back to the browser and hit the "Unfave" button. The fave count should decrement by one.

6. If at any point you want to delete all the favorites to get a clean slate, just jump into a `console` session and use:

   ```
   >> Favorite.delete_all
   ```

Now you should only be able to fave a movie once! Change your mind? No problem. Just unfave it!

6. Write a Custom View Helper

Conditional statements in view templates are often difficult to read at a glance. You can easily tidy these up by writing a well-named view helper method. Let's get some more practice doing that...

1. In the `FavoritesHelper`, define a `fave_or_unfave_button` method that encapsulates the conditional statement for determining which button to display.

   Hint:

   You can copy the ERb code into the method and then remove the Erb tags, as we did in the video. Remember, however, that view helpers should not rely on instance variables being set. Instead, the method will need to take two arguments: a movie and a favorite. Then inside the method use the `movie` and `favorite` local variables rather than instance variables.

   Answer:

   ```
   module FavoritesHelper
     def fave_or_unfave_button(movie, favorite)
       if favorite
         button_to "♡ Unfave", movie_favorite_path(movie, favorite), method: :delete
       else
         button_to "❤️ Fave", movie_favorites_path(movie)
       end
     end
   end
   ```

2. Then call that helper from the movie `show` template in place of the original conditional statement.

   Answer:

   ```
   <div class="faves">
     <%= fave_or_unfave_button(@movie, @favorite) %>
     <div class="count">
       <%= @fans.size %>
     </div>
   </div>
   ```

Solution

The full solution for this exercise is in the `through-2` directory of the code bundle.

Bonus Round

Remove Duplication

You may have noticed a wee bit of duplication in the `create` and `destroy` actions of the `FavoritesController`. Both actions look up the movie using identical code. Remember how to clean up that sort of duplication?

1. Write a `private` method named `set_movie` that fetches the movie and sets the `@movie` instance variable.

   Answer:

   ```
   private

     def set_movie
       @movie = Movie.find(params[:movie_id])
     end
   ```

2. Then make sure to run the `set_movie` method *before* running any action's code.

   Answer:

   ```
   before_action :set_movie
   ```

3. Finally, as your reward, remove the duplicated line of code from each action.

Wrap Up

If your brain happens to feel a bit sore after the last few sections that's because it's swollen with a good deal of new info! You've learned a lot about many-to-many relationships:

- Strictly speaking, there is no such thing as a many-to-many relationship. Instead, this type of relationship is achieved with **two one-to-many relationships** with a join model in between.
- At a minimum, **a join table has two foreign keys**: one key referencing each of the entities being joined. That's the case with our `favorites` join table. A join table can also have other columns, for example "stars" and "comments" as is the case with our `reviews` join table.
- **Through associations** are a powerful yet simple way to jump between records that are related to each other, but stand two tables apart. They are also more efficient than hopping table to table. And you can give them descriptive names that better reflect your domain, such as fans!
- You can **traverse across many-to-many relationships in multiple ways** according to what pieces of data you want. For example, you can call `movie.fans` when you want to traverse through the `favorites` relationship, and `movie.favorites` if you want to work just with the join table. Similarly, you call `user.favorite_movies` or `user.favorites` depending on your requirements.
- When you **don't need a form to create a many-to-many relationship** (because you already know the two pieces of information required), you can use a button that sends a POST request to the `create` action. Doing so will immediately create a new association in the join table. You can also configure a button to send a DELETE request to delete an existing association from the join table.
- It's always a good idea to **make sure your associations are working correctly in the console first** before making the trickle-down changes to the other parts of your app.

Coming up next, we'll look at another way to design a many-to-many relationship. In the case of movies having fans, we designed favorites as its own resource. Next we want a movie to be able to have many genres, such as romance and comedy, or fantasy and comedy. How would we design that type of relationship? And how could we let admins assign multiple genres to a movie when its created or updated? That's up next...

Dive Deeper

To learn more about through associations, refer to Section 2.4 in Rails Guide: Active Record Associations.

Many-to-Many with Checkboxes: Part 1

Exercises

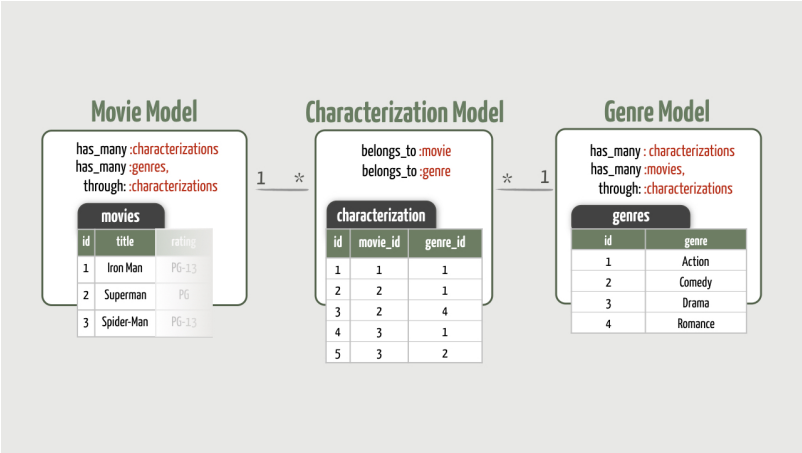
Objective

An online movie review app wouldn't be worth its weight in ticket stubs if it didn't list genres for its movies. So we'll need yet another many-to-many relationship. This time the relationship will be between movies and genres such that a movie can have multiple genres and a genre can be assigned to multiple movies.

We'll start by setting up the database tables and models in this exercise, and then tackle the UI in the next exercise. For now, we'll need to:

- Generate a `genre` resource
- Generate a `Characterization` join model. (Naming is hard! Even though it's not necessarily fun to type out that name, it does fit the domain fairly well. If you come up with a better name, by all means go for it!)
- Declare the model associations, including `through` associations

If you like diagrams (can you tell we do?), here's our goal:



Given everything you've learned about many-to-many relationships, this exercise is well within your reach...

1. Generate the Genre Resource

First we need to create a new resource to represent a genre, which simply has a name such as "Action" or "Drama". We'll use the resource generator to get a model and migration. In the event you decide to create a web interface for managing genres (as a bonus), you'll also have a controller and resource routes.

1. Start by generating a resource named `genre` with a `name` string field.

   Answer:

   ```
   rails g resource genre name:string
   ```

2. Then apply the generated migration to create the `genres` database table.

   Answer:

   ```
   rails db:migrate
   ```

3. Next, add validations to the generated `Genre` model to ensure that a genre always has a non-blank and unique name.

   Answer:

   ```
   class Genre < ApplicationRecord
     validates :name, presence: true, uniqueness: true
   end
   ```

4. Then hop into a console session and create a handful of example movie genres.

   Answer:

   ```
   Genre.create(name: "Action")
   Genre.create(name: "Comedy")
   Genre.create(name: "Drama")
   Genre.create(name: "Romance")
   Genre.create(name: "Thriller")
   Genre.create(name: "Fantasy")
   Genre.create(name: "Documentary")
   Genre.create(name: "Adventure")
   Genre.create(name: "Animation")
   Genre.create(name: "Sci-Fi")
   ```

   You might also consider creating example genres in your `db/seeds.rb` file so that a stock set of genres can easily be recreated.

2. Generate the Characterization Join Model

Next, we need a `Characterization` join model that will connect movies to genres. That is, any particular row in the `characterizations` table will effectively join a row in the `movies` table and a row in the `genres` table. In this case we don't need a controller, so we'll use the model generator to generate a join model and migration file.

1. Generate a model named `Characterization` that references both a `movie` and a `genre`.

   Answer:

   ```
   rails g model characterization movie:references genre:references
   ```

2. Can you guess what the generated migration file looks like?

   Answer:

   ```
   class CreateCharacterizations < ActiveRecord::Migration[7.0]
     def change
       create_table :characterizations do |t|
         t.references :movie, null: false, foreign_key: true
         t.references :genre, null: false, foreign_key: true

         t.timestamps
       end
   ```

```
  end
end
```

3. Then create the `characterizations` database table and you're done with this step!

   Answer:

   ```
   rails db:migrate
   ```

3. Declare the Model Associations

The next step is to declare associations in the models so that a movie can exist in many genres and a genre can contain many movies. We want our queries to be efficient, so we'll also declare *through* associations that let us navigate directly from a movie to its genres, and vice versa.

1. First, the `Characterization` model needs to `belong_to` both a movie and a genre. That's already taken care of since you wisely used `movie:references` and `genre:references` when generating the model.

   Answer:

   ```
   class Characterization < ApplicationRecord
     belongs_to :movie
     belongs_to :genre
   end
   ```

2. Next up, in the `Movie` model declare a `has_many` association with `characterizations` and also a `has_many` association with `genres` that goes *through* the `characterizations` association.

   Answer:

   ```
   has_many :characterizations, dependent: :destroy
   has_many :genres, through: :characterizations
   ```

3. Then, over in the `Genre` model, set up a reciprocal association. Declare a `has_many` association with `characterizations` and also a `has_many` association with `movies` that goes *through* the `characterizations` association.

   Answer:

   ```
   has_many :characterizations, dependent: :destroy
   has_many :movies, through: :characterizations
   ```

4. Experiment with the Associations

Now that we have a many-to-many association between movies and genres, and we have example movies and genres in the database, let's see how we'd assign genres to movies in the console...

1. First, find the "Black Panther" movie and ask for its associated genres.

   Answer:

   ```
   >> movie = Movie.find_by(title: "Black Panther")
   >> movie.genres
   ```

   We haven't assigned any genres to the movie yet, so you should get back an empty array. But at least you know that the `through` association is set up correctly. Hey, check out that single `INNER JOIN` query!

2. Then find the "Action" genre and ask for its associated movies.

   Answer:

   ```
   >> genre = Genre.find_by(name: "Action")
   >> genre.movies
   ```

   The genre doesn't yet contain any movies, so you should get another empty array. But again, the query tells us that the `through` association is doing its job.

3. The movie "Black Panther" is certainly an action flick, so now you need to assign that genre to the movie.

   The `through` association gives you a couple different ways to do that. The simplest way is to add the genre object to the array of genres that are associated with the movie. You can get that array by calling `movie.genres`.

   Hint:

   You can "push" an object onto an array using the << operator.

   Answer:

   ```
   >> movie.genres << genre
   ```

   You should see a SQL `INSERT` statement executed immediately that creates a new `characterizations` row in the database. Notice that the `movie_id` and `genre_id` foreign keys are automatically set to reference the respective `movie` and `genre` objects.

4. Now ask the movie for its genres again.

   Answer:

   ```
   >> movie.genres
   ```

   This time you should get back an array that contains the one genre you added to the array.

Just for practice, go ahead and ask the genre for its associated movies.

Answer:

```
>> genre.movies.reload
```

You should get back an array that contains the one movie characterized as being in that genre.

5. The `through` association also lets us access the ids of the genres that are associated with the movie. To get the array of genre ids, read the `genre_ids` attribute of the `movie` object.

   Answer:

   ```
   >> movie.genre_ids
   ```

   You should get back an array that contains a single genre id, 1 for example.

6. You can also *assign* an array of genre ids to the `genre_ids` attribute. Use that technique to assign the "Action", "Adventure", and "Sci-Fi" genres to the movie.

   Answer:

   ```
   >> movie.genre_ids = [1, 8, 10]
   ```

   This automatically creates two new rows in the `characterizations` table! The "Action" genre was already associated with the movie, so a new row wasn't created for it. However, new rows were created to join the movie with the "Adventure" and "Sci-Fi" genres.

7. Now suppose you only wanted to associate the "Action" and "Sci-Fi" genres with the movie, and disassociate the "Adventure" genre. To do that, assign an array of two genre ids.

   Answer:

   ```
   >> movie.genre_ids = [1, 10]
   ```

   This automatically deletes one row in the `characterizations` table! Notice that a SQL `DELETE` statement was executed to delete the row that joined the movie with the "Adventure" genre. Also, because "Action" and "Sci-Fi" were already associated with the movie, no new rows were created.

8. Any guesses as to how you'd disassociate all the genres from the movie?

   Hint:

   Simply assign an empty array to `movie.genre_ids`.

   Answer:

   ```
   >> movie.genre_ids = []
   ```

9. Now, to give us something to work with going forward, assign the "Action", "Adventure", and "Sci-Fi" genres back to the movie.

   Answer:

   ```
   >> movie.genre_ids = [1, 8, 10]
   ```

10. Finally, ask the movie for its genres one more time.

    Answer:

    ```
    >> movie.genres
    ```

    You should get back an array that contains three genres. Notice that even though we assigned genre ids, the association knows to fetch the appropriate genres matching those ids.

Hey, that's a handy way to assign multiple genres!

Solution

The full solution for this exercise is in the `checkboxes-1` directory of the code bundle.

Wrap Up

Now that we have some genres in the database and a movie with multiple genres assigned, in the next exercise we'll use the `through` association to list a movie's genres. We'll also design a UI that uses checkboxes to let admin users assign multiple genres to a movie.

Many-to-Many with Checkboxes: Part 2

Exercises

Objective

Since we assigned genres using the console in the previous exercise, we'll start by listing a movie's genres on the movie `show` page.

Then we need to change the movie form to include a set of checkboxes that can be checked or unchecked to assign specific genres when a movie is either created or updated.

1. List a Movie's Genres

You should be able to knock this out on your own!

1. In the `show` action of the `MoviesController`, fetch the movie's associated genres, in alphabetic order by name, and assign them to an instance variable.

   Answer:

   ```
   def show
     @movie = Movie.find(params[:id])
     @fans = @movie.fans
     @genres = @movie.genres.order(:name)

     if current_user
       @favorite = current_user.favorites.find_by(movie_id: @movie.id)
     end
   end
   ```

2. Then list out the genres on the movie `show` page, inside the `aside` tag after the list of movie fans.

   Answer:

   ```
   <% if @genres.present? %>
     <h4>Genres</h4>
     <ul>
       <% @genres.each do |g| %>
         <li><%= g.name %></li>
       <% end %>
     </ul>
   <% end %>
   ```

3. Back in your browser, the "Black Panther" movie should now be characterized as being in the genres you assigned in the console in the previous exercise.

2. Add Checkboxes to the Movie Form

So while we can assign genres in the console, our admins will most certainly want to assign genres in the web interface when they create or edit movies. To support that, we'll put checkboxes on the movie form. Then admins can check or uncheck specific genres that best characterize the movie.

Keep in mind that when an admin user clicks "Create New Movie" or "Update Movie", they are routed to the **Movies**Controller and the movie form is displayed in the browser. It's in this movie form that we want to add the genre checkboxes. When the form is then submitted, it POSTs to the `create` action in the `MoviesController`. So, to assign genres to movies (create a row in the `characterizations` table), we don't need a **Characterizations**Controller. Instead, the assignment happens in the `MoviesController`.

1. Start by signing in as an admin and then editing the "Black Panther" movie. You won't see any checkboxes yet, but this sets you up for a big reveal later.

2. Then change the movie form (it's a partial) to generate a checkbox for each possible genre, in alphabetic order by name.

   Hint:

   Use the `collection_check_boxes` helper method. It takes four parameters: the name of the attribute (`:genre_ids`), the collection of objects (`Genre.order(:name)`), the value to use for the checkbox input (`:id`), and the text to be displayed next to the checkbox (`:name`). Put it in a `div` with a class of `checkboxes` to trigger our CSS styling.

   Answer:

   ```
   <div class="checkboxes">
     <%= f.collection_check_boxes(:genre_ids, Genre.order(:name), :id, :name) %>
   </div>
   ```

3. Now for the first part of that big reveal, reload the form. You should see a list of genre names with a checkbox to the left of each one. Whichever genres you assigned in the console should already be checked!

4. To see what's going on behind the scenes, view the page source and you should see a pair of `input` and `label` elements for each genre, like so:

   ```
   <input type="checkbox" value="1" checked="checked" name="movie[genre_ids][]" id="movie_genre_ids_1" />
   <label for="movie_genre_ids_1">Action</label>

   <input type="checkbox" value="2" name="movie[genre_ids][]" id="movie_genre_ids_2" />
   <label for="movie_genre_ids_2">Comedy</label>

   ...
   ```

   Here's what happened: The `collection_check_boxes` helper generated one `input` and `label` pair for each object in the collection returned by `Genre.order(:name)`. It called each genre object's `id` attribute to populate the `value` attribute of the `input` element. It called each genre object's `name` attribute to generate the text in the `label` element. And it called the movie's `genre_ids` attribute to determine which genres should be "checked" by default.

   Pretty slick!

3. Handle the Form Data

1. Just to see the form data that's being submitted, add `fail` as the first line in the `update` action of the `MoviesController`.

   ```
   def update
      fail
      ...
    end
   ```

2. Then submit the form and on the error page check out the parameters under the "Request" heading. You should see something like this:

```
{"_method"=>"patch",
 "authenticity_token"=>"[FILTERED]",
 "movie"=>
  {"title"=>"Black Panther",
   "description"=>"T'Challa, heir to the hidden but advanced kingdom of Wakanda, must step forward to lead his people into a new future a
   "rating"=>"PG-13",
   "released_on(1i)"=>"2018",
   "released_on(2i)"=>"2",
   "released_on(3i)"=>"16",
   "total_gross"=>"1346913161.0",
   "director"=>"Ryan Coogler",
   "duration"=>"134 min",
   "image_file_name"=>"black-panther.png",
   "genre_ids"=>["", "1", "8", "10"]},
 "commit"=>"Update Movie",
 "id"=>"3"}
```

In addition to all the expected keys and values, notice that the parameters hash includes a `genre_ids` key whose value is an array. And that array contains the ids of the genres that were checked in the form. (Remember, the ids are in the `value` attributes of the checkbox `input` elements.)

Hey, this is really convenient! Earlier in the console, we saw how we could add/remove genres based on an array of genre ids. And when we submit the form, it gives us an array of genre ids representing the "checked" genres. Perfect!

3. Back in the `update` action, remove the `fail` line. Now, you might think we need to do something special in `update` to handle the genre ids. But this existing line takes care of all that:

   ```
   @movie.update(movie_params)
   ```

   In the same way, we don't need to update the `create` action because it already initializes a movie using the submitted form data, like so:

   ```
   @movie = Movie.new(movie_params)
   ```

   In both cases, the array of genre ids in the request parameters will get assigned to the `genre_ids` attribute, just like any other piece of form data. And we know that assigning to that attribute automatically creates or deletes rows in the `characterizations` table. So you don't have to do anything special here. (Don't you love it when a plan comes together?!)

4. However, you do need to update the `movie_params` method to allow genre ids to be mass-assigned. Because we're dealing with an array, we have to set the `genre_ids` key to an empty array. Here's how to do that:

   ```
   def movie_params
     params.require(:movie).permit(..., genre_ids: [])
   end
   ```

5. Now for the final big reveal! Hop back into your browser and check (or uncheck) specific genres. When you submit the form, you should end up on the movie `show` page and the list of genres should reflect your changes. And of course if you go back to the form, the associated genres should already be checked.

And with that, your admin users can easily assign multiple genres to a movie when it's created or updated. Nicely done!

Solution

The full solution for this exercise is in the `checkboxes-2` directory of the code bundle.

Bonus Round

Make a CRUD Interface for Genres

As our app stands, the list of genres can only be changed in the console. Although the set of familiar genres is fairly static and unlikely to change frequently, you still might consider standing up a web interface for managing genres. Bear in mind, only admin users should be able to create, update, and destroy genres.

You might also want a UI for listing out all the "comedy" movies or "action" movies. For example, on the genre `show` page, you could list the movies that are in that particular genre. And then on the `movie` show page, you could link each genre name so a user can see other movies in the same genre.

There's nothing new here to learn, but it's great practice if you want to layer in this functionality.

Wrap Up

In this section we put together a whole bunch of many-to-many concepts for a super-duper practice exercise. And we learned a little something new along the way: checkboxes! Hopefully at this point you're feeling more comfortable designing a variety of model associations.

Now, wouldn't it be nice to include links for showing hit or flop movies? That way you'd know which movies to watch (or skip) this weekend. We'll do that next using a new concept called *scopes*.

Dive Deeper

- If you have a one-to-many association and you want to allow a user to pick a parent for the child using radio buttons, check out the collection_radio_buttons helper.
- If you have a `belongs_to` association and you want to allow a user to select a single parent from a drop-down select box, check out the collection_select helper.

Custom Scopes and Routes: Part 1

Exercises

Objective

Currently our movie listing page shows only the released movies. That's a good start, but movie-savvy folks might also like to see the movies that haven't yet been released—the upcoming movies. Or perhaps the most recent movies.

To list movies that match different criteria, we'll first need to define custom queries that fetch a subset of movies from the database depending on the criteria. Earlier in the course we learned how to define custom queries by writing class-level methods in our models. For example, we're currently listing the released movies by calling the `Movie.released` method. And we could certainly continue writing class-level methods for our new queries.

But a more idiomatic, concise way to write custom queries in Rails is using the declarative style offered by the `scope` method. You can think of using `scope` as "syntactic sugar" for defining an class-level query method. Rails programmers always enjoy a good bit of syntactic sugar, and as such you'll see scopes used prevalently across Rails apps. So it's worth understanding how to use them properly.

In this exercise we'll focus on defining all the custom queries we need as scopes. As such, we'll need to:

- Convert the existing `released` class-level query method to a scope
- Declare an `upcoming` scope that returns all the movies that have *not* yet been released, ordered with the soonest movie first
- Declare a parameterized `recent` scope that returns an arbitrary number of released movies, for example the last 5 movies

Then in the next exercise we'll update the `index` action in the `MoviesController` to run the appropriate query depending on the URL.

So let's get rolling...

1. Convert 'Released' to a Scope

We'll start by converting the existing `released` class-level query method to a `scope`...

1. As a refresher, check out the `index` action in the `MoviesController` and you should see the following:

   ```
   def index
     @movies = Movie.released
   end
   ```

   So when you visit http://localhost:3000/movies, the `index` action calls the `released` custom query method we wrote earlier in the course. Therefore, you see a list of only the movies that have been released.

2. Drill down into the `released` class-level method in the `Movie` model and you'll see the query:

   ```
   def self.released
     where("released_on < ?", Time.now).order("released_on desc")
   end
   ```

   It fetches all the movies that have a released on date in the past, ordered with the most-recently released movie first.

   That class-level method works fine and dandy, but all it's really doing is defining a named query. The name of the query is the name of the method (`released`) and the query itself is the body of the method. A more concise way to define a named query is by using a `scope`, so let's do that...

3. Convert the `released` class-level method to a scope named `released`.

   Hint:

   The `scope` method takes two parameters. The first parameter is the name of the scope (a symbol) which is `:released` in this case. The second parameter is the custom query code. In this case, it's the same query code you currently have in the body of the `released` class-level method. However, in a `scope` declaration the query code must be wrapped with a lambda using either `lambda { }` or `-> { }` to make it a callable Ruby object (a `Proc` object). That way the query code gets evaluated every time the scope is called.

   Answer:

   ```
   scope :released, -> { where("released_on < ?", Time.now).order("released_on desc") }
   ```

   Don't forget to remove your `released` class-level method when you're done. The `scope` declaration will dynamically define the equivalent `released` class-level method for you.

4. Now hop over into a console session and call the `released` method that the `scope` defined.

   Answer:

   ```
   >> reload!

   >> Movie.released
   ```

   It should run the same query you had before, returning all the released movies. The point is, you can call scopes as if they were class-level methods.

So from outside of the `Movie` model, nothing has changed. The `index` action in the `MoviesController` continues to call `Movie.released` without caring how the query is defined. And that's a great example of why you always want to encapsulate your queries in the model!

2. Declare a New 'Upcoming' Scope

Now we're ready to declare a new `upcoming` scope...

1. Declare a new scope named `upcoming` that queries for all the movies that have *not* yet been released, ordered with the soonest movie first.

   Answer:

   ```
   scope :upcoming, -> { where("released_on > ?", Time.now).order("released_on asc") }

   # or

   scope :upcoming, lambda { where("released_on > ?", Time.now).order("released_on asc") }
   ```

2. Then check your work by running the query in the console.

   Answer:

   ```
   >> reload!

   >> Movie.upcoming
   ```

3. Declare a Parameterized 'Recent' Scope

We also need a `recent` scope that returns an arbitrary number of released movies. How many movies? You guessed it, that's a number parameter! And rather than duplicating the query code to fetch the released movies, we can instead reuse the `released` scope.

1. Declare a scope named `recent` that reuses the existing `released` scope and accepts the maximum number of movies as a parameter, with a default value of 5.

   Hint:

   Just as methods can be called with parameters, lambdas (all callable objects) can also be called with parameters. The lambda associated with the `recent` scope needs to accept the maximum number as a parameter. The syntax is a bit funky until you get used to it. The parameter needs to go in parentheses after the `->` or as a block parameter if you use `lambda`. Then inside of the block you can refer to the parameter. Use the `limit` method to limit the results to the maximum number. And just as a method parameter can have a default value, a lambda parameter can also have a default value.

   Answer:

   ```
   scope :recent, ->(max=5) { released.limit(max) }

   # or

   scope :recent, lambda { |max=5| released.limit(max) }
   ```

2. Over in a console, call the `recent` method with no parameters.

   Answer:

   ```
   >> reload!

   >> Movie.recent
   ```

   It should return a maximum of 5 movies, the default. Check out the generated SQL and notice it includes an `AND` clause that fetches all the movies that are both released and recent in one database query. This is incredibly powerful! It means we can chain multiple scopes together to compose one complex query out of several small queries.

3. Now query for the 1, 5, and 10 most-recent movies.

   Answer:

   ```
   >> reload!

   >> Movie.recent(1)
   >> Movie.recent(5)
   >> Movie.recent(10)
   ```

Solution

The full solution for this exercise is in the `scopes-1` directory of the code bundle.

Bonus Round

Convert 'Hits' and 'Flops' to Scopes

If you completed a previous bonus exercise where you declared `hits` and `flops` class-level query methods, then they are good candidates for scopes.

But let's suppose we want to further scope (constrain) the `hits` and `flops` queries to only include *released* movies. Movies don't become hits until they're released anyway. And movies that haven't been released will be flops by definition. So we only want released movies that are also hits or flops. Rather than duplicating the query code, you can instead chain the `released` scope together with the `hits` and `flops` scopes.

1. Convert the `hits` class-level method to a scope with the same name, and combine it with the existing `released` scope.

   Answer:

   ```
   scope :hits, -> { released.where("total_gross >= 300000000").order(total_gross: :desc) }
   ```

   Don't forget to remove the class-level method when you're done.

2. In the same way, convert the `flops` class-level method to a scope and reuse the existing `released` scope.

   Answer:

   ```
   scope :flops, -> { released.where("total_gross < 225000000").order(total_gross: :asc) }
   ```

   Again, don't forget to remove the class-level methods when you're done.

3. As a quick sanity check of your work, run each query in the console.

   Answer:

```
>> reload!

>> Movie.hits
>> Movie.flops
```

4. It's worth noting that you can continue to chain other familiar query methods onto a scope. For example, suppose the bean counters come to you and want to know which hit movies exceeded $400 million in total gross. The standard `hits` query queries for $300 million and above, but you can add a `where` clause to the chain like so:

   ```
   >> Movie.hits.where("total_gross > 400000000")
   ```

   This is a neat example of how scopes are lazily evaluated. When you call the `hits` scope, it doesn't immediately query the database. Instead, it builds a query object (an `ActiveRecord::Relation` object). When you then call `where` it modifies that query object to include the `where` conditions. Finally, when you hit Return in the console, it generates a single SQL query statement and performs the query. Inside of a Rails app, it's clever enough to only run the query when you begin to iterate through the results.

   How would you then count the number of hits over $400 million?

   Answer:

   ```
   >> Movie.hits.where("total_gross > 400000000").count
   ```

5. Finally, you might be surprised to know that scopes can also be called on a `through` association. For example, suppose you wanted to get all the hits and flops that a particular user favorited. Recall that a user already has a `through` association called `favorite_movies`. So to further scope the favorites to the hits and flops, you could do this:

   ```
   >> user = User.find_by(name: "Daisy")

   >> user.favorite_movies.hits

   >> user.favorite_movies.flops
   ```

Declare User-Level Scopes

Currently, the user listing page shows all users. Suppose you didn't want to list admin users, only non-admin users. Suppose you also wanted to list the users ordered by their name. How might you do that using small scopes chained together?

1. First, declare a `by_name` scope in the `User` model that returns all the users ordered by their name in alphabetical order.

   Answer:

   ```
   scope :by_name, -> { order(:name) }
   ```

2. Then declare a second scope in the `User` model that builds off the first scope but returns only those users who are not admins, ordered by their name.

   Answer:

   ```
   scope :not_admins, -> { by_name.where(admin: false) }
   ```

3. Finally, change the `index` action of the `UsersController` so that only non-admin users are listed.

   Answer:

   ```
   def index
     @users = User.not_admins
   end
   ```

More Scope Ideas

Want to continue building your scoping muscles? Good for you! Here are some more ideas:

- Suppose you wanted an easy way to fetch all the reviews that have been written in the past `n` days. For example, you might use this query to find all the reviews written in the past week (7 days) like so:

  ```
  Review.past_n_days(7)
  ```

  Or in the last month (30 days) like so:

  ```
  Review.past_n_days(30)
  ```

  Declare a `past_n_days` scope that takes the number of days as a parameter and returns the reviews written during that period. As a hint, remember in Rails you can use `3.days.ago` to get the date as of 3 days ago.

  Answer:

  ```
  scope :past_n_days, ->(days) { where("created_at >= ?" , days.days.ago) }

  # or

  scope :past_n_days, lambda { |days| where("created_at >= ?" , days.days.ago) }
  ```

- Keep the Hollywood bookkeepers happy by declaring scopes that return released movies that grossed less than or greater than a specified amount. In the console, they want to be able to run the queries like so:

  ```
  Movie.grossed_less_than(25000000)
  Movie.grossed_greater_than(500000000)
  ```

  Declare the scopes to make that possible.

Answer:

```
scope :grossed_less_than, ->(amount) { released.where("total_gross < ?", amount) }
scope :grossed_greater_than, ->(amount) { released.where("total_gross > ?", amount) }
```

Another Use for Lambdas

Now that you have a handle on Ruby lambdas, you might be interested to learn that you can pass a Ruby lambda as the second parameter to the `has_many` method to customize the query.

For example, suppose you wanted to change the ordering of movie reviews so that the most-recent review appeared first in the listing on http://localhost:3000/movies/1/reviews. To do that, change the `has_many :reviews` declaration in the `Movie` model like so:

```
has_many :reviews, -> { order(created_at: :desc) }, dependent: :destroy
```

You can use any of the usual query methods inside of `-> { }` to customize the query.

Wrap Up

Using scopes is a great way to define custom queries in the declarative style you've come to expect in Rails. When you begin to build your own app, take the time to declare concise, expressive queries as scopes that you can then use to compose more complex queries by chaining them together. In addition to making your application code easier to read, using scopes leads to more reusable, flexible, and efficient queries.

As we've seen, declaring a `scope` is simply a shortcut to defining the equivalent class method. Indeed, you can do anything in a class method that you can do in a scope. So you might be wondering why you'd ever use a class method instead of a scope.

Personally, we tend to use scopes when the query logic is fairly concise and uses simple `where`, `order`, and `limit` clauses, for example. When the query logic gets more complex, requires some computation or logic, or requires a large number of parameters, we'll use a class method instead.

In the next exercise we'll put our new scopes to work in the `index` action of the `MoviesController` so we can list movies in a variety of ways.

Dive Deeper

You might find it helpful to refer to the scope method documentation which includes other example uses.

Custom Scopes and Routes: Part 2

Exercises

Objective

Now that we've declared scopes for querying movies with specific criteria, it's time to put the scopes to work. Our objective is to be able to list upcoming and recent movies by putting links in the header that correspond to the following URLs:

- http://localhost:3000/movies/filter/upcoming
- http://localhost:3000/movies/filter/recent

So which action should handle those URLs? Well, the action needs to run the scope query and display the resulting movies in a formatted list. That's pretty close to what the existing `index` action of the `MoviesController` does. It queries for released movies and renders the `index.html.erb` view template which formats the movies listing quite nicely. And as a well-designed view template, it's not concerned with how those movies were fetched or even where they originated. It just expects a collection of movies to exist in an `@movies` instance variable.

Rather than defining a new action, we'll instead piggyback on the existing `index` action and its view template. In that action we'll determine which scope to call based on the last part of the URL. Then we'll assign the resulting movies to the `@movies` instance variable for the view template to list out.

1. List Upcoming and Recent Movies

1. Try browsing to http://localhost:3000/movies/filter/upcoming and and you'll get this expected error:

   ```
   Routing Error

   No route matches [GET] "/movies/filter/upcoming"
   ```

   To support this URL, we'll need a new route. And we want that route to also support browsing to http://localhost:3000/movies/filter/recent.

2. To do that, define a new route that uses a `:filter` parameter to match the URL path `movies/filter` followed by any scope name. For example, a request for `movies/filter/upcoming` should fill in the `:filter` parameter with the value "upcoming". And a request for `movies/filter/recent` should fill in the `:filter` parameter with the value "recent". When that route matches, it should send the request to the `index` action of the `MoviesController`

   Hint:

   Remember that whenever you see a colon (:) followed by a name in the route URL, that parameter will get filled in with whatever shows up at that place in the requested URL.

   Answer:

   ```
   get "movies/filter/:filter" => "movies#index"
   ```

3. Now reload http://localhost:3000/movies/filter/upcoming and you should see the movie listing, which means that the `index` action was run as intended. However, it's still listing all the released movies rather than the upcoming movies. When this route is matched, you need to tell the `index` action to only list upcoming movies.

4. To see that the `:filter` parameter flows through to the action, add `fail` as the first line in the `index` action of the `MoviesController` like so:

```
def index
  fail
  @movies = Movie.released
end
```

5. Then reload and on the resulting error page check out the parameters under the "Request" heading. You should see this:

```
{"filter"=>"upcoming"}
```

Perfect. So when the `movies/filter/upcoming` route invokes the `index` action, the `params` hash includes a `filter` key whose value is `upcoming`. In other words, the router simply passes the parameter name and value through to the `index` action. This comes in really handy! You can use the value of the `:filter` parameter as a flag to determine which query to run.

6. Back in the `index` action, remove the `fail` line. Instead, use a `case` statement to run the appropriate query depending on the value of the `:filter` parameter. If the value is `upcoming`, list the upcoming movies. If the value is `recent`, list the recent movies. You can use the default number (5) or explicitly set a number. Otherwise, if the `:filter` parameter has any other value, list the released movies.

Answer:

```
def index
  case params[:filter]
  when "upcoming"
    @movies = Movie.upcoming
  when "recent"
    @movies = Movie.recent
  else
    @movies = Movie.released
  end
end
```

It's important to note that the value of the route parameter need not be the same as the name of the scope. For example, you could arrange things so that if the value of `:filter` was `upcoming-movies` (the URL path would be `movies/filter/upcoming-movies`) then the `Movie.upcoming` query would be run. We decided to make the value of the route parameter the same as the name of the scope, just to keep things simple and direct.

7. Finally, make sure you have at least one upcoming movie in your database and then verify that http://localhost:3000/movies/filter/upcoming lists only the upcoming movies.

Also verify that http://localhost:3000/movies/filter/recent list the recent movies.

And verify that http://localhost:3000/movies lists all the released movies, same as before.

2. Add Links In The Header

To make it easy to see the upcoming and recent movies, let's add links to the header...

1. We'll need a route helper method to generate the links, but currently our route doesn't have an associated helper. So to get a helper method, give the route the name `filtered_movies`.

Hint:

Use the `as` option to name the route. By convention, the name of any route that maps to the `index` action should be plural since it deals with multiple things.

Answer:

```
get "movies/filter/:filter" => "movies#index", as: :filtered_movies
```

2. Then in the `app/views/layouts/_header.html.erb` partial, first change the "All Movies" link to "Released" just to be more consistent with the other links. Then use the route helper method to add "Upcoming" and "Recent" links.

Hint:

The name of the helper method is `filtered_movies_path`. You need to pass it a value to fill in the `:filter` route parameter.

Answer:

```
<li>
  <%= link_to "Released", movies_path %>
</li>
<li>
  <%= link_to 'Upcoming', filtered_movies_path(:upcoming) %>
</li>
<li>
  <%= link_to 'Recent', filtered_movies_path(:recent) %>
</li>
```

3. Finally, verify the links works as you'd expect.

Solution

The full solution for this exercise is in the `scopes-2` directory of the code bundle.

Bonus Round

List Hits and Flops

If you declared `hits` and `flops` scopes in the previous bonus exercise, then in the header add links that correspond to the following URLs:

- http://localhost:3000/movies/filter/hits
- http://localhost:3000/movies/filter/flops

1. First, in the `index` action add `case` clauses to handle the case when the `:filter` parameter is either `hits` or `flops`.

   Answer:

   ```
   def index
     case params[:filter]
     when "upcoming"
       @movies = Movie.upcoming
     when "recent"
       @movies = Movie.recent
     when "hits"
       @movies = Movie.hits
     when "flops"
       @movies = Movie.flops
     else
       @movies = Movie.released
     end
   end
   ```

2. Then add "Hits" and "Flops" links in the header.

   Answer:

   ```
   <li>
     <%= link_to "Hits", filtered_movies_path(:hits) %>
   </li>
   <li>
     <%= link_to "Flops", filtered_movies_path(:flops) %>
   </li>
   ```

3. Finally, verify the links work as you'd expect!.

Highlight the Active Link

From the user's perspective, it's currently not clear which scope is being used to list the movies. As a finishing touch, when the user clicks a link in the header it would be useful to highlight that link as being the "active" link. That way the user knows they're currently viewing the upcoming movies, for example.

To help get you started, we already have an `active` CSS class that underlines the active link. To see what it looks like, add the `active` class to the "Upcoming" link, like so:

```
<%= link_to "Upcoming", filtered_movies_path(:upcoming), class: "active" %>
```

Then reload, and the "Upcoming" link should be underlined.

Now for the challenge: You only want to highlight the "Upcoming" link if the user is currently on the page with the URL `movies/filter/upcoming`. More generally, you want to highlight whichever link matches the current URL. To determine if the URL generated by a link is the same as the URL of the current page, you can use the built-in current_page? method. If you pass it the URL of a link and `true` is returned, then it's the "active" page. In that case, you need to add the `active` class to the link.

Encapsulate that view-level logic in a helper called `nav_link_to` (put it in `movies_helper.rb`) and change all the navigation links in the header (the `_header.html.erb` partial) to use that helper, like so:

```
<%= nav_link_to "Released", movies_path %>
<%= nav_link_to "Upcoming", filtered_movies_path(:upcoming) %>
<%= nav_link_to "Recent", filtered_movies_path(:recent) %>
<%= nav_link_to "Hits", filtered_movies_path(:hits) %>
<%= nav_link_to "Flops", filtered_movies_path(:flops) %>
```

Hint:

If the current page matches the URL of the link, generate a link that has a `class` attribute whose value is `active`. Otherwise, generate a link without that class.

Answer:

```
def nav_link_to(text, url)
  if current_page?(url)
    link_to(text, url, class: "active")
  else
    link_to(text, url)
  end
end
```

Make It More Dynamic, Carefully!

Using a `case` statement to decide which scope to run is a perfectly valid solution. Sure, you have to add a clause every time you come up with a new scope, but in reality defining new scopes will be fairly infrequent. So maintaining the `case` statement isn't a big deal, and it has the benefit of being straightforward and explicit.

But suppose you wanted a more dynamic solution that could handle new scopes without any changes to the `index` action. How might you do that? One solution that often comes to mind, usually after a web search, is to use the Ruby `send` method. The `send` method lets you invoke a method dynamically, by its name. For example, try this in a `console` session:

```
>> Movie.send("upcoming")

>> Movie.send("recent")
```

Pretty cool! Calling `send("upcoming")` on the `Movie` class is the same as calling `Movie.upcoming`. Using `send` is simply a more dynamic way to call the method. So what you can do with that kind of power?

Assuming that the name of the scope used in the URL is the same as the scope method in the `Movie` class, you might be *tempted* to do something like this in your `index` action:

```
def index
  if params[:filter]
    @movies = Movie.send(params[:filter])
  else
    @movies = Movie.released
  end
end
```

DO NOT DO THIS! This code is a major security vulnerability! It allows anyone on the big, bad Internet to run any method of your `Movie` class. For example, suppose you had a scope named `secret` that was only intended to be used internally. Surprise! Anyone can now run that scope using the URL http://localhost:3000/movies/filter/secret. Or worse yet, how about that `delete_all` method that's defined on all models? With a wee bit of guesswork, a malicious user could delete all the movies in your database!

As they say, with great power comes great responsibility. The responsible way to do this is to only invoke the `send` method for a white-listed set of scope names. Similar to how you define a private method to only allow specific parameters to be mass-assigned from forms, you need to define a private method that only allows specific scope names, like so:

```
def movies_filter
  if params[:filter].in? %w(upcoming recent hits flops)
    params[:filter]
  else
    :released
  end
end
```

Think of this method as vetting the `:filter` parameter. If the value of `params[:filter]` is in the array of acceptable scope names, then that value is returned. Otherwise, if any other name is included in the URL, the default `released` scope name is returned instead. (Yup, you'll have to update this list if you want to expose new scopes.)

Then in your `index` action, you need to make sure to call `movies_filter` and only send its result to `send`, like so:

```
def index
  @movies = Movie.send(movies_filter)
end
```

The lesson here is to be very, very careful with any sort of dynamic programming (meta-programming) that involves user-supplied data. And don't trust any code you find on the Internet. :-)

Wrap Up

In the last two sections we focused on scopes and routes and you now know quite a bit about them:

- Scopes dynamically define class-level methods.
- Scopes take two parameters: the first parameter is the name of the scope (a symbol) and the second parameter is the custom query code.
- The query code in a scope declaration must be wrapped with a lamdba (`-> { }`) to make it a callable Ruby object (a `Proc` object) so that the query code gets evaluated every time the scope is invoked.
- To pass arguments to a scope, specify the arguments as parameters to the lambda, for example `->(max)`. Just as a method parameter can have a default value, a lambda parameter can also have a default value, for example `->(max=3)`.
- Scopes can be chained together to make a single complex query out of several small queries.
- Scopes are lazily evaluated, meaning they don't immediately query the database. A good example of this is the use of `where` to modify the query object.
- Scopes can be called on `through` associations.
- Declare one route that handles multiple filters (scopes) by using a route parameter, such as `:filter`, to match the name of the filter in the URL.

It's important to point out that not all scopes need to be directly accessible from the web. Most Rails apps have scopes (custom queries) that are only used internally in service of some higher-level business logic.

Speaking of scopes, now might be a good time to scope out a snack in preparation for the next section on cheerful, affectionate, friendly URLs.

Dive Deeper

To learn more about routing arbitrary URLs to actions, refer to Section 3 in Rails Guide: Rails Routing from the Outside In.

Friendly URLs and Callbacks

Exercises

Objective

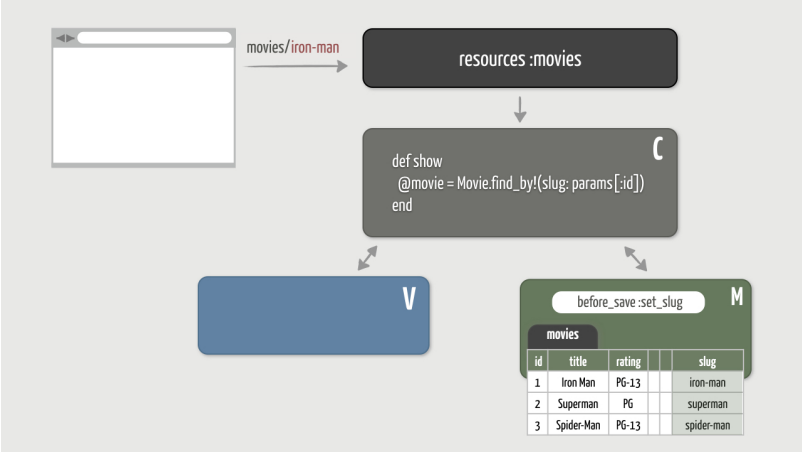
The folks in Hollywood decided to go all vain on us. It turns out those `movies/1`, `movies/2`, and `movies/3` URLs aren't good enough for them. Instead, they'd prefer vanity URLs such as `movies/iron-man`, `movies/avengers-endgame`, and `movies/x-men-the-last-stand`. So we'll roll out the red carpet for them.

Kidding aside, how your URLs look is actually worth serious consideration. After all, URLs are the user interface of the web. We browse to them, read them, bookmark them, post them to Twitter, email them to friends, and share them in other ways. An expressive URL can also help with search engine optimization (SEO).

To change the URLs to be a bit more Hollywood-friendly, we'll need to:

- Add a `slug` column to the `movies` database table
- Define a model callback to automatically generate and assign a slug to the `slug` attribute of the `Movie` model
- Define a `to_param` method in the `Movie` model to return the value of the `slug` attribute.
- Modify existing controller actions to find movies using the `slug` attribute

Here's what we want to do visually:



Let's get right to it!

1. Add a Slug Column

To support the URL `movies/iron-man`, for example, we'll need to be able to find a movie by its slug. That means we need to store each movie's slug in the database.

1. Start by generating a new migration that adds a `slug` column of type `string` to the `movies` table.

   Answer:

   ```
   rails g migration AddSlugToMovies slug
   ```

2. Don't forget to apply the migration.

   Answer:

   ```
   rails db:migrate
   ```

3. The slug for each movie must be unique, and since it will be derived from the `title` of the movie, the `title` must be unique. Add validations to the `Movie` model to ensure the title is unique.

   Answer:

   ```
   validates :title, presence: true, uniqueness: true
   ```

At this point the `movies` database table has a `slug` column, but none of the current movies have a slug. And if a new movie is created, it won't have a slug. This gives us a great opportunity to learn about model *callbacks*.

2. Define a Model Callback

Since the slug will be derived from the movie's title, there's no need to bother our admins with entering a slug on the movie form. Instead, we want to automatically generate and assign a slug when a movie is saved. To do that, we'll register a model *callback*.

1. Start by defining a private method named `set_slug` in the `Movie` model that assigns the parameterized version of the `title` to the `slug` attribute.

   Hint:

   When assigning a value to the `slug` attribute, you must use `self`. Otherwise, without using `self` the assignment would create a local variable called `slug`. Use the built-in parameterize method which converts any string into a URL-friendly format. It lowercases the string, replaces spaces with hyphens, and removes dots, colons, and ampersands!

   Answer:

   ```
   def set_slug
     self.slug = title.parameterize
   end
   ```

2. The `set_slug` method needs to be run *before* the movie is saved. To do that, add a `before_save` callback that calls the `set_slug` method.

   Hint:

Call the `before_save` method and pass it the *name* of the callback method as a symbol.

Answer:

```
class Movie < ApplicationRecord
  before_save :set_slug

  ...

private

  def set_slug
    self.slug = title.parameterize
  end
end
```

3. To see the callback in action, hop into a `console` session and find any movie. Initially it won't have a slug:

```
>> reload!

>> movie = Movie.find_by(title: "Avengers: Endgame")

>> movie.slug
=> nil
```

Then save the movie and verify that a slug was automatically assigned:

```
>> movie.save

>> movie.slug
=> "avengers-endgame"
```

Auto-magic... with no actual magic!

4. What about all the other existing movies? They don't have slugs either. That's easy enough to fix in the `console`. Just load each movie and re-save it.

Answer:

```
>> Movie.all.each { |m| m.save }
```

Here's a neat trick to verify that appropriate slugs were assigned to each movie. The `pluck` method queries for the specified attribute(s) without having to load up all the records:

```
>> Movie.pluck(:slug)
=> ["avengers-endgame",
 "captain-marvel",
 "black-panther",
 "avengers-infinity-war",
 "green-lantern",
 "fantastic-four",
 "iron-man",
 "superman",
 "spider-man",
 "batman",
 "catwoman",
 "wonder-woman"]
```

3. Define a `to_param` Method

With each movie now having a slug, we want movie-related URLs to include the slug rather than the id.

Anytime a model object needs to be converted into a URL parameter, the `to_param` method is called on the model object. All models that subclass `ApplicationRecord` inherit a default `to_param` method that simply returns the `id` of the record as a string. If your model doesn't explicitly define a `to_param` method, then the default implementation is used. But you can override the `to_param` method in any model to return a custom string that represents the URL parameter for that model. So all we need to do is return the movie's `slug`.

1. In the `Movie` model, define a `to_param` method that returns the value of the `slug` attribute.

Hint:

You don't have to use `self` when reading the `slug` attribute. You only have to use `self` when writing to an attribute.

Answer:

```
def to_param
  slug
end
```

2. Then jump back into the `console` and call the `to_param` on a movie object:

```
>> reload!

>> movie = Movie.find_by(title: "Avengers: Endgame")

>> movie.to_param
=> "avengers-endgame"
```

You should get a string formatted like `avengers-endgame` which is sometimes called a slug.

3. Now use the `movie_path` route helper method to generate the URL path for the movie, like so:

```
>> app.movie_path(movie)
=> "/movies/avengers-endgame"
```

Behind the scenes, the route helper method automatically called the `to_param` on the `movie` object to convert it to a slug. Then it used the result to generate the final URL path.

For this to work, it's important that we pass the full `movie` object to the `movie_path` method. Check out what happens if instead we pass the `id`:

```
>> app.movie_path(movie.id)
=> "/movies/1"
```

This time it doesn't call `to_param` because we explicitly passed the value of the `id` attribute.

This is why it's important to always pass the model *object* to route helper methods and let the `to_param` method decide how to represent that object as a URL parameter.

4. To bring that full circle, look in the `app/views/movies/index.html.erb` view template and recall that we're currently generating movie links like so:

   ```
   <%= link_to movie.title, movie %>
   ```

   Remember, that's just a shortcut for using the `movie_path` route helper like so:

   ```
   <%= link_to movie.title, movie_path(movie) %>
   ```

   Just as we did in the console, the `movie_path` route helper method is being called and passed a `movie` object, not just the movie's id. So the generated links include the movie's slug.

4. Fix the Controller Actions

Now that we have slugs and movie URLs are generated with those slugs, we next need to modify the affected controller actions to find movies using the provided slug rather than the id.

1. Reload the movie listing page and and if you hover over a movie link it should now have a proper slug. Great! Now click the "Avengers: Endgame" link, for example, and you should end up at http://localhost:3000/movies/avengers-endgame. The URL is correct, but you'll get this error:

   ```
   ActiveRecord::RecordNotFound in MoviesController#show

   Couldn't find Movie with 'id'=avengers-endgame
   ```

   And the following line of the `show` action is highlighted:

   ```
   @movie = Movie.find(params[:id])
   ```

   What happened was the slug `"avengers-endgame"` was stored in `params[:id]`. Then the `show` action called `find` which always queries the `id` field. And of course there is no movie that has an id of `"avengers-endgame"`.

2. To fix that, modify the `show` action to find a movie by its slug rather than its id.

   Hint:

   Use the `find_by!` method to query the `slug` field. Using `find_by!` (with the exclamation mark) will raise an exception if a movie with the specific slug could not be found. This mimics the behavior of `find` which also raises an exception. If an exception is raised in production, you'll get a 404 page which is exactly what we want if a movie with the given slug isn't found.

   Answer:

   ```
   def show
     @movie = Movie.find_by!(slug: params[:id])

     ...
   end
   ```

3. Now if you reload the page (you'll still on http://localhost:3000/movies/avengers-endgame) it should show the movie!

4. But if you try editing the movie (you need to be signed in as an admin), then you'll bump into a similar error. The `edit` action also needs to be modified to query the `slug` field. As does the `update` action. And you guessed it, the `destroy` action, too! You could certainly change each of these actions, but you should take this as an opportunity to clean up the duplication.

   Define a `private` method named `set_movie` that finds a movie by its slug and assigns the movie to an `@movie` instance variable.

   Answer:

   ```
   private

   def set_movie
     @movie = Movie.find_by!(slug: params[:id])
   end
   ```

   Then use a `before_action` to call that method *before* the `show`, `edit`, `update`, and `destroy` actions.

   Answer:

   ```
   class MoviesController < ApplicationController
     before_action :require_signin, except: [:index, :show]
     before_action :require_admin, except: [:index, :show]
     before_action :set_movie, only: [:show, :edit, :update, :destroy]

     # existing code
   end
   ```

Finally, remove the line of code in those actions that finds the movie.

5. At this point, you should be able run any of the actions in the `MoviesController`. However, the `ReviewsController` and the `FavoritesController` also need to be modified to find their parent movie by the slug rather than the id. Thankfully, those controllers already use a tidy `set_movie` method so you just need to change one line of code in each controller.

   Answer:

   ```
   # in reviews_controller.rb

   def set_movie
     @movie = Movie.find_by!(slug: params[:movie_id])
   end

   # in favorites_controller.rb

   def set_movie
     @movie = Movie.find_by!(slug: params[:movie_id])
   end
   ```

And there you go! All the movie links across the entire app now use the movie slug. Roll the credits...

Solution

The full solution for this exercise is in the `friendly-urls` directory of the code bundle.

Bonus Round

Practice With More Callbacks

To get more practice with model callbacks, you might consider using callback to format user-entered data before it's saved to the database. For example, if you added a `username` field to the `User` model in a previous bonus exercise, use a callback to convert it to a lowercase form.

Answer:

```
before_save :format_username

def format_username
  self.username = username.downcase
end
```

E-mail addresses should generally be lower-cased as well, so use another callback to convert them as well.

Once you learn about callbacks, it's tempting to go overboard with them. Callbacks are a good choice when they're focused on manipulating and keeping the model data consistent. Callbacks are the wrong choice when they're used to trigger behavior that may have unintended side effects or out-of-band processes, such as sending emails.

Friendly User URLs

Users are typically fairly friendly, so why not give them a friendly URL? If you did a previous bonus exercise, then you already have a `username` field that's both unique and URL-safe. Return it from the `to_param` method to generate URLs such as:

- http://localhost:3000/users/daisy
- http://localhost:3000/users/larry
- http://localhost:3000/users/lucy

Friendly Genre URLs

While you're on a roll, here's another friendly suggestion. If you did the previous bonus where you listed genres, then the next step would be to support URLs such as:

- http://localhost:3000/genres/action
- http://localhost:3000/genres/comedy
- http://localhost:3000/genres/drama
- http://localhost:3000/genres/romance
- http://localhost:3000/genres/sci-fi

As with movies, in the case of genres you can't just use the `name` field because it's not URL-safe. For example, you might have genres such as "Film Noir", "Variety Show", or "Children's Series". So you'll need to add a new field to support URL-friendly slugs.

Wrap Up

We used friendly URLs as an opportunity to demystify how URLs are generated and learn about model callbacks. So you now have a recipe for supporting friendly URLs and, more important, you have a general technique for running custom code at certain points in a model's lifecycle.

Next up we'll deploy this app! So dig out your favorite party hat and queue up some lively tunes because it will soon be time to celebrate your first launch!

Dive Deeper

- To learn more about callbacks, refer to Rails Guides: Active Record Callbacks. The full list of callback methods is available in the API documentation. They all work pretty much as you'd expect after learning the `before_save` method.
- If you need a "Swiss Army bulldozer" (their description) solution, you might consider checking out the FriendlyID gem. Among other advanced features, it maintains a slug history so old URLs don't break. But unless you need more features, we recommend sticking with a simple solution that you understand.

- For more examples of callbacks, if you're using the Devise gem on a project you'll find that it uses a number of callbacks. For example, it uses a callback to generate a confirmation token that's used to confirm a new user before they can sign in. Just search for `before_` in the source code.

Deployment

Exercises

Objective

It's time to deploy your app to a production server so you can share it with others!

Now, you may think you're off the hook because your company has an entire ops team dedicated to deploying apps to their servers. If that's the case, you probably owe them a round of coffee and donuts. But if you're like the rest of us, you'll be doing it all yourself. Either way, we think being able to deploy a Rails app is a fundamental skill for every Rails developer. And there's nothing quite as satisfying as seeing your app running on the 'net!

Deploying Rails apps has gotten easier over the years thanks in large part to cloud services such as Heroku. And that means we can focus our time and attention on delivering new application features and let Heroku worry about all the system-level details.

It's time to make this app real, so let's get right to it!

1. Prepare the App

Add the PostgreSQL Gem

When we run our application in the `development` environment (the default), the app talks to a local SQLite database. In production, however, Heroku will arrange things so that our app is connected to a PostgreSQL database. The PostgreSQL database is already provisioned on the Heroku servers, but we need to add the PostgreSQL adapter gem in order for our app to talk to the PostgreSQL database.

1. First, in the `Gemfile` find the line for the `sqlite3` gem. We only want that gem to be required when running in the `development` or `test` environments. To do that, put the `sqlite3` line inside of a `group` block, like so:

   ```
   group :development, :test do
     gem 'sqlite3', '~> 1.4'
   end
   ```

   Heroku will automatically ignore any gem dependencies that are in the `development` or `test` environments, so the `sqlite3` gem won't get installed or loaded when we deploy the app to Heroku's servers.

2. Then add the PostgreSQL gem (`pg`) in the `production` group like this:

   ```
   group :production do
     gem 'pg'
   end
   ```

3. Save the `Gemfile` and then run

   ```
   bundle config set --local without 'production'
   ```

   This configures Bundler for the local application to ignore any gem dependencies that are in the `production` environment.

   Then run

   ```
   bundle install
   ```

   Because the `pg` gem is in the `production` environment, it won't be installed on your development machine. Otherwise, you'd need to install PostgreSQL locally since the `pg` gem won't install if PostgreSQL isn't also installed.

   All your gem dependencies should already be installed, so running that command shouldn't install any new gems. So why run the command at all? Because running it updates the `Gemfile.lock` file. And this is the file Heroku will use to determine which gems to install. The upshot is when the application runs in the `production` environment on Heroku's servers, the `pg` gem will get installed and loaded by default.

4. At this point, you might guess that the next step would be to update the `database.yml` file with PostgreSQL connection settings in the `production` environment. Actually, Heroku takes care of that for us. When our app is deployed, Heroku auto-generates a `database.yml` with connection settings for Heroku's PostgreSQL database service. We don't have to mess around with the PostgreSQL database at all. It just works!

Add the Linux Platform to Gemfile.lock

By default, the generated `Gemfile.lock` file only supports the platform where the file was generated. For example, if you're developing on a Mac then the generated `Gemfile.lock` is only supported on a Mac. However, Heroku runs on Linux. So we need to add the Linux platform to the `Gemfile.lock` file.

To do that, run:

```
bundle lock --add-platform x86_64-linux
```

2. Create a Local Git Repository

Next we need a way to deploy (transfer) our application code to Heroku's servers which live in the cloud. And we do that using the Git distributed version control system.

One of the benefits of Git is that it's designed for distributed development. This makes it easy to share code and collaborate with other developers, but it also makes it easy to share code with a production server such as Heroku. In fact, Heroku requires that our application code be version controlled with Git. Here's how it works:

1. We create a *local Git repository* and add all the application code to it.
2. We also create a *remote Git repository* on the Heroku server.
3. Then, when we want to deploy the application to Heroku, we *push* the code from our local repository to the remote repository.

If you're worried about this turning into an exercise in system administration, fear not! With Git and the tools provided by Heroku, we'll have everything deployed in just a few simple commands.

Let's start by adding all the application code to a local Git repository...

1. First you'll need to install Git for your operating system if you don't already have a version installed. You can check by typing

   ```
   git --version
   ```

2. Then, if you haven't already set your user name and e-mail address, it's a good idea to do that now so Git knows your identity.

   To see if your user name and e-mail are already configured, run these two commands:

   ```
   git config user.name
   git config user.email
   ```

   If they don't return anything, set your user name and e-mail address like so:

   ```
   git config --global user.name "Your Name"
   git config --global user.email you@example.com
   ```

   Since this affects the global configuration, you only have to do it once. Now whenever you do anything with Git it will remember that you were in the driver's seat. If you need to change your identity later, just run the commands again.

3. Next, make sure you're in the `flix` project directory, or change directory into it.

   Mac OS X or **Linux**:

   ```
   cd ~/rails-studio/flix
   ```

   Windows:

   ```
   cd \rails-studio\flix
   ```

4. Then create a new *local Git repository* by typing:

   ```
   git init
   ```

   You should get output like the following which confirms that the local repository was created in the `.git` subdirectory:

   ```
   Initialized empty Git repository in /Users/mike/rails-studio/flix/.git/
   ```

5. The repository is empty, so the next step is to add all the files (and subdirectories) in the `flix` directory to the repository. To do that, type:

   ```
   git add .
   ```

   You won't see any output. Notice that because we used a dot (`.`), which represents the current working directory (`flix`), Git recursively adds all the files and subdirectories in one fell swoop. (Files that match the patterns in the `.gitignore` file won't be automatically added, however.)

6. It turns out that the `add` command doesn't really add the files to the repository. It actually adds them to a *staging area*. The staging area becomes important in more advanced scenarios, but don't worry about it for our purposes. So to really, truly add the files to the repository, the last step is to use the `commit` command:

   ```
   git commit -m "Ready to deploy v1.0"
   ```

   You'll see a long list of files that were committed. Whenever you commit a change, it's polite to include a short commit message. To add a message, we've used the `-m` flag followed by the message in double quotes. If you leave off the `-m`, Git will attempt to open a text editor where you can enter the commit message.

Now we have a local Git repository and our application code has been committed to that repository. But we haven't yet sent the code to the Heroku server. That's the next step...

3. Create a Remote Git Repository on Heroku

Next we need to create a *remote Git repository* on the Heroku server.

1. First, sign up for a Heroku account. You'll receive a confirmation email with instructions for activating your new account.

   As of November 2022, Heroku no longer has a free plan. However, they have a $5/month plan that's more than adequate for deploying apps as you're learning Rails. After activating your account, you'll then need to go to your account settings in the Heroku Dashboard, click the "Billing" tab, and click "Add Credit Card".

   The $5/mo plan is called the "Eco Dyno Plan". It gives you 1000 Eco dyno hours per month, and the hours are shared across all of your apps that use the Eco dyno type. So on that same page, click the "Subscribe to Eco" button. Any new apps you create will automatically share your Eco dyno hours.

2. After setting up your Heroku account, download and install the Heroku Command Line Interface (CLI) for your operating system.

3. Next, open a new command prompt just to make sure your environment includes the newly-installed Heroku CLI. Then change back into the `flix` project directory.

   Mac OS X or **Linux**:

   ```
   cd ~/rails-studio/flix
   ```

   Windows:

```
cd \rails-studio\flix
```

4. The Heroku CLI includes the `heroku` command-line tool for creating and managing Heroku apps. Start by logging in:

   ```
   heroku login
   ```

   You'll be prompted to enter any key to open your web browser where you can login and the CLI will then log you in automatically.

   Or you can stay in the CLI to enter your credentials by using the `-i` option, like so:

   ```
   heroku login -i
   ```

   You'll be prompted for your credentials:

   ```
   heroku: Enter your login credentials
   Email:
   Password:
   ```

   Use the email address and password you used when creating your Heroku account.

   Either way, you should end up seeing a reassuring message such as

   ```
   Logged in as daisy@example.com
   ```

5. Next, to create the app on Heroku, make sure you're still in your `flix` project directory and type:

   ```
   heroku create
   ```

   This command creates a new application on Heroku with a placeholder name, and also sets up the remote Git repository. Here's an example of what you should see:

   ```
   Creating app... done, ⬢ mighty-thicket-8732
   https://mighty-thicket-8732.herokuapp.com/ | https://git.heroku.com/mighty-thicket-8732.git
   ```

   We didn't specify the name of the application when using the `create` command, so Heroku automatically assigned a unique application name (`mighty-thicket-8732`). (We'll see how to rename the app a bit later.) A remote repo was also created and added as a remote endpoint of your local repo.

6. Finally, open the app either by browsing to the printed URL or by typing:

   ```
   heroku open
   ```

   That command automatically pops open your default browser and points it to the right web address.

   You should see a welcome message!

Cool—now we have a web address to share with our friends. But creating the app on Heroku doesn't automatically send our application code. That's the next step...

4. Deploy the App!

Now that we have both a local and remote Git repo, deploying our application code is trivial. All we need to do is push the code in our local repo up to the remote repo.

1. To deploy the application code to Heroku, type:

   ```
   git push heroku main
   ```

   This command pushes all the code committed in the local repo to the remote repo named `heroku`. We've specified that we want to push the code that's in the `main` branch. All Git repos start out with a `main` branch from which you can create other branches, but don't worry about branching right now.

   You'll see a bunch of output as Heroku chugs through the deployment process, and at the end you should get something like this:

   ```
   https://mighty-thicket-8732.herokuapp.com deployed to Heroku
   ```

2. We're almost there, but the production application is missing something crucial: the database tables! To create those tables, you need to apply the migrations to the PostgreSQL database that's running on the Heroku server. To do that, run:

   ```
   heroku run rails db:migrate
   ```

   Note that this is the same task we've been using to apply migrations locally (`rails db:migrate`), but prefixing any task with `heroku run` causes the task to run on the Heroku server.

3. Then to populate those tables with the example movies (and other seed data) contained in your `db/seeds.rb` file, run:

   ```
   heroku run rails db:seed
   ```

4. Now for the moment of truth! Reload the app in your browser using the production URL, and you should see the Flix app! 🎉

OK, now what? Well, since we know that apps are constantly undergoing change, let's see how we'd deploy the next iteration of the app...

5. Deploy a New Version

Suppose you share your freshly-deployed app with a few friends and one of them suggests a change. It's inevitable. As soon as you release an app, invariably a keen-eyed user will notice something that needs to be changed. Your first reaction might be a deep sigh, a palm to the forehead, or a big smile knowing that you've just enlisted your first human tester!

Either way, now that we have everything set up, it's super easy to release changes.

1. Just to keep it simple, suppose you need to add a link for contacting support. It might be an e-mail address, a Twitter link, or (if you're really daring) your personal phone number. Choose one and add it to the footer, for example.

2. Next, add and commit the changes to your local Git repo:

   ```
   git add .
   git commit -m "Add a support link"
   ```

3. Then deploy the changes by pushing the code to Heroku just like before:

   ```
   git push heroku main
   ```

4. There is no step 5!

And that's all there is to it! You can now deploy new versions of your app whenever you want with a couple commands. Some people call that being *agile*. We call it working smarter.

6. Create Your Admin User Account

At this point you may want to create an account for yourself. That's easy enough to do in the browser, but if you want to be an admin then you'll need to use a Rails console to set the `admin` flag on your account. And that console session needs to be connected to your *production* database. You've probably already guessed how to do that...

To start a Rails console session that's attached to the Heroku server, type

```
heroku run rails console
```

It's just like being in a local Rails console session, but just remember that any changes you make in this remote console session will affect your production database!

Troubleshooting

Hopefully it's smooth sailing from here, but if trouble strikes you may need to see what's going on behind the scenes of the Heroku servers. Here are a couple troubleshooting commands worth keeping in your back pocket:

- To list your apps, use

  ```
  heroku apps
  ```

- To check the status of your processes on Heroku, use

  ```
  heroku ps
  ```

- To look at the remote logs, use

  ```
  heroku logs
  ```

  Or you can stream the remote logs by using the `--tail` option, like so

  ```
  heroku logs --tail
  ```

Renaming or Removing the App

At some point, you might want to rename your app (to change the subdomain URL) or remove the app entirely. Here's how to do that:

1. Heroku automatically generated a unique, obscure subdomain name for your app. To list your apps, type:

   ```
   heroku apps
   ```

   You should see something like this, but your application name will most certainly be different:

   ```
   mighty-thicket-8732
   ```

   If you wanted share this app with your closest friends, you'd send them the link `https://mighty-thicket-8732.herokuapp.com/`.

   Honestly, unless you have good reason to do otherwise, it's probably best to stick with it. Only people you share this link with will be able to find the app, and it's unlikely that a malicious person would go looking for an app called `mighty-thicket-8732`. It's kinda like finding a needle in a thicket, and a mighty one at that.

   However, if you do want to change the app name, use the `rename` command like so:

   ```
   $ heroku rename flix
   ```

   You'll see output confirming that the app was renamed and the subdomain was updated:

   ```
   Renaming mighty-thicket-8732 to flix... done
   http://flix.herokuapp.com/ | https://git.heroku.com/flix.git
   Git remote heroku updated
   ```

   Remember though that Heroku application names must be unique, so it's possible that `flix` is already taken! It's also a lot easier to guess, so beware.

2. Alternatively, if you ever want to delete the app, simply use the `destroy` command and specify the name of the app to destroy, like so

   ```
   heroku destroy mighty-thicket-8732
   ```

   This is a destructive, permanent action, so you'll be prompted to re-enter the application name as confirmation that you really (really!) want to delete it.

3. Heroku also provides a convenient dashboard for administering and configuring your application.

Solution

The full solution for this exercise is in the `deployment` directory of the code bundle.

Bonus Round

Share Code on GitHub

If you want to share your application code and collaborate with other developers, you might consider also pushing your code up to GitHub. It's a popular social site for hosting and sharing Git repositories.

1. Sign up for a GitHub account if you don't have one already. The free account allows you to host open-source code.
2. After signing up, click on the link to create a repository. For the repository name, use "flix" and leave the "Initialize this repository with a README" option unchecked since your app already has that file.
3. Then use these commands to add GitHub as a remote repository and push your code up to your new GitHub repo:

   ```
   git remote add origin https://github.com/<username>/flix.git
   git push -u origin main
   ```

   Of course you'll need to replace `<username>` with your GitHub user name.
4. Refresh the GitHub repository page and you should see a list of your application files, a history of all the commits you've made, and a bunch of other options.
5. After making a change to the application, make sure to check your code into your local Git repo:

   ```
   git commit -am "Some useful message about the change"
   ```

   The `-am` flag adds all the changes to the repo and adds the commit message in one fell swoop.
6. Then push your changes to the GitHub remote repository:

   ```
   git push origin main
   ```
7. Finally, deploy the revised application code to Heroku:

   ```
   git push heroku main
   ```

Wrap Up

Congratulations on releasing your Rails app!

That's a huge accomplishment and you should totally give a couple fist pumps and sound off with a hearty W00T! And if it isn't too much to ask, e-mail us. We'd love to be one of the first people to visit your new Rails app!

Dive Deeper

Deployment

- See Getting Started on Heroku with Rails 7.x for more details on using Rails 7 on Heroku.
- To get answers to any Heroku-related questions, check out the Heroku DevCenter.

Using Git

In this section, we looked at Git primarily in the context of sharing code with a production server, in this case Heroku. But this barely scratches the surface of Git. More than just for deployment, Git is a very powerful version control system used by both solo developers and development teams throughout the entire lifecycle of app development. To learn more about Git, we recommend the online Git Immersion tutorial. For a more comprehensive look at Git (as well as a more searchable reference), check out the free online book Pro Git.

Active Storage File Uploads: Part 1

Exercises

Objective

As the application stands, when you create a new movie you have to use an existing image file that's in the `app/assets/images` directory. It's not a huge deal when you're developing the app. You can just drop a new image into that directory. But it's totally unfeasible when the app is running on a production server. To create a new movie you would need to add a new image to your local Git repo and then re-deploy the app to the server. Anyone volunteering to do that every time an admin user wants to create a movie?

Instead, when admin users create (or edit) a movie we'd like them to be able to select a movie image file on their local computer and upload it to the server. Uploading files in this way turns out to be fairly easy to do using Active Storage.

We'll start by using Active Storage to upload and store images on our local machine during development.

1. Remove the image_file_name Attribute

First up, since we're no longer going to tie movies to images in the `app/assets/images` directory, we'll remove the `image_file_name` column from the `movies` database table and also remove existing validations for that attribute.

1. Generate a migration file that removes the `image_file_name` column from the `movies` database table.

   Answer:

   ```
   rails g migration RemoveImageFileNameFromMovies image_file_name:string
   ```

2. Run that migration.

   Answer:

   ```
   rails db:migrate
   ```

3. Then remove the existing `format` validation for the `image_file_name` attribute in the `Movie` model.

That causes some unavoidable breakage. A few view templates still reference the defunct `image_file_name` attribute, so you'll see errors if you navigate around the app. We could temporarily fix those by, for example, always displaying a placeholder image as we did in the video. But we know where we're headed so instead of patching things up, let's just leave the templates as they are for now.

2. Install Active Storage

With that out of the way, we're ready to start using Active Storage to tie movies to uploaded images. Naturally the first step is to install Active Storage.

1. Run the Active Storage installer:

   ```
   rails active_storage:install
   ```

   This simply copies a migration file into our `db/migrate` directory.

2. Then run that migration:

   ```
   rails db:migrate
   ```

   This creates three database tables: `active_storage_blobs`, `active_storage_attachments`, and `active_storage_variant_records`.

3. Although you won't use these new tables directly, it's always empowering to have a cursory understanding of what goes on behind the scenes. So crack open the generated migration file and have a peek.

   The `active_storage_blobs` table is fairly straightforward. It stores all the details of uploaded files including their `filename`, `content_type`, `metadata`, and other such information. And the `key` is an encoded value that points to the file in the Active Storage service.

   The `active_storage_attachments` table is much more interesting. Focus in on the following two lines:

   ```
   t.references :record, null: false, polymorphic: true, index: false, type: foreign_key_type
   t.references :blob, null: false, type: foreign_key_type
   ```

   Each row of this table references a `blob` (a row in the `active_storage_blobs` table) and also references a `record`. In other words, each row of the `active_storage_attachments` table joins a blob and record.

   Now, from previous exercises we already know how join tables work. But what's different about *this* join table is it's a special type of join table: a *polymorphic* join table. You can tell because the first line uses the `polymorphic: true` option:

   ```
   t.references :record, null: false, polymorphic: true, index: false, type: foreign_key_type
   ```

   This means that the `record` it's referencing can be *any* ActiveRecord model. In our case it's going to be a `Movie` record. But it just as well could be a `User` record if we were to allow users to upload their avatar, for example.

   How does it know what kind of record it's referencing? To answer that, remember what happens when you use `t.references :record` without the `polymorphic: true` option. The migration would create a column named `record_id` that's a foreign key pointing to a record. But it wouldn't know what kind of record it's pointing to. So when you add the `polymorphic: true` option, the migration also creates a column named `record_type` which contains the name of an ActiveRecord model class, for example `Movie`. And with these two pieces of information—a foreign key and a class name—the row knows *exactly* which record it's referencing.

   Pretty clever, huh?

   Finally, the `active_storage_variant_records` table is used to store information about any variants of the original uploaded image. We'll learn more about variants in the next module.

3. Declare Attachment Associations

The database is ready to join any ActiveRecord model to a blob (an uploaded image), but we still need to declare any respective associations in our models. In particular, we want a movie to have one attachment image named `main_image`.

Add such a declaration to the `Movie` model.

Answer:

```
has_one_attached :main_image
```

4. Update the Movie Form

The `Movie` model can now accept a file attachment. Next we need to change the form so users can actually upload an image file when creating or editing a movie.

1. In the `app/views/movies/_form.html.erb` partial, find the `label` and `text_field` that currently reference the old `image_file_name` attribute. Change them to reference the new `main_image` attribute and swap out the `text_field` for a `file_field` which generates an input for selecting a file to upload.

   Answer:

   ```
   <%= f.label :main_image %>
   <%= f.file_field :main_image %>
   ```

2. For this to work, you need to add the `main_image` attribute to the list of permitted parameters which are defined in the `MoviesController`. While you're at it, remove `image_file_name` from the list.

   Answer:

   ```
   def movie_params
     params.require(:movie).
       permit(... :main_image ...)
   end
   ```

5. Use a Helper to Display Attached Images

Before uploading an image, let's first fix those broken view templates that are still trying to display an image using the old `image_file_name` attribute. If a movie has an attached main image, we want to display it. But if it doesn't have an image attached, we want to display the placeholder image. We're going to need to perform this same check in several templates, so we're wise to encapsulate it in a custom view helper.

1. In the `MoviesHelper` module, define a helper method named `main_image` that takes an `Movie` object and returns an `image_tag` either for the placeholder image or the main image if one is attached to the movie.

   Hint:

   You can call the `attached?` method on the `main_image` attribute to determine if a main image is indeed attached.

   Answer:

   ```
   def main_image(movie)
     if movie.main_image.attached?
       image_tag movie.main_image
     else
       image_tag "placeholder.png"
     end
   end
   ```

2. Then in the `app/views/movies/index.html.erb` template, find the line that references the old `image_file_name` attribute and change it to use the `main_image` helper method.

   Answer:

   ```
   <%= main_image(movie) %>
   ```

3. Similarly, change the `app/views/movies/show.html.erb` template to also use that helper method.

   Answer:

   ```
   <%= main_image(@movie) %>
   ```

4. Finally, change the `app/views/users/show.html.erb` template to use the helper to display the main image for the user's favorite movies.

   Answer:

   ```
   <%= link_to main_image(movie), movie %>
   ```

5. Now if you navigate around the app, the placeholder should be displayed everywhere since we haven't yet uploaded a movie image.

6. Upload a Movie Image File

Time for the moment of truth!

Go to the form for editing a movie and you should see a "Choose File" button near the bottom of the form. Click it and select an image file you want to attach to the movie. For example, you might want to find a different poster image for "Captain Marvel", download it to your desktop, and select it. Or you can select an image that's already in the `app/assets/images` directory.

Then click "Update Movie" and you should get redirected back to the movie's show page and see the uploaded image displayed!

7. Find the Stored File

By default in the `development` environment, Active Storage stores all uploaded images on your local disk in the `storage` directory.

If you now drill down into that directory, buried under a few subdirectories you'll find a file whose name is the same as the blob key. It'll be something obscure like `hizv9iu78dmll6rzal6kcfak02gf`. That's the file you just uploaded!

Solution

The full solution for this exercise is in the `active-storage-1` directory of the code bundle.

Wrap Up

We're now using Active Storage to store all uploaded movie images in the `storage` directory. With the basics in place, next up we'll add validations, resize the images, and upload images for all the remaining movies in a more developer-friendly way.

Active Storage File Uploads: Part 2

Exercises

Objective

Now that we've converted over to using Active Storage for uploading movie images, we're in a position to make the process more robust and flexible.

We want to do two things:

1. Add reasonable validations to restrict what can be uploaded
2. Attach images using code instead of using the form

1. Add Validations

We trust that admin users aren't malicious, and yet we'd be foolhardy if we didn't add validations to set limitations on the size and content type of uploaded files. What's acceptable? That's up to you. Just to put a reasonable stake in the ground, suppose we won't accept images over 1 MB and they must be either JPEG or PNG images.

You might think Active Storage would include built-in validations for this kind of thing. Unfortunately that's not the case. But not to worry: it's surprisingly easy to write custom validations.

Start by adding this to your `Movie` model:

```
validate :acceptable_image
```

This says "Hey, call the `acceptable_image` method when you're trying to validate a movie!"

Then down in the `private` section, define that method like so:

```
def acceptable_image
  errors.add(:main_image, "is too big")
end
```

Now when the movie is validated, a validation error will *always* be added to the `main_image` attribute with the blunt message "is too big".

So that's how you add validation errors when writing custom validations in this way.

But how do you know if the main image is too big? Well, you can read the value of the `main_image` attribute, which has an asssociated `blob`, which in turn has a `byte_size` attribute:

```
main_image.blob.byte_size
```

Likewise, you can read the `content_type` attribute of the `blob`:

```
main_image.blob.content_type
```

Armed with that knowledge, you're off to the races!

In the `acceptable_image` method, validate that the main image isn't over 1 MB and is either a JPEG or PNG image. A main image need not be attached, in which case you don't want to validate it. (Feel free to set your own reasonable limitations on the main image.)

Answer:

```
def acceptable_image
  return unless main_image.attached?

  unless main_image.blob.byte_size <= 1.megabyte
    errors.add(:main_image, "is too big")
  end

  acceptable_types = ["image/jpeg", "image/png"]
  unless acceptable_types.include?(main_image.blob.content_type)
    errors.add(:main_image, "must be a JPEG or PNG")
  end
end
```

Don't forget to test the validations by trying to upload a big PDF or animated GIF, for example.

2. Attach Images Using Code

Now that we have validations, we're in good shape to upload images for all the movies. Unless you've already done that using the form, you'll still see the default placeholder image for all the movies that don't have an image attached.

Using the form works and you could certainly take that route to upload images for the remaining movies. But sometimes it's convenient to be able to upload files programmatically using code.

There's no better place to try that out than in a Rails `console` session...

1. Start by finding a movie:

```
>> movie = Movie.find_by(title: "Avengers: Endgame")
```

2. Then let's suppose we want to upload the `avengers-end-game.png` image file that's currently in the `app/assets/images` directory. First you need to open that file like so:

   ```
   >> file = File.open(Rails.root.join("app/assets/images/avengers-end-game.png"))
   ```

   We're using this image because it's readily available, but you could just as well use an image on your Desktop. Just make sure to pass an absolute filename to the `File.open` method.

3. Then to upload that file and attach it to the movie, you call the `attach` method on the `main_image` attribute, like so:

   ```
   >> movie.main_image.attach(io: file, filename: "avengers-end-game.png")
   ```

   You must specify both an I/O object, in this case a file, and a filename.

   When this code is executed, a copy of the `app/assets/images/avengers-end-game.png` file is uploaded into the `storage` directory.

   And if you browse to the movie listing page or the show page for that movie, you should see the uploaded file. No form required!

4. That's handy! Where else might you want to upload images this way? Well, your `db/seeds.rb` file currently creates movies to "seed" the database so the application starts out with a collection of example movies. It would be convenient to also upload images for all those movies as part of the seeding process. And since you already know how to attach an image to a single movie using code, it's straightforward from there to attach images to all the movies created by the `seeds.rb` file.

   Give it a go on your own, then check the answer to see how we did it.

   Answer:

   ```
   [
     ["Avengers: Endgame", "avengers-end-game.png"],
     ["Captain Marvel", "captain-marvel.png"],
     ["Black Panther", "black-panther.png"],
     ["Avengers: Infinity War", "avengers-infinity-war.png"],
     ["Green Lantern", "green-lantern.png"],
     ["Fantastic Four", "fantastic-four.png"],
     ["Iron Man", "ironman.png"],
     ["Superman", "superman.png"],
     ["Spider-Man", "spiderman.png"],
     ["Batman", "batman.png"],
     ["Catwoman", "catwoman.png"],
     ["Wonder Woman", "wonder-woman.png"]
   ].each do |movie_title, file_name|
     movie = Movie.find_by!(title: movie_title)
     file = File.open(Rails.root.join("app/assets/images/#{file_name}"))
     movie.main_image.attach(io: file, filename: file_name)
   end
   ```

   In that file, you'll also need to remove all references to the `image_file_name` attribute. Then if you were to run the seeds file, all the seeded movies would have images.

Solution

The full solution for this exercise is in the `active-storage-2` directory of the code bundle.

Wrap Up

In the last two sections we learned a lot about Active Storage:

- It's built right into Rails and installing it is a snap.
- You use `has_one_attached :main_image` in an ActiveRecord model to declare its association to an attached image named `main_image`, for example.
- To attach an image, you either use a form with a `file_field` bound to the `main_image` attribute or you call the `attach` method on the `main_image` attribute.
- All uploaded images are stored in the `storage` directory by default when the app is running in the `development` environment. Behind the scenes, Active Storages uses three database tables—`active_storage_attachments`, `active_storage_blobs`, and `active_storage_variant_records`—to store file information and join models to their attachments.
- Uploaded images can be resized and transformed in other ways by creating what Active Storage calls a *variant*. Variants are generated on demand and also stored in the `storage` directory.

Next we turn our attention to using Active Storage in the *production* environment where we want all uploaded files to be stored on Amazon S3. Before we can do that, first we need to encrypt our super-secret Amazon credentials. That's up next...

Dive Deeper

- To learn more about writing custom validations, refer to Rails Guides: Active Record Validations.
- As an alternative to writing custom validations for image uploads, check out the Active Storage Validations gem which includes a variety of common validations you might want to perform.

Encrypted Credentials

Exercises

Objective

To communicate with external APIs or services that require authentication, an application generally needs to use access keys, tokens, or passwords. Indeed, all of the cloud-based storage services supported by Active Storage require some set of credentials. So to use Amazon S3 in production to store uploaded files, we need to hand over our credentials.

Here's the rub: those credentials are super-secret! If they were revealed, someone could hijack our account and run up our storage bill. So how do we safely give a Rails application our super-secret credentials so it can use them to access our Amazon S3 account?

This is such a common need that Rails has a built-in, and secure, solution.

We'll let Rails encrypt our Amazon S3 credentials in a special file that gets deployed with the application. And when the application runs, it will decrypt that encrypted file and use our credentials to communicate with Amazon S3.

1. Create an AWS Account and an S3 Bucket

First you'll need to create and activate an Amazon Web Services account if you don't already have one. The free tier offers sufficient storage space for our educational purposes.

Access to S3 is governed by a set of credentials: an access key id and a secret access key. The access key identifies your S3 account and the secret access key should be treated like a password. Your S3 credentials can be found on the Your Security Credentials section of the AWS console. For now just make sure you have these keys handy.

All files in S3 are stored in *buckets* which act basically like directories. Create an S3 bucket where your uploaded images will be stored. You can name the bucket anything you want, but the name must be unique across the entire Amazon S3 system. To avoid naming conflicts, you might want to use the name of the app that Heroku generated as the bucket name. For example, if your app is named `still-thunder` then you might name your bucket `still-thunder-images`.

2. Encrypt Your Access Keys

Now we're ready to encrypt our secret access keys in a file that the Rails app can decrypt when its running.

1. To do that, first run

   ```
   rails credentials:edit
   ```

   This will open a temporary file in your default editor. It uses the value of the `EDITOR` environment variable to determine your default editor. If nothing happens, you can set the `EDITOR` variable when running the command. For example, if VS Code is your preferred editor, then run

   ```
   EDITOR="code --wait" rails credentials:edit
   ```

   The `--wait` option prevents the command from exiting immediately. It waits until you're done editing the credentials and you've closed the file.

2. Once the temporary file opens you'll see all the credentials in plaintext. Uncomment the following section, and replace the values of both keys with your AWS access keys:

   ```
   aws:
     access_key_id: 123
     secret_access_key: 345
   ```

3. Here comes the semi-magical part: Save the temporary file and close it in your editor. Doing that causes the contents of the temporary file to be encrypted and written to the `config/credentials.yml.enc` file. If you open up that file, it's total gibberish. That's the result of good encryption.

   So `config/credentials.yml.enc` is obviously an important file. But on its own, this file is useless. To unlock the secrets held within that file, you need a master key.

   Where's the master key? You guessed it: in the aptly-named `config/master.key` file. This file was generated when you initially generated the Rails app. And if you look inside, you'll see that it also generated a master key that unlocks the secrets of your app!

It's worth repeating that **under no circumstances should you store the `config/master.key` file in your Git repo**. Otherwise anybody with access to that repo could unlock your secret credentials. Rails helps prevent that by generating a `.gitignore` file that ignores the `config/master.key` file.

You will, however, need to share the master key with your development team. Just be careful how you share it! Your best bet is to put it in a password manager such as 1Password that your team can access.

3. Accessing Credentials

How then does our Rails app access those credentials when it's running?

To answer that, hop back into a Rails `console` session and give this a whirl:

```
>> Rails.application.credentials.dig(:aws, :access_key_id)
```

The result should be the value of the `access_key_id` you typed into the temporary file.

To retrieve that credential, first the master key was used to unlock the secrets in the encrypted credentials file. Then we used the `dig` method to dig down into the credentials and extract the value that's nested under the given sequence of keys.

Using the same approach you can get your secret access key:

```
>> Rails.application.credentials.dig(:aws, :secret_access_key)
```

And if you look in the `config/storage.yml` file, that's exactly how the credentials are accessed to configure the Amazon S3 storage service:

```
amazon:
  service: S3
```

```
access_key_id: <%= Rails.application.credentials.dig(:aws, :access_key_id) %>
secret_access_key: <%= Rails.application.credentials.dig(:aws, :secret_access_key) %>
```

Solution

The full solution for this exercise is in the `encrypted-credentials` directory of the code bundle.

Bonus Round

Print All Credentials

If you just want to print all the credentials, you can run

```
rails credentials:show
```

Generate a New Master Key

If for some reason you need to generate a new master key, run

```
rails secret
```

Get Help

To learn more about credentials, run

```
rails credentials:help
```

Wrap Up

The takeaway is there is one and only one way to add or edit credentials needed by an application, and that's by running

```
rails credentials:edit
```

In this way, *all* secret credentials for an application are stored in an encrypted format in the `config/credentials.yml.enc` file using the master key contained in the `config/master.key` file.

With our super-secret credentials safely tucked away in an encrypted file, we're ready to take the application into production.

Dive Deeper

- Detailed instructions on how to create an S3 bucket.

Uploading Files to Amazon S3

Exercises

Objective

Now we're ready to deploy all this to production! 🎉

Here's what we need to do:

1. change the storage service to Amazon S3
2. set the master key on Heroku
3. install the AWS gem
4. deploy!

Let's make it happen...

1. Change the Production Storage Service

In the production environment we need to use the storage service named `amazon`.

1. Start by uncommenting that service in the `config/storage.yml` file and setting your `region` and the name of the `bucket` where you want uploaded images to be stored:

   ```
   amazon:
     service: S3
     access_key_id: <%= Rails.application.credentials.dig(:aws, :access_key_id) %>
     secret_access_key: <%= Rails.application.credentials.dig(:aws, :secret_access_key) %>
     region: us-east-1
     bucket: your_own_bucket
   ```

2. Then we need to configure Active Storage to use the `amazon` storage service when the application is running in the production environment. To do that, in `config/environments/production.rb` switch the ActiveStorage service from `local` to `amazon`:

   ```
   config.active_storage.service = :amazon
   ```

2. Set the Master Key on Heroku

When the application runs in production on a Heroku server, the app needs to read your Amazon S3 credentials which are encrypted in the `config/credentials.yml.enc` file. And to decrypt that file, the app needs your master key.

But remember, the `master.key` file won't be in your local Git repo. So when you push changes from your local Git repo to the remote Git repo on Heroku, the `master.key` file won't go along for the ride. Instead, you have to set the master key in a Heroku application-level configuration variable named `RAILS_MASTER_KEY`.

To do that, use the `heroku` command like so:

```
heroku config:set RAILS_MASTER_KEY=12345678
```

(Yup, you need to replace `12345678` with your application's master key.)

When the app is deployed, Rails will first look for the master key in a `config/master.key` file. And when it doesn't find that file, Rails then looks for the master key in the `RAILS_MASTER_KEY` variable.

Forgot whether you set the key or not? No problem. You can review all the Heroku config variables you've set using

```
heroku config
```

3. Install the AWS Gem

Since we'll be storing movie images in an Amazon S3 bucket, we need to install the AWS gem. Drop this into your `Gemfile`:

```
gem "aws-sdk-s3"
```

And install it:

```
bundle install
```

4. Commit Changes and Deploy!

Now all you need to do is commit the changes to your local Git repo and push them to the remote Git repo on Heroku, which re-deploys the app.

1. Start by adding and committing the changes to your local Git repo:

   ```
   git add .
   git commit -m "Use Active Storage"
   ```

2. Then deploy the changes by pushing the code to Heroku:

   ```
   git push heroku main
   ```

   You might want to grab a beverage as Heroku churns through the deployment process. When you get back, you'll be refreshed and the app should have re-deployed.

3. Next you need to create the Active Storage database tables in the PostgreSQL database that's running on the Heroku server. The deployed code already has a migration file that creates those tables, so it's just a matter of applying that migration to the production database:

   ```
   heroku run rails db:migrate
   ```

4. Now if you reload the app in your browser using the production URL, you shouldn't see any errors... but all the movies will be using the placeholder image. No surprise.

   You should now be able to upload movie images to your production app and the image files will be automatically stored in S3!

   You might want to upload all the images in one fell swoop. To do that, first fire up a console session that's attached to the Heroku server:

   ```
   heroku run rails console
   ```

   Then paste in the image-upload code that you previously added to your `seeds.rb` file. You'll see the images get pushed up into the cloud and when you reload all the movies will have respectable images!

Bonus Round

Providing the Master Key to Other Servers

When you set a Heroku config variable, behind the scenes an environment variable is set. So if you're running on a non-Heroku server that you have privileged access to, then you can simply set the `RAILS_MASTER_KEY` environment variable.

Another way to provide the master key to a server is to ssh the `master.key` file to a privileged directory on the server (for example a `shared` directory), and then create a symbolic link to the `master.key` file when the app is deployed.

Mirrors

It's worth pointing out that in addition to the standard cloud-based services, the `storage.yml` also includes a commented service named `mirror`. This lets you specify a primary service (for example `amazon`) and also a collection of mirror services which will store backup copies of uploaded files. The mirror services are automatically kept in sync with the primary service. This is especially handy if you're migrating from one service to the other in production.

Just remember that to use the `mirror` service in production you need to change `config/environments/production.rb` file to use the `mirror` service.

Solution

The full solution for this exercise is in the `amazon-s3` directory of the code bundle.

Wrap Up

Now your admin users can create and edit movies as fast as Hollywood produces them! And the best part is they don't have to bother you to do it. 👍