Unit 4: Rails and Databases

[Overview](#h.ocl3rjzdnt04)

[Lesson 4.1: Intro to Databases](#h.cf00662p171w)

[Gem of the Day](#h.aur53d8hpoa8)

[What is a Relational Database?](#h.mx2fj2lb2fgd)

[Relationships](#h.g0wi05jl08ro)

[Simple Rails Example: Doggie Daycare](#h.ijxwaax1tptx)

[Setting Up the Tables](#h.380rw3idotg9)

[Setting Up the Relationships](#h.9f6b06bu0n71)

[Updating the Views](#h.6841t4t7ux8o)

[Collection Select](#h.gljy3ihjoouo)

[Planning for Database Design: MusicDB](#h.b0nzevfv8wly)

[Database Design](#h.owmmulxbo4ys)

[Set Up and Gems](#h.5l2xd15gizh7)

[Creating the Model](#h.mzp5guv4u7k3)

[Starter Generators](#h.uzrrhyuv3w52)

[Updating the Views](#h.ccwcrwavujpi)

[Make it Pretty](#h.v6e16qgaw6qh)

[Lesson 4.2: Data Validation, Scoping, and Rails Console](#h.zeof47yix258)

[Review Homework](#h.l8otuv7enb0b)

[Gem of the Day](#h.cgcwrgduhrc)

[What Is Data Validation?](#h.mulzi3r54fvy)

[Data Validation in Music DB](#h.1kg35onl7gp7)

[Rails Console](#h.d0sy6geddka5)

[ActiveRecord](#h.v2th9f5fafvh)

[Read](#h.vpz320m07ogj)

[Create](#h.welhs9lc7gki)

[Update](#h.hk7mb7kl89le)

[Delete](#h.v9gj57mpvyh6)

[Scopes](#h.j0bbh2p5b7u0)

[Lesson 4.3: Seeding and Populating Databases](#h.sw1f5g8kvabx)

[Gem of the Day](#h.zn00kjo0nyc)

[Seeding Our Database](#h.hdcdppm2sjsl)

[Populating Development Database](#h.kmdbog3qniqs)

[Lesson 4.4: Rails and Database Review Using Group Project](#h.w2hskcy9csu4)

[Gem of the Day](#h.r66a55o7yixj)

[Rails Review: Doggy Daycare](#h.hzlfkvmqmq8i)

[Planning](#h.l5l34q1u0mae)

[Implementing What We Already Know](#h.4drxkbylh7ov)

[Adding Photos With Paperclip](#h.fpjzb9pjnffw)

[Group Project Summary: Doggie Daycare](#h.ejd90vxt3qib)

# Overview

1. Rails
   1. Rails console, DB console
   2. More gems
2. Data
   1. Generating models
   2. Data validations
   3. What are the elements of an SQL database (tables, rows, columns/fields)
   4. Basic SQL commands and CRUD

# Lesson 4.1: Intro to Databases

## Gem of the Day

Roll call!!

Starter Generators <https://github.com/starterleague/starter_generators>

## What is a Relational Database?

A **relational database** organizes data into one or more **tables** (or "**relations**") of **rows** and **columns**, with a **unique key** for each row.

Generally, each entity type described in a database has its own table, the rows representing instances of that type of entity and the columns representing values attributed to that instance. Each table is named according to the data it contains, such as people or addresses. Let’s break that down into an example. (Draw this on board:)

***Dogs***

|  |  |  |  |
| --- | --- | --- | --- |
| ***ID*** | ***name*** | ***breed*** | ***age*** |
| *1* | *Harry* | *Jack Russell* | *2* |
| *2* | *Priscilla* | *Australian Shepherd* | *4* |
| *3* | *Nolte* | *Shih-tzu* | *6* |

In this example, the table is called Dogs which is also the name of the relation. The rows represent instances of the entity type of “Dogs”, similar to the way classes and object instances work in Ruby. The columns, like name and breed, represent values attributed to each instance of Dog. Columns can also be called fields.

Because each row in a table has its own **unique key**, or **primary key**, rows in a table can be **linked** to rows in other tables by storing the unique key of the row to which it should be linked (where such unique key is known as a "**foreign key**").

Let’s add another table to better illustrate this example (draw on board - add owners column to dogs):

***Dogs***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***ID*** | ***name*** | ***breed*** | ***age*** | ***owner\_id*** |
| *1* | *Harry* | *Jack Russell* | *2* | *1* |
| *2* | *Priscilla* | *Australian Shepherd* | *4* | *1* |
| *3* | *Nolte* | *Shih-tzu* | *6* | *2* |
| *4* | *Flambeaux* | *Boxer-Pittie* | *1* | *2* |
| *5* | *Joy* | *Lab* | *1* | *3* |

***Owners***

|  |  |  |
| --- | --- | --- |
| ***ID*** | ***name*** | ***phone*** |
| *1* | *Sia* | *504-405-1234* |
| *2* | *Courtney* | *504-555-5432* |
| *3* | *Jackie* | *512-333-9876* |

In this case, ID in Dogs is the unique or primary key for Dogs. ID in Owners is the primary key for Owners. When I add a relationship between the two, I reference the Owner\_ID inside the dog table. The owner\_id is a foreign key and tells me which owner is linked to that instance of a dog. So, you can see that Sia owns both Harry and Priscilla while Nolte is owned by Courtney. Who owns Joy?

This is a huge benefit over using one giant table or spreadsheet. To convert this simple database to one spreadsheet, we would need it to repeat so much data (show by adding extra columns).

Just to recap, here is some of the language we just used, which you may hear interchangeably:

* table, relation, also resource in Rails
* row, record, instance, object, also tuple in some languages
* column, attribute, field
* primary keys - used to define the relationships among the tables
* foreign key - a field in a relational table that matches the primary key column of another table. The foreign key is used to cross-reference tables

Virtually all relational database systems use [SQL](http://en.wikipedia.org/wiki/SQL) (Structured Query Language) as the language for querying and maintaining the database. We’ll talk more about SQL tomorrow.

## Relationships

The example on the board shows us a one-to-many relationship. One owner can have many dogs. In this simplified case, one dog cannot have many owners.

Relationships can take three logical forms: **one-to-one**, **one-to-many**, or **many-to-many**. Take a minute to think of an example database with one of these types of relationships, then everyone will share out in turn. Bonus points for one-to-one (emergency contact and then more info about that contact) and many-to-many (books and authors).

Most relational database designs resolve many-to-many relationships by creating an additional table that contains the PKs from both of the other entity tables. This is known as a **join table**. We probably won’t cover them in detail here, but you should definitely read up on them.

## Simple Rails Example: Doggie Daycare

### Setting Up the Tables

Let’s go ahead and implement our owner example from the board. You actually already know more than you think. If we had started our app with Owners instead of Dogs, how would we build that table?

rails g scaffold Owner name:string phone:string

It’s actually the same. The only problem is that because we already generated Dogs, we have nothing linking the two tables. They are essentially **data islands**. The way we link them is by adding owner\_id to Dogs and setting up the relationships in the Model part of the app. How would we add a column to Dogs?

rails g migration AddOwneridToDogs owner:references

It’s exactly like before except that instead of doing owner\_id:integer, we simply call the name of the other table and say it is a reference with owner:references. Rails will magically know to add the foreign key. Let’s take a look at our migrations to check them, then go ahead and run rake db:migrate. Verify your schema… oops, looks like we still have our old owners column in there. Let’s go ahead and delete it. Who remembers how to remove a column from a table?

rails g migration RemoveOwnerFromDogs owner:string

Check migration, then rake db:migrate.

Check your schema again, and commit your changes!

### Setting Up the Relationships

We’re not quite done yet. In our Models, we need to define what type of relationships exist between our tables. Do you think this is a one-to-one, one-to-many, or many-to-many?

For simplification, let’s stick with one-to-many. The simplest way to figure out how to define your model relationships is to ask the question, “Which relation has many of the other?” Owners have many dogs. Great, now we go into the owner.rb model and put:

has\_many :dogs

And we go into the dogs.rb model and put:

belongs\_to :owner

Be careful with your pluralization! The easiest way to remember is just to pretend it’s a sentence and use proper English.

### Updating the Views

Let’s run our server and play with our first database! Go to /owners and enter a few owners.

Awesome. Now let’s go to /dogs. Uh-oh. First, it’s still showing our non-existent owner string column, and there is no way to add the actual owner. Let’s start by updating our controller and form.

1. We just added a field to Dogs. What do we need to do first before Rails will recognize when we submit it? We need to add it to the white-listed parameters in the dogs controller (remove :owner, and add owner\_id).
2. Now let’s edit the form view for Dogs (edit the owner lines):  
   f.label :owner\_id  
   f.number\_field :owner\_id
3. Edit or create a new dog and put in an owner number. Once you go back to the index, you can see that it put in what looks like the address for the instance of the object Owner for the Dog. That’s kind of interesting, but not really. Let’s make it give us the owner’s name.
4. Edit the dog index and change dog.owner to dog.owner.name. Yes, you can string together the dot methods for calling the fields. (walk through it from left to right)

### Collection Select

Awesome! Now, it would be even better if our form allowed us to pick from a list of owners. We can do that with a collection select field instead of the number field.

Let’s edit the form again:  
f.collection\_select :owner\_id, Owner.all, :id, :name

Let’s break down the different arguments in that collection\_select statement:

* :owner\_id - Collection select requires us to explicitly state the name of the foreign key. This is the link between the current table (Dogs) and the table we want to grab the selections from (Owners)
* Owner.all - Grab a list of all the owners
* :id - Grab owners by their id’s (to match the owner\_id)
* :name - List the owners by their name

Now try editing and creating new dogs. If you want, you can update the show view as well for homework. Commit your changes!

## Planning for Database Design: MusicDB

### Database Design

That was fun, but we actually had to do more work than really needed because we didn’t properly plan our database/model in advance. Let’s build a new app from scratch with a bit more pre-planning.

Our new app is a music database that will have artists, songs, and genres. For the sake of simplifications, we will assume that each artist belongs to only one genre and each song belongs to only one artist (bummer, I know). Let’s draw out our database on the board - you might want to follow along on paper for your notes (get them to give me tables, fields, and relationships):

**Genres**

|  |
| --- |
| **name** |
| pop |
| country |

* Genre has many Artists

**Artists**

|  |  |
| --- | --- |
| **name** | **genre** |
| Madonna | pop |
| Garth Brooks | country |

* Artist belong to Genre
* Artist has many Songs

**Songs**

|  |  |
| --- | --- |
| **name** | **artist** |
| Material Girl | Madonna |
| Friends In Low Places | Garth Brooks |

* Song belongs to Artist

### Set Up and Gems

Create a new Rails application called “music\_db” - if you want to continue building out this app after today, I suggest creating it outside of your rails\_practice repo.

Let’s go ahead and add some new gems. Add thin, quiet assets, and starter\_generators:

# Better server than WEBrick

gem 'thin'

# Easily add Bootstrap themes and elements

gem 'starter\_generators'

group :development do

# Quiet those assets down!

gem 'quiet\_assets'

end

Bundle. Now let’s take a look at the starter\_generators [documentation](https://github.com/starterleague/starter_generators). It’s pretty light, but it at least tells us that we can get more info by running either of these commands:

$ rails g starter:resource  
$ rails g starter:style

We’re just using the style one, so run it and look over the output. Sounds awesome! However, it looks like we might want to generate our model first so that the navbar will automatically populate with it.

Commit your changes.

### Creating the Model

Based on our database design, how would we write our scaffolds?

rails g scaffold Genre name:string

rails g scaffold Artist genre:references name:string

rails g scaffold Song artist:references name:string

Double-check your migration files then migrate and check your schema.

Now, we need to set up the associations in the Model files. What should we put where?

Genre

* has\_many :artists

Artist

* belongs\_to :genre
* has\_many :songs

Song

* belongs\_to :artist

Save your changes then run rails server to check that /artists, /songs, and /genres all look right. Don’t add any records yet. Commit your changes.

### Starter Generators

Let’s re-run that help command to remind us how to use starter generators:

rails g starter:style

Cool, looks like we can select some cool styles from Bootswatch (just a note - not all of them work). Check out the different ones in your free time. For now, let’s use Flatly:

rails g starter:style flatly

Type “y” when it asks to overwrite application.html.erb (this is what we want it to do!).

Before we run our server, we need to set a root url or the new navbar will cause an error. Go ahead and set your root to the artist index - if you forgot what this is, run **rake routes**:

root 'artists#index'

Now run your server and take a look (!). Commit your changes.

### Updating the Views

Our music app is already way better than doggie daycare originally was, but we still need to fix a few things to make it more user-friendly:

* Change our index pages to show the names rather than the ids
* Change our forms to use collection\_select boxes
* Change our show views to show the selected name of the reference rather than the id

Classroom challenge: Pair-program to update the index, form, and show views to properly show names and collection boxes rather than IDs. Then add a few values to each of your tables.

* index and show:
  + change from artist.genre to artist.genre.name, and
  + song.artist to song.artist.name
* form:
  + f.collection\_select :genre\_id, Genre.all, :id, :name
  + f.collection\_select :artist\_id, Artist.all, :id, :name

Wouldn’t it be cool if on the show views, we could actually see all artists of a genre or all songs of an artist, AND in order by name? Let’s do that now by writing some of our own erb.

Show Artists by Genre in genre show file (we do):

<strong>Artists in <%= @genre.name.capitalize %> music:</strong>

<ul>

<% @genre.artists.order('name').each do |artist| %>

<li><%= artist.name %></li>

<% end %>

</ul>

Classroom challenge: Repeat to show all songs for an artist in the artist show view.

<p>

<strong>Genre:</strong>

<%= @artist.genre.name %>

</p>

<p>

<strong>Songs by <%= @artist.name.capitalize %>:</strong>

<ul>

<% @artist.songs.order('name').each do |song| %>

<li><%= song.name %></li>

<% end %>

</ul>

</p>

Tip: You can similarly order your index pages by whichever field you like.

### Make it Pretty

First, let’s give our body some room. When the navbar was put in, it overlapped the top part of the page, so add a top padding to your body. It’s already set at 50, so let’s increase it to 70px.

Classroom challenge: Update your pages to use the correct Bootstrap tags and classes to make your tables, buttons, and forms (look up on www.getbootstrap.com/css) look pretty. Feel free to continue pair-programming.

* Add table class to each table: <table class="table">
* Add button class to “new” link at bottom of each index: class: "btn btn-success"
* Add a class of ”form-group” to each label and input pair, then class of form-control and placeholder text on each input item like so:  
  f.text\_field :name, class: "form-control", placeholder: "Genre"
* Add button classes to the submit button like so: <%= f.submit class: "btn btn-info" %>

Ending roll call: who feels like a real programmer today? (even if you feel lost)

# Lesson 4.2: Data Validation, Scoping, and Rails Console

## Review Homework

Roll call, then have students share their database models for students, courses, sections. Also, vote on group project app.

## Gem of the Day

Awesome Print <https://github.com/michaeldv/awesome_print>

Annotate <https://github.com/ctran/annotate_models>

# 

Let’s go ahead and add these gems to our Music DB - add to your gemfile and then bundle:

# Enable prettier rails console printing

gem 'awesome\_print'

# Annotate our models with our schema

gem 'annotate', '~> 2.6.6'

Now, let’s actually add the annotation now so that we can see it when we work on our validation. First, let’s update the configuration for annotation - to do that we need to install the config file with this command:

rails g annotate:install

Notice the file name and location. Now we can change all the “befores” to “afters”. Save the file, then run “annotate”. Take a look at the files it changed. Commit your changes.

## What Is Data Validation?

In computer science, data validation is the process of ensuring that a program operates on clean, correct and useful data. Bad data could make your database difficult to use, make it less secure, or potentially even break it. In other words, we only want to save good data to our database.

For example, when a user registers, we want to make sure that they enter a valid email address and password. Or, when we ask for a phone number, we get actual numbers.

### Data Validation in Music DB

Open up Music DB. Now, go to app>models. This is where we define our data validations. Let’s start with genre:

validates :name, presence: true, length: {minimum: 3}, uniqueness: true

This statement says that for the field “name” in the genre table, it must be **present** (not nil nor blank), at least 3 characters in **length**, and it must be **unique** - no other genre names can be the same (to prevent duplicates). **Length** can also check a maximum, within an interval range (1..15), or even is an exact number.

Classroom challenge: Run your rails server, and try to add genres that are empty, less than three characters, and duplicates. When you’re done, go ahead and repeat for artist and songs, though think about which validation options make sense. Commit your changes.

* artist:   
   validates :name, presence: true, length: {minimum: 2}, uniqueness: true  
   validates :genre\_id, presence: true
* song:   
   validates :name, presence: true, length: {minimum: 3}  
   validates :artist\_id, presence: true

We can even combine validations for multiple fields at once like so:

validates :name, :description, :rating, presence: true

Some other cool validations include:

* **acceptance** - This method validates that a checkbox on the user interface was checked when a form was submitted. This is typically used when the user needs to agree to your application's terms of service, confirm reading some text, or any similar concept. This validation is very specific to web applications and this 'acceptance' does not need to be recorded anywhere in your database (if you don't have a field for it, the helper will just create a virtual attribute).

class Person < ActiveRecord::Base  
 validates :terms\_of\_service, acceptance: true  
end

* **if/unless (conditional)** - You can associate the :if and :unless options with a symbol corresponding to the name of a method that will get called right before validation happens. This is the most commonly used option.

class Order < ActiveRecord::Base  
 validates :card\_number, presence: true, if: :paid\_with\_card?   
  
 def paid\_with\_card?  
 payment\_type == "card"  
 end  
end

You can read more about validations and options here (also listed in homework): <http://edgeguides.rubyonrails.org/active_record_validations.html>

## Rails Console

Let’s take a second to learn about Rails console. Remember IRB? Rails console is essentially IRB but with your entire Rails app (classes and database) already loaded. Inside Rails console, we can access all **CRUD operations** - create, read, update, and delete (write on board).

Access it by typing “**rails console**” or “rails c” on your command line inside your Rails application folder. Exit by typing “exit”

### ActiveRecord

Have you noticed how all of our models inherit from something called ActiveRecord? Active Record is the M in MVC - the model - which is the layer of the system responsible for representing business data and logic. Essentially, it’s how Rails manages your database with Ruby on top of SQL. It basically reduces the need for you to use complex SQL. All of the commands we will run on the console are ActiveRecord methods.

### Read

Access your rails console now and run these commands:

* Genre.all
* ap Genre.all → So much better with awesome\_print! Feel free to preface things with ap.
* Genre.first
* Genre.last
* Genre.find(2)
* Song.find\_by(name: "Material Girl") → limits to one result
* Song.where(name: "Material Girl") → gives us all results (I made two of these - one by Madonna and one by Garth Brooks)
* Artist.where("name LIKE 'G%'") → gives us all artists that start with G.

It’s kind of ugly, but pretty easy to access based off of what we already know about Ruby. Also, if you look close, you will see the actual SQL commands print right before the returned data.

### Create

Now let’s try creating some records. Does anyone remember how we instantiated a new cup when we learned about classes in Ruby? x = Cup.new

Run through these commands in rails console:

* g = Genre.new
* g.name = “blues”
* g → notice that ID, created, and updated are all nil
* g.save → notice that they are now set
* An alternate method is to create everything at once:  
  Artist.create(name: "Michael Jackson", genre\_id: "1")
* ap Artist.last

Classroom challenge: Create at least 5 new records, mixed between genres, artists, and songs. What happens when you try to go against your validation? You can run variablename.errors.messages to see the exact reasons why a save failed.

### Update

Let’s try updating records. For example, Garth Brooks doesn’t really have a song called “Material Girl”. Let’s change that entry to something else.

* ap Artist.where("name LIKE 'G%'")
* ap Song.where(artist\_id: 2)
* song = Song.find(4)
* song.name = “The Thunder Rolls”
* song.save
* ap Song.all

Classroom challenge: Update at least 3 records (create new ones first if you like).

### Delete

Deleting is pretty straightforward:

* Genre.create(name: “popcorn”)
* ap Genre.last
* genre = Genre.find\_by(name: "popcorn")
* genre.destroy
* ap Genre.all

## Scopes

You may have run into some problems with songs. Multiple songs can have the same name, but not with the same artist. We can actually account for this using **scopes** in Rails. Let’s change our song validation to:

validates\_uniqueness\_of :name, :scope => :artist\_id

Scoping allows you to specify commonly-used queries which can be referenced as method calls on the association objects or models. To define a simple scope, we use the scope method inside the class, passing the query that we'd like to run when this scope is called:

|  |
| --- |
| class Article < ActiveRecord::Base  scope :published, -> { where(published: true) }  end |

Let’s create a recently created scope for our Artists:

# Creating a recent scope that gives items created within the last indicated minutes.

# The lambda will freshly run every time the scope is called.

# Must exit and re-enter Rails console each time this file is updated.

scope :recent, -> (minutes\_past=60) {where("created\_at > ?", minutes\_past.minutes.ago)}

scope :today, -> { where('DATE(created\_at) = ?', Date.today)}

Then jump into Rails console to test it out:

Artist.recent

Artist.recent(30)

Artist.today

Commit changes.

Ending roll call: What did you learn today?

# Lesson 4.3: Seeding and Populating Databases

Roll call.

Answer any homework questions.

## Gem of the Day

Show and discuss what they are:

* Faker <https://github.com/stympy/faker>
* Populator <https://github.com/ryanb/populator>

## Seeding Our Database

Rails has a 'seeds' feature that should be used for seeding a database with **initial** data. If you know you will always need the same data, for example a list of states, you can seed them so that whenever a collaborator starts working on your app or when you push to production, you will have that data pre-populated.

It's a really simple feature: just fill up db/seeds.rb with some Ruby code, and run rake db:seed:

|  |
| --- |
| 5.times do |i|  Product.create(name: "Product ##{i}", description: "A product.")  end |

Let’s do this for our Music DB app so that genres will automatically be populated any time we do a db setup. This is convenient if we have multiple collaborators but also for setting up production.

Open up db/seeds.rb. Who remembers how we created new content using Rails console and ActiveRecord? Add all your genres now:

Genre.create(name: "pop")

Genre.create(name: "country")

Genre.create(name: "hip hop")

Genre.create(name: "rock")

Genre.create(name: "jazz")

Genre.create(name: "blues")

Save your file.

Before we can seed our database, we need to drop it and then set it up again. You can do this in two separate commands:

rake db:drop

rake db:setup

Or, you can do it in one command with:

rake db:reset

To see all your rake command options, type rake -T. (do this)

Go ahead and drop your database and set it up again.

## Populating Development Database

Use faker and populator to add data to development database for Doggie Daycare.

Faker::Name - scope resolution operator. Name lives inside of Faker. It’s a courtesy code so that you don’t accidentally collide with the Faker classes or subclasses. If you know you won’t have any collisions, then you can write “include faker” (the module) and then you can just call without the Faker:: prefix

In gemfile, add faker and populator to development only. Bundle.

Don’t do gem install faker. That installs at a system level. This is bad because different projects might be written on different versions and could break. Also, if you push to production to Heroku, for example, the production environment would not know to use that gem.

In lib/tasks, create a new file called populate.rake, then inside create this empty rake task:

namespace :db do

desc "Clear the DB and fill with excellent sample data"

task :populate => :environment do

end

end

Save and run rake -T - notice that your task is already created!

Example? <https://github.com/siakaramalegos/tts_resources/blob/master/lib/tasks/populate.rake>

Homework: seed another database, design model for Group project, install ImageMagick

# Lesson 4.4: Rails and Database Review Using Group Project

Roll call.

Answer any homework questions.

## Gem of the Day

Paperclip <https://github.com/thoughtbot/paperclip>

## Rails Review: Doggy Daycare

### Planning

First, let’s draw out what we want to generate for our app. Basically, we want an app that helps a business manage it’s doggy daycare while also allowing customers to order pet products online. We might add more features as we go along.

Trello is a great tool for managing app development. We can dump in all of our customer’s feature requests into a board called “icebox”. When we start working on a feature, we can move it to “wip”. When we are done, we move it to “complete”. We could even create a staging board for the next feature. The idea is that sometimes a customer will have feature ideas that sound great, but priorities change over time.

Today, we are going to focus on generating a cleaner model as well as adding dog photos and updating views. We will save building the merchant and user models for when we work on those features. Let’s decide what we want to build today - (draw these tables and fields on the board)

|  |  |  |
| --- | --- | --- |
| Dogs | Owners | Breeds |
| dog name  last name of owner (can pull in through owner\_id)  owner (owner\_id)  breed  date of birth  vet name  vet phone  currently at daycare? (boolean)  photo (avatar, later today) | first name  last name  primary phone  secondary phone  street address  city  state  zip  emergency contact name  emergency contact number | breed |

What steps do we need to take in order to re-create our app (get them to provide this list)?

rails new

cd into it

git/github

gems, bundle

scaffolds and model - generate and relationships

seed Breeds

generate pages

set root url

starter generator theme and navbar

fix forms and views to show actual data, not ID’s

### Implementing What We Already Know

Let’s do a Rails and database review by creating a clean copy of doggy daycare from scratch.

1. In your TTS folder (not in rails\_practice), run rails new doggy\_daycare (or whatever animal type you like). Make sure it is not inside another git repo as we will want to push this to Heroku later on.
2. cd into doggy\_daycare
3. Initialize a git repo, make your initial commit, create a repo on github, and push.
4. Add our gems, bundle, and commit:

gem 'thin'

gem 'starter\_generators'

gem 'awesome\_print'

gem 'annotate'

gem 'stamp'

group :development do

gem 'quiet\_assets'

gem 'faker'

gem 'populator'

end

1. Generate our scaffolds and model, double-check the migrations, migrate, and commit:  
   rails g scaffold Owner first\_name:string last\_name:string primary\_phone:string secondary\_phone:string address:string city:string state:string zip:string emerg\_name:string emerg\_phone:string  
     
   rails g model Breed breed:string  
     
   rails g scaffold Dog name:string owner:references breed:references dob:date vet:string vet\_phone:string in\_daycare:boolean
2. Finish setting the remaining relationships in your app/model files (has\_many).
3. Seed breeds with valid breed names. Then run rake db:seed. Commit.
4. Generate two pages - index and about. rails g controller Pages index about
5. Set the root to the pages index.
6. Now let’s dump in our starter generator theme. Run rails g starter:style flatly, or use another theme you like from Bootswatch.
7. Try to run rails s. It will error out because we don’t have routes for Breeds, but that’s ok. Let’s just manually take that code out of the navbar (in application.html.erb). While we are there, let’s make the About link on the navbar active - replace the anchor html tag with an erb link\_to method to the about page.  
   <%= link\_to "About", pages\_about\_url %>
8. Run rails server to check your pages, then commit your changes.
9. Now, let’s fix our forms and views to show actual owners and breeds, not ID’s. First, add at least one owner. Have you noticed how we have first and last names separate for owners? Well, that makes collection\_selects have a tiny obstacle. We can make it easier by creating a first\_last method in the Owner model:  
    def last\_first  
    "#{last\_name}, #{first\_name}"  
    end  
     
   While we are there, let’s add the table class on the index tables.
   1. form  
      f.collection\_select :owner\_id, Owner.all, :id, :last\_first, {prompt: "Select an owner"}   
      f.collection\_select :breed\_id, Breed.all, :id, :breed, {prompt: "Select a breed"}
   2. index and show  
      dog.owner.last\_name, or @dog.owner.first\_name + ' ' + @dog.owner.last\_name  
      dog.breed.breed
10. Add a dog to make sure the form, show, and index pages work. Commit your changes.

We will add validation and make our pages look prettier in the homework.

## Git Branch

(feature branches - create a new branch for paperclip)

## Setting up Paperclip

Paperclip is an awesome gem that handles file uploads and manipulates images. First, you must have ImageMagick installed. See the Lesson 4.3 homework for more info.

Take a look at the Paperclip gem documentation. The readme file is actually quite detailed. It can be overwhelming, but when you take it step-by-step it’s not too bad.

1. First, we need to let Paperclip have access to ImageMagick. See [here](https://github.com/thoughtbot/paperclip#image-processor). First, run which convert. Grab that path and put it in config/environments/development.rb:  
    # Telling Paperclip gem where to find ImageMagick  
    Paperclip.options[:command\_path] = "/usr/local/bin/"
2. Add gem ‘paperclip’, bundle, commit.  
    gem "paperclip", "~> 4.2"
3. Now we can generate our Paperclip stuff. Paperclip conveniently comes with an automatic migration generator to add an avatar to a table. Read more [here](https://github.com/thoughtbot/paperclip#migrations). Run:  
    rails g paperclip dog avatar
4. Check the migration then run rake db:migrate. Add :avatar to the Dog controller whitelist.
5. Now, let’s add some validations to Dog for avatar for safety.  
    has\_attached\_file :avatar, :styles => { :medium => "300x300>", :thumb => "100x100>" }, :default\_url => "/images/:style/missing.png"  
    validates\_attachment\_content\_type :avatar, :content\_type => /\Aimage\/.\*\Z/
6. Now we can add the file upload to our dog form:  
    <div class="field">  
    <%= f.label :avatar %><br>  
    <%= f.file\_field :avatar %>  
    </div>
7. And to show:

<p>

<%= image\_tag @dog.avatar.url %>

<%= image\_tag @dog.avatar.url(:medium) %>

<%= image\_tag @dog.avatar.url(:thumb) %>

</p>

Run your rails server, upload a file for a dog, and test everything out. So cool! If everything worked, **commit your changes**. Work through any errors during office hours and over the weekend.