[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)

# 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)