\*All queries are done from the rails console, may differ slightly when done in rails

\*The database used is either SQLlight, or Postgres normally, but ruby’s version is also referred to as Active Record

\*Can test out all of these queries in the blogs2 project in Rails\_models, it uses the same models

\*see model\_basics\_valdiations\_and\_callbacks for more info about models and associations/relationships

**QUERIES FROM CONSOLE**

**Models worked with in the examples**

* User - first\_name, last\_name, email

Can own several blogs, and write several posts and messages

* Blog - name, description

Owned by one or more users, and can have several posts and messages

* Owner - user\_id, blog\_id

User and Blog have a many to many relationship, so the Owner is the intermediary table

* Post - title, content, user\_id, blog\_id

Post is owned by one user and one blog. Can have many messages

* Message - message, post\_id, user\_id

Messages are owned by one user and own post (and through that post, a blog)

**Personal Notes**

* capitalization matters, all the models are singular by themselves, meaning User.first works, but user.first would error out, this changes when it’s not first, User.first.blogs is correct
* when you retrieve a single row (ex: User.first), it’s an instance, but if you get more than one back as a result, it is delivered as an array of instances/objects (array of objects in rails at least)
* Pay attention to when you write them as plural or singular. You always write singular when it’s a query for a single model (ex: User.first, Blog.all). It gets more complicated when you query one table through another. It’s easiest to think about how you write their model relations in the models folder inside the app folder of your project. So, if I want to find what blogs are owned by the first user, I would do this: User.first.blogs the blogs is plural because one user can own multiple blogs (has\_many in the model file). Even if that user only has one blog, you still write it plural. If I were to write User.first.blog it would error out. Similarly, if I wanted to see all the posts for the first user, since one user can have multiple posts, I would type: User.first.posts. However, if I wanted to see which user owned the first post, I would type Post.first.user, this is because each post is only owned by one user (belongs\_to on the Post model file). Think about whether you wrote has\_many or belongs\_to when you’re thinking about whether it needs to be plural or singular

**Basic Queries**

* **User.first** - finds the first user
* **User.last** - finds the last user
* **User.find(1)** - finds the user with an id of 1. will error out in the console if there is no user with an id of 1
* **test = User.all -** creates a variable called test and puts all of the users into it
* **User.where(:first\_name => ‘Bradly’) -** Finds any user with the first name of ‘Bradly’
* **User.where( [1,3]) -** finds any users within the array given, so any user between and including the id of one and three
* **User.all.select(:first\_name, :id).order(“created\_at Desc”)** - The select grabs only specific columns, in this case the first\_name and primary id. The order is the same as order by in SQL, you specify which column to order by, and I put DESC so that they would go in descending order.

**Creating**

* **User.create(:first\_name => 'Jerrod', :last\_name => 'Quintana', :email => 'j@j.com') -** creates a new user, the format is slightly different in the rails project, it’s formatted as an associative array, basically the :first\_name would be changed to “first\_name” =>…. ex: {"name"=>"Lightbulb", "description"=>"used for lighting", "pricing"=>"5.00"}
* **User.create(:first\_name => 'Jerrod', :last\_name => 'Quintana', :email => 'j@j.com', :password => “foobar”, :password\_confirmation => “foobar”)** - example of when it has a password, see starting a project doc for further info
* **Post.create(user: User.first, blog: Blog.find(2), title: ‘title’, content: ‘content’) -** creating a new user, But since it is owned by a user and blog, you need to include those, but never reference the actual id or foreign key. For example: (user: User.first) is good, it puts the entire instance there, but (user\_id: 1) is bad. Also, notice that you don’t need to put the colon : mark before each column, like I did in the User.create, you can do it two ways. Way one: (:first\_name => ‘Jerrod’) or (first\_name: ‘Jerrod’). Either a colon first, followed by =>, or just a colon inbetween the column name and value.
* **Owner.create(user: User.first, blog: Blog.last) -** creating a row in the owners table, since it is just an intermediary table between users and blogs due to them having a many to many relationship, all I need to do is give it a user\_id and blog\_id. this is so that I can make the first user be one of the owners of the last blog. Another way of doing this is shown below
* **User.first.owners.create(blog: Blog.last) -** this is another way of doing the query above, making the first user be one of the owners of the last blog. Except this way, I first query owner rows that are related to the first user, or owner rows that have that the user\_id of the first User. Then I create a new one, which automatically puts the first user in the user\_id column, so all I have to do is specify the blog. Make sure to make ‘owners’ plural, since one user can have many blogs through owners (thus specified in the model file in the rails project).
* **3.times{ Post.create( user:User.first, blog:Blog.find(2), title:Faker::Book.title, content:Faker::Hipster.sentence) }** - creates 3 posts, with its user being the first user and its blog being the blog with an id of 2, and their titles and content being randomized from faker.
* **1.upto(4){ |i|**
* **1.upto(4){ |x| Post.create(user.User.third, blog:Blog.find(i), title:Faker::Book.title, content:Faker::Hipster.sentence) }**
* **}** - Goes with the two bullets above. Basically, it creates several posts for several blogs, with the 3rd user being the owner of all the posts.

**Updating**

* **User.update(1, :first\_name => ‘Jerrodd’, :last\_name => ‘quintana’) -** the most common way to update, and a single line statement as opposed to saving it. the format is **Table.update(ID, :attribute => new\_value)**, so in my example, the record in the User table with an id of 1 would be updated, and the attributes that were updated would be the first\_name and last\_name. You can update a single column, all of them, or any amount you want.
* **Post.second.update(content: ‘content’)** - updates the content column of the second post. When used in this fashion, you don’t need to specify the Id
* **user = User.find\_by(name: ‘Jerrod’)** followed by **user.name = ‘Dave’** and finally **user.save** - a simple way of updating a record, assigning it to a variable, changing one of its attributes/column values and saving it.
* **user = {1 => {“first\_name” => ‘David’}, 2 => {“first\_name” => ‘John’} }**  followed by **User.update(user.keys, user.values)** - A way of updating multiple records at once, you plug a hash into a variable. Each key in the hash is an Id number, and the value for each Id number is a hash itself with the key being a column and the value being the new value for that column. Then you type **Table.update(variable.keys, variable,values)**.
* **Post.update(2, user: User.last) -** updates the post with an Id of 2, and change the user who owns it to be the last user
* **Owner.find\_by(blog: 1).update(user: User.third)** - a roundabout way of changing the blog with an id of 1, to be owned by the third user. It targets the Owner table, finds all rows where the blog has an id of 1, and changes the user to be the third user.

**Relational Queries - Queries Through Multiple Models That Are Connected**

* **User.find(3).blogs** - Basically finds all of the blogs that are related to or owned by the user with an id of 3. Techinically, it first goes through the owners table, and find any rows that have a user\_id of 3, then grabs the blogs in that row. I can do this shortcut because in the user.rb file in models, I wrote “has\_many :blogs, through: :owners”. If I didn’t have this written, I’d have to do something like plug this into a variable **var = Owner.where(user:User.third).includes(:blog)**, and then access a blog using **var.first.blog**, you can’t even see all blogs through this method that I know of.You could plug the first method into a variable and only the blogs would be saved
* **User.third.posts** - Grabs all the posts that are owned by the third user.
* **post1 = Blog**.**first.posts.includes(:user)** - This grabs all the posts owned by the first blog, and includes the users who created those posts. Basically, if I just type **post1**, it would show all of the the posts. then, if I wanted to see the user related to a post, I would have have to target individual posts, for example: **post1.first.user**
* **1.upto(3){ |i| User.first.owners.create(blog: Blog.find(i)) }** - Makes the first three blogs (blogs with an id of 1, 2, and 3) be owned by the first user. More precisely creates 3 Owner rows, with the user\_id equal to the first user, and the blog\_id equal to the first 3 blogs.
* **User.second.owners.create(blog: Blog.find(4))** - Makes the blog with an id of 4 be owned by the second user.
* **for i in Blog.all**
* **User.third.owners.create(blog:Blog.find(i))**
* **end** - goes with the two lines above. Makes the third user own all of the blogs. However, since this is reliant on Blog.find( id), it would error if say, you had already deleted the first blog created. then the first blog would have an id of 2, but the first loop would still run Blog.find(1), which doesn’t exist, therefore error out, at least in the console, may not in the actual rails application.

**Deleting/Destroying**

* **Note:** The difference between **delete** and **destroy** is that destroy runs any callbacks on the model, while delete does not, therefore, it is better to use destroy. For example, in the model, if you have a before\_destory method for users that tells it to destroy any blogs, posts, or messages that are owned by said user, then that would be activated if I used User.last.destroy, but not with .delete
* **User**.**last.destroy** - deletes the last user. Can also be done with **User.last.delete**
* **User.where(email: ‘r@r.com’).destroy\_all** - destroys all users that can an email of ‘r@r.com’
* **User.destroy\_all(email: ‘r@r.com’)** - Same as the above bullet. You don’t need to use a where, if you put **Table.destroy\_all()**, then the condition is expected to be after the destroy\_all in parenthesis as shown above. If those parenthesis are empty, then the below bullet point would happen.
* **User.destroy\_all** - without any conditions before or after the destroy\_all, all records in that table will be destroyed, never do this unless you want to empty the table.

**Creating Multiple rows at the same time with the Faker Gem**

* **5.times{ User.create(first\_name:Faker::Name.first\_name, last\_name:Faker::Name.last\_name, email:Faker::Internet.email) }** - Creates 5 users with randomized first/last names, and emails from the faker gem
* **3.times{ Post.create( user:User.first, blog:Blog.find(2), title:Faker::Book.title, content:Faker::Hipster.sentence) }** - creates 3 posts, with its user being the first user and its blog being the blog with an id of 2, and their titles and content being randomized from faker.
* **1.upto(4){ |i|**
* **1.upto(4){ |x| Post.create(user.User.third, blog:Blog.find(i), title:Faker::Book.title, content:Faker::Hipster.sentence) }**
* **}** - Goes with the two bullets above. Basically, it creates several posts for several blogs, with the 3rd user being the owner of all the posts. More specifically, it goes something like this: for the first outer loop, it creates 4 posts for the blog with an id of 1, then the second outer loop creates 4 posts for the blog with an id of 2, etc.