Table of Contents

[USEFUL LINKS 1](#_Toc496379743)

[THINGS TO REMEMBER/KEEP IN MIND 1](#_Toc496379744)

[CREATING MODELS 2](#_Toc496379745)

[RELATIONSHIPS 3](#_Toc496379746)

[VALIDATIONS 5](#_Toc496379747)

# USEFUL LINKS

* For validations used on the models - <https://apidock.com/rails/v2.3.8/ActiveRecord/Validations/ClassMethods/validates_length_of>
* For more validations - <http://guides.rubyonrails.org/active_record_validations.html>
* Validations and callbacks - <http://guides.rubyonrails.org/v3.2.13/active_record_validations_callbacks.html>
* Validations and class methods - <http://api.rubyonrails.org/v5.1/classes/ActiveModel/Validations/ClassMethods.html>
* Model Association/relationship basics - <http://guides.rubyonrails.org/association_basics.html>

# THINGS TO REMEMBER/KEEP IN MIND

* When creating your models, any column that doesn’t have a data type specified will be considered a string type.
* When it comes to relationships, it is very important how you write them in your model files, ex: has\_many, belongs\_to, etc. Your rails app reads these and determins what works and what doesn’t. For example, if I have a user model that can have many comments, then in the rails console, I can query for all the comments of the first user with **User.first.comments**, but if I don’t have “has\_many :comments” in the user model file, then that won’t work, rails won’t know what you’re talking about.
* To go along with the bullet point above, how you write those relationships determines how you refer to them in your queries. for example, in the query shown in the bullet above, I write comments plural, because on user can have\_many comments. But, if one user could only have\_one, then the above query would error out, it would have to be **User.first.comment**, think about how you wrote the relationships in the model file. One last example, If I had a user model that had a many to many relationship with a blog model and created a model to serve as an intermediary table, owner, then both the user and blog would have a have\_many relationship, and a have\_many with owner. But the owner model would have belongs\_to for both. So, if I wanted to get a blogs that is owned by the first user, I would type: **User.first.owners.first.blog**, the owners is plural because one user can have many of them, but blog is singular, since at that point I’m going through the owner model, and it belongs\_to a blog, singular
* Last point to go with the two above, if I use the has\_many: through:, then it makes my queries much easier. For example, in the query at the end of the last bullet, I couldn’t get all the blogs owned by the first user without writing this in the User model, if I types **User.first.blogs**, it would error out, because rails doesn’t know how it is related to blogs. In order for that to work I would need to include both “has\_many :blogs, through: :owners” and “has\_many :owners” in the User model file

# CREATING MODELS

* You create your project with postgres with **rails new AppName --database=postgresql**, if you want to create it without postgres, stop at the name of your project. Type: **rails db:create** after that if you’re using postgres
* **CREATING A MODEL:** Now, you create a model with **rails g model ModelName** **Column:Type**, your model name should always be Capital and Singular. the column name will be saved exactly as it is typed, it is followed by a colon mark :, no space, and the type of column. Ex: **rails g model User first\_name last\_name age:integer description:text password\_digest:string**, this creates a model named User, with the columns first\_name, last\_name, age, description, and password (or password\_digest as it is referred to in the schema.rb). As shown below, the first and last\_name are strings, age is an integer column, description a text, and password\_digest a string.
* IMPORTANT NOTE: any column that doesn’t have its type specified will be a string. ex: **rails g model User first\_name age:integer**, the first\_name column is a string data type.
* IMPORTANT NOTE: To use a password properly, it needs to be created as “password\_digest”, as shown in the example above the important notes, and as a string.
* **CREATING WITH REFERENCES:** to create a model with a relationship, such as one to one, one to many, etc. when you create your model, you create the foreign key column as type “references”, make sure the column name is lowercase and singular. Only the model that has the foreign key needs that column. For example, if I wanted users to be able to create comments, it would be one to many relationship, so when creating the comment model, I type: **rails g model Comment comment:text user:references**, in the actual model, the column would be user\_id.
* **TYPES OF DATA TYPES FOR YOUR COLUMNS:** The basic types you would use for a column are - :primary\_key, :string, :text, :integer, :bigint, :float, :decimal, :numeric, :datetime, :time, :date, :binary, :boolean, and remember, any column that doesn’t have a specified data type will become a string.

# RELATIONSHIPS

* the different types of relationships are: belongs\_to, has\_one, has\_many, has\_many :through, has\_one :through, and has\_and\_belongs\_to\_many
* It is very important how you write these in your model file, it will be reflected in your queries due to how rails reads them. Also, based on a past experience, it seems like you can’t use underscores in the names of has\_many, belongs\_to, etc. Meaning, “has\_many :invite\_senders” errors out, but “has\_many :senders” works.
* **EXAMPLE:** (this can be seen in blogs2 in Rails\_model) If I had a User, Blog, Owner, Post, and Message model, and I wanted one User to be able to have/own many blogs, posts and messages, and one Blog to be able to have/own many users and posts, due to one blogs being owned by several users, the owner table would act as their intermediary table. Also, one Post belongs to a single blog and can have many messages. This is how the relationships would be written:

**Users model:**

has\_many :blogs, through: :owners

has\_many :owners

has\_many :posts

has\_many :messages

**Blog model:**

has\_many :users, through: :owners

has\_many :owners

has\_many :posts

has\_many :messages, through: :posts

**Owner model:**

belongs\_to :user

belongs\_to :blog

**Post model:**

belongs\_to :user

belongs\_to :blog

has\_many :messages

**Message model:**

belongs\_to :post

belongs\_to :user

* As noted in the things to remember section at the top, writing the has\_many, through: makes the queries much easier. Also, putting “has\_many :messages, through: :posts” in the blog model allows me to do things like **Blog.first.messages**, even though they aren’t directly related, they go through the post table first.
* **SECOND EXAMPLE:** In the belt\_review\_events app in full rails, I have a User, Event, and Roster model. A user can create many events, but an event can only be owned by a single user. However, one user can attend many events and likewise, one event can have many users attending it, rosters is their intermediary table. So, this creates the issue of, in the User model, it would have “has\_many :events” twice, even though one would also have “through :rosters”, so this is where “source:” comes into play, you have to refer to one of them differently so that rails will know exactly what you mean in your queries. This is what those three models look like

**User model:**

has\_many :events

has\_many :rosters

has\_many :attending\_events, through: :rosters, source: :event

**Roster model:**

belongs\_to :user

belongs\_to :event

**Event model:**

belongs\_to :user

has\_many :rosters

has\_many :attending\_users, through: :rosters, source: :user

* By looking at the through and source lines above, rails knows what you mean in your queries. So, going by what I have above, if I wanted to get all of the events created by the first user, I would type: **User.first.events**, but if I wanted to actually see what events the first user is going to/attending, I would type: **User.first.attending\_events**
* **BELONGS\_TO EXAMPLE:** It is important to note that the different type so relationships often use different options. For example, “belongs\_to” doesn’t support “through:” like has\_many does. For example, if I have a User model and an Invitation model, basically the User model has a many to many relationship with itself. It also has a many to many relationship with itself through network, so for the has\_many on the User model, I can use through:, but on the invitation and network model, which both use “belongs\_to”, I have to use something different, see below

**User Model:**

has\_many :networks

has\_many :associates, through: :networks

has\_many :invitations, foreign\_key: 'reciever\_id'

has\_many :senders, through: :invitations

**Invitation Model:**

belongs\_to :reciever, class\_name: 'User'

belongs\_to :sender, :class\_name => 'User'

**Network Model:**

belongs\_to :user

belongs\_to :associate, :class\_name => 'User'

* **IMPORTANT NOTES FOR ABOVE MODELS:** **FIRST POINT -** For the above example, notice first how I use “foreign\_key” in the User model. When I created the Invitation model, I created a sender\_id and receiver\_id, even though both refer to the User model, since I couldn’t have two user\_id columns, I just named them like this to be simpler to understand which is which. therefore, I need to put “foreign\_key: key\_name” in the User model so Rails knows which foreign key column I am referring to. Without it, if I typed: **User.first.invitations**, the rails console errors out and said that it couldn’t find a “user\_id” column in the Invitation Model. Using “foreign\_key” tells rails which column that has\_many aspect of User it is attached to or which column I am referring to. HOWEVER, this isn’t necessary when whatever I name the has\_many is the same name as the actual foreign\_key. Look at “has\_many :senders”, I don’t need to use the foreign\_key because that column in the Invitation model is actually called “sender\_id”. Since I named the has\_many the same as the foreign key, I don’t need to specify which foreign key I am referring to. **SECOND POINT** - Notice how I use “class\_name:” (either format works, colon before or after). I do this because the actual columns are named “sender\_id” and “receiver\_id”, so rails doesn’t know that they are supposed to be fore the User model, neither are named “user\_id”. This tells rails that those foreign keys are named differently but are actually “user\_id” or are for the User model. It’s basically the same as saying for the User model “has\_many :colleagues, through: :networks, source: :associates”, since the actual column is “associate\_id”. Both source and class\_name specify the foreign key column.

# VALIDATIONS

* Any validations or changes you make to models take effect when saved, but if you’re in the rails console then you need to exit and re-enter the console for the effects to take place.
* Typically google the validations you need, or see the links at the top of the page, especially more specific or unusual ones.
* Here is a list of your basic validations:

**:length**- validates the length of an attribute's value

:minimum, :maximum, :in, :is

**:numericality -**validates whether an attribute is a numeric value

**:** :odd, :even, and many others

**:presence -**validates that the specified attributes are not empty

**:uniqueness** - validates whether the value is unique in the corresponding database table. NOTE: always create a unique index in the database too.

**:confirmation** - use this when you have two text fields that should receive exactly the same content; assumes the second field name has "\_confirmation" appended

**validates\_associated** - use this when your model has associations with other models and they also need to be validated

**:acceptance** - validate whether a checkbox was checked when a form was submitted (usually for 'terms and conditions')

* You can chain validations together and they can be for more than column at the same time, you first type what validation it is, just validates, validates\_confirmation\_of, etc. followed by what columns it affects, and the validation at the end, assuming it isn’t at the beginning like validates\_confirmation\_of
* In the below example, I make first\_name, last\_name, and email all required upon creation and they all have to be at least 2 characters. For length, you could also use **{in: 2..20}** or **{is: 2}** for more specific cases. For the email, I make it not case sensitive, meaning it sees “Jerrod@yahoo.com” and “jerrod@yahoo.com” as the same thing, I also use format so that it matches “EMAIL\_REGEX”, which just makes sure that it’s in proper email format.

**validates :first\_name, :last\_name, :email, presence: true, length: {minimum: }**

**validates :email, uniqueness: {case\_sensitive: false}, format: {with: EMAIL\_REGEX}**

* **IMPORTANT NOTE:** If you’re using a secure password, you need to have “**has\_secure\_password**” written in your model file, it allows you to save a hashed password\_digest, allows you to use password\_confirmation, and allows you to use an authenticate method.
* In the example below, I make sure that the amount value is actually a numerical value.

**validates\_numericality\_of :amount, :on => :create**

CALLBACKS

* Callbacks are usually activated before or after an action takes place.
* A list of before\_callbacks that are useful:

1. before\_validation
2. before\_validation\_on\_create
3. after\_validation
4. after\_validation\_on\_create
5. before\_save
6. before\_create
7. after\_create
8. after\_save
9. before\_destroy

* For example: If I have a user who has\_many posts, I want to make sure I delete the posts with the user, otherwise some query in my app will wind up erroring out. So I write a callback (before\_destroy) that destroys all posts created by the user first, before the actual user gets destroyed. See below:

**Class User < ActiveRecord::Base**

**…..validations and relationships here……..**

**before\_destroy :delete\_posts**

**def delete\_posts**

**User.find(self.id).posts.destroy\_all**

**end**

**end**