skillcrush

ACTIVE RECORD cheatsheet

ACTIVE RECORD

Active Record is how you will tell your database what data it is working with. Then, Active Record will help you with all manner of database related activities such as creating new objects, finding existing objects, or even, deleting objects that gots to go!

Active Record is also a major underpinning of the Rails framework, so keep this cheatsheet handy!

For more info:

Active Record Basics

Active Record Migrations

Active Record Querying

SETTING UP YOUR DATABASE

rake db:create FIRST STEP

Once you have configured your development environment in your database.yml you can go ahead and create your databases. To do that simply run:

```
rake db:create
```

rake generate:migration NAME=name_of_migration SECOND STEP

In order to set up your database tables you will need to create a migration. You will use this same process to make changes to your database tables. Run the following command, but replace "create_people" with a name that describes what your migration is looking to accomplish:

```
rake generate:migration NAME=create_people
```

Edit the migration file THIRD STEP

In order to complete the migration, you will need to find the migration file and edit it directly. The migration file will vary depending on what you are looking to accomplish but will look something like this:

```
class CreatePeople < ActiveRecord::Migration
  def change
    create_table :people do |t|
      t.string :first_name
      t.string :last_name
      t.datetime :birthdate
    end
  end
end
```

rake db:migrate FOURTH STEP

Once you have finished editing your migration file, it's time to let the database know what's up by running the migration. To do that type:

```
rake db:migrate
```

rake console

Once your database tables have been created you can hop into the rake console and create some entries. To do access the rake console type:

```
rake console
```

COMMON METHODS & CODE SAMPLES

.new

Want to create a new instance of your class? Use the new method!

```
user = User.new
```

.create

Alternatively, you can use the create method. The create method takes an unlimited amount of parameters where each key is the name of an attribute you have given your class.

```
user = User.create(name: "David", occupation: "Code Artist")
```

Setting attributes

You can also set attributes one at a time like so:

```
user = User.new
user.name = "David"
user.occupation = "Code Artist"
```

.all

Want ALL the instances of your class? Use the all method:

```
# return a collection with all users
users = User.all
```

.first

Need to return the first instance of your class? Use the first method!

```
# return the first user
user = User.first
```

.find_by

You can also search for instances of your class by certain attributes. Be careful though! This will always return the first user that matches the parameters.

```
# return the first user named David
david = User.find_by(name: 'David')
```

.where

You can also search for resources using the where method. You can pass as many attributes as you would like to the method:

```
# find all users named David who are Code Artists
users = User.where(name: 'David', occupation: 'Code Artist')
```

.order

You can order the data by using the order method.

```
# find all users named David who are Code Artists and sort by
created_at in reverse chronological order
users = User.where(name: 'David', occupation: 'Code
Artist').order('created_at DESC')
```

.save

In order to update resource you first need to grab it using one of the many methods, such as find_by, you will then need to change the attribute and save the changes:

```
user = User.find_by(name: 'David')
user.name = 'Dave'
user.save
```

.update

Alternatively, you can update the resource in one step by passing the new attributes into the update method.

```
user = User.find_by(name: 'David')
user.update(name: 'Dave')
```

.destroy

Should you want to delete a resource, you can use the destroy method:

```
user = User.find_by(name: 'David')
user.destroy
```

MORE QUERY METHODS

One of the amazing things that Active Record does is make it super easy for you to run queries on your database.

Whereas in the past developers had to write their OWN SQL queries and pass data between their databases and web apps, Active Record takes care of all that for you and all you need to do is write Active Record methods which will in turn generate SQL queries for you.

FOR EXAMPLE

You might use the find method:

```
# Find the client with id 10.
client = Client.find(10)
```

Which is the equivalent of writing:

```
SELECT * FROM clients WHERE (clients.id = 10) LIMIT 1
```

But you don't even need to worry about that!

distinct

If you would like to only grab a single record per unique value in a certain field, you can use distinct:

```
Client.select(:name).distinct
```

group

To apply a GROUP BY clause to the SQL fired by the finder, you can specify the group method on the find. For example, if you want to find a collection of the dates orders were created on:

```
Order.select("date(created_at) as ordered_date, sum(price) as
total_price").group("date(created_at)")
```

having

SQL uses the HAVING clause to specify conditions on the GROUP BY fields. You can add the HAVING clause to the SQL fired by the Model.find by adding the :having option to the find.

```
Order.select("date(created_at) as ordered_date, sum(price) as
total_price").group("date(created_at)").having("sum(price) > ?", 100)
```

limit

You can use limit to specify the number of records to be retrieved, for example:

```
Client.limit(5)
```

offset

You can use offset to specify the number of records to skip before starting to return the records. The offset method is often used with the limit method, for example:

```
Client.limit(5).offset(30)
```

order

You can use the order method to order the results by a specific parameter:

```
Client.order(created_at: :desc)
```

reorder

The reorder method overrides the default scope order. For example:

```
class Article < ActiveRecord::Base
  has_many :comments, -> { order('posted_at DESC') }
end

Article.find(10).comments.reorder('name')
```

select

To select only a subset of fields from the result set, you can specify the subset via the select method. If you would like to only grab a single record per unique value in a certain field, you can use distinct:

```
Cient.select(:name).distinct
```