RoR - Active Record

Active Record

Ruby on Rails uses Active Record as its ORM (Object Relational Mapping) system, taking out the need for writing SQL commands by hand in the model. Active Record give the ability to:

* Represent models and their data
* Represent associations between models
* Represent inheritance hierarchies through related models
* Validate models before they get persited to the database
* Perform database operations in an object-oriented fashion

Rails provides a default way to write models and relations, meaning if following the convention creating Active Record models can be very quick.

Conventions:

* Pluralise database table names - database names will be pluralised name of the object they contain, e.g. posts
* Singular model class names - CamleCased model names in singular, e.g. Post
* Singular foreign keys - named as <singularised\_table\_name>\_id, e.g. post\_id
* Standard Primary key - all primary keys are simply a column named id

Active Record - Models

Creating a model in Active Record is done by subclassing the ActiveRecord::Base, which is the same as creating an SQL query mapped to the table of related name in the database. In the example below, a post model is mapped in Active Record.

*class Post < ActiveRecord::Base*

*end*

Default conventions can be overridden by assigning them in the model, such as the below example:

*class Post < ActiveRecord::Base*

*self.primary\_key = "product\_id"*

*self.table\_name = "PRODUCT”*

*end*

Data can be added to the columns of a model either during initialization via hash, or after, once data is added, it can be saved to the database using .save (.save! will raise errors if validation fails)

*p = Post.new(name: ‘post\_1’)*

*p.content = ‘im a post’*

*p.save*

Blocks can also be used for mass creation:

user = User.new do |u|

u.name = "David"

u.occupation = "Code Artist"

end

There are various ways to access data entries from models:

* User.all - returns all data entries for model
* User.first - returns first data entry
* User.find\_by(<attribute>: ‘<entry>’) - returns first entry matching hash selector
* User.where(<attribute>: ‘<entry’).order(‘<column> DESC’) - return all matching selector and sort

Updating can be done by extracting the model and updating via hash:

user = *User.find\_by(<attribute>: ‘<value>’)*

*user.update(<attribute>: ‘<value>’)*

or update bulk:

*User.update\_all ‘<attribute> = <value>’*

Delete uses item.destroy.

Active Record - Validations

Validations allow the state of the model to be checked before updating the database, this is important to ensure data integrity during creates and updates. Validates are put in the model class and they come in two forms. The first form is a standard validation of attribute data value helper written into Active Record, such as:

*validates : attribute, presence: true*

Helpers can accept multiple attributes, the ‘:on’ option to define when they are run and the ‘:message’ option to define what message should be added to the error list. The list of helpers is:

* acceptance - checks if box was checked when form was submitted, the acceptance attribute will be virtual and not added to the database, value for acceptance can be changed with {accept: ‘value’}
* confirmation - checks if

The second is a custom validation created in a class with a validate(<item>) method, placed in the concerns folder of the model. Custom validation classes can do logic checks on the new model data and return custom errors, as is Rails convention the class name of the validation class must match file name:

*validates\_with CustomValidation*

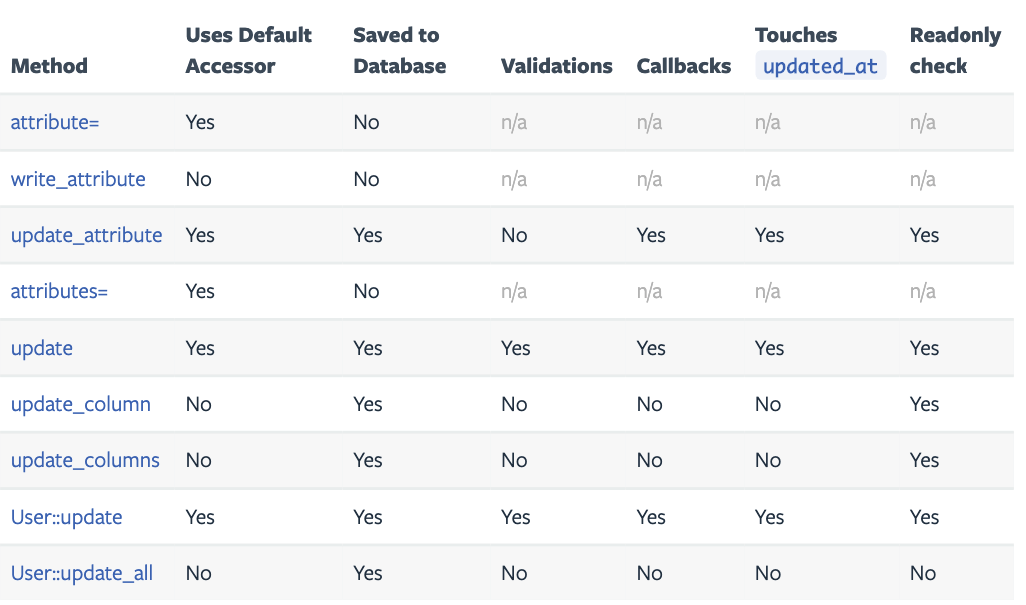
*def validate(item)*

*item.errors.add(:attribute, ‘length error’) if item.attribute.length > 10*

*end*

Using item<save(!), create(!), update(!)> will raise ActiveRecord:RecordInvalid errors, the none bang versions of the methods will return true or false depending on the validation success.

When updating records, most methods will perform validations:



Using update will perform validations, even if the attribute validated is not that being updated. For example updating a username, without touching the password will not pass validation since the password validation is still run:

user.update(username: 'new-username')

=> false

To prevent this error, either updates can be performed in the model itself (which won't run validations), or the singluar :update\_attribute method can be used:

user.update\_attribute(:username, 'new-username')

=> true

Validation notes:

* Some actions skip validations such as increment and update\_all
* .valid? can be used to check validitiy without error rasing
* error can be accessed with item.errors.messages (checking validity will generate error without rasing them, which can then be rendered)
* errors for a specific attribute can be accessed with errors[:attribute}

Testing Validations - RSpec

When a validation fails, the update or create method returns false (or throws error if banged) and its error messages are added to the model.errors.messages array.

There are a couple ways to test validations, firstly by checking if the model returns false or throws error:

expect(@user.update(username: nil)).to eq(false)

Checking whether the model is valid:

# test setup

expect(@user).to\_not be\_valid

Checking error messages if there are custom validation messages:

expect(@user.errors.messages[:password]).to eq ['custom password error message']

Confirmations

It is common to want to confirm attributes such as user passwords. To do this use a active record virtual confirmations by simply adding the 'confirmation: true' tag to the validation, then confirming the presence of the validation:

validates :password,

presence: true,

length: minimum: 8,

confirmation: true

validates :password\_confirmation,

presence: true

Uniqueness

While ActiveRecord has a built in check for uniqueness in validations, since it is not at database level, it does not protect against race conditions. An example race condition would be two requests to add user running concurrently in different workers. Each would check the database for the username, find it free, and add a username leading to a database with two similar usernames.

To prevent this issue, uniqueness checks should also be performed at database level, by adding the unique tag in the migration. For example:

class CreateUsers < ActiveRecord::Migration

def change

create\_table :users do |t|

t.string :email

...

end

add\_index :users, :email, unique: true

end

end

This will then cause a PG error when the above race condition occurs, so also add error catching such as the following to the create/update controller for the resource:

rescue ActiveRecord::RecordNotUnique

flash[:danger] = 'The username you entered is already taken!'

render action: 'new'

Scopes

Scopes are custom queries which can be added to a Rails model and called like a method to return an ActiveRecord:Relation object with models which match the scope. There are several reasons to use scopes:

* reduction is reused code (DRY), making a project easier to maintain.
* while simialr to a method, scopes are more precice since they do one thing

Define a scope using the scope method:

class Shirt < ActiveRecord::Base

scope :red, where(:color => 'red')

scope :dry\_clean\_only, joins(:washing\_instructions).where('washing\_instructions.dry\_clean\_only = ?', true)

end

The model can then be called with

shirt\_instance.red.dry\_clean\_only

Scopes can also be defined to accept arguments:

class Book

scope :with\_long\_title, ->(length = 20) { where("LENGTH(title) > ?", length) }

end