Ruby on Rails

Overview

Ruby on Rails is a framework for quickly creating webapps.

Rails applications are designed to separate concerns, so are several different parts each with their own responsibility:

* Controller
* View
* Model
* Routes
* Config
* Spec

How to Setup

Use the command ```rails new app-name``` to create a new folder of app name in the current directory with the framework automatically generated.

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 check 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.

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}

DB Migrations - Database Setup

Set up PSQL database with app as username and password (can do in rakefile):

create role myapp with createdb login password 'password1'

Create rails application folder setup to use PSQL instead of default mySQL:

rails new myapp --database=postgresql

The database requirements then need configuring in the rails database config:

/config/database.yml

The config uses the YAML serialization standard, it should look like this:

default: &default

adapter: postgresql

encoding: unicode

username: <%= ENV['POSTGRES\_USER'] %>

password: <%= ENV['POSTGRES\_PASSWORD'] %>

pool: 5

timeout: 5000

host: <%= ENV['POSTGRES\_HOST'] %>

development:

<<: \*default

database: <%= ENV['POSTGRES\_DB'] %>

test:

<<: \*default

database: <%= ENV['POSTGRES\_TEST\_DB'] %>

production:

<<: \*default

database: <%= ENV['POSTGRES\_DB'] %>

Rake setup can then be used to setup the database in the local PSQL database:

rake db:setup

DB Migrations - Generate Tables

Although ActiveRecord models ActiveRecord can be used to automatically generate tables for migration and testing. A migration is a subclass of ActiveRecord::Migration, which implements two methods:

* up - creates table and schema
* down - removes table

ActiveRecord then provides methods which perform common data definition tasks:

* create\_table
* change\_table
* drop\_table
* add\_column
* change\_column
* rename\_column
* remove\_column
* add\_index
* remove\_index

Migrations are stored in db/migrate, with one for each migration class, the naming convetion is a timestamp followed by the classname:

*YYYYMMDDHHMMSS\_<class-name>.rb*

Migrations are datestamped to allow for rails to run all migrations in order when there are multiple versions of one class, it allows allows for rollbacks to previous versions.

The class name should then relate directly to what the class does, such as creating a table called ‘users’ should then be called ‘create\_users’.

Template migration files can be made using:

rails generate migration <table-name> <column1>:<data-type1> <column1>:<data-type1> etc..

Migrations are also created when running generation tasks such as ‘model’ ‘resource’ and ‘scaffold’.

ActiveRecord doesn’t use SQL commands directly, instead it converts its own commands into them automatically. When creating tables, column data types are generated from ActiveRecord types. the ID column is automatically created as a primary key, other columns can be added like so:

*def self.up*

*create\_table :products do |t|*

*t.string :name, :null => false* 🡸 string type column called name, with no null

*t.text :description* 🡸 text type column called description

*t.timestamps* 🡸 automatic timestamps column

*end*

*end*

The ```self.down``` method of the migration should reverse whatever is done in the ```self.up``` method of the migration, this may simply require dropping a newly created table, or could be more complicated by reversing exactly what modifications were made to the schema. If a migration does something irreversible, such as destorying some data, the IrreversibleMigration error should be raised in the ```self.down``` method.

DB Migrations - Running

Running ```rake db:migrate``` will simply run all not previously run migrations ‘up’ methods, modifiying real tables and updating the schema file (using db:schema:dump). The schema file keeps a record of the database in Ruby DSL, seperating the schema from the database language.

It is important to note, once a migration has been filled in and ‘rake db:migrate’ run, its details will be added to the schema and it won’t be run again when migrate is run. Therefore it is important to make sure that the migration file is correct on first run, then if any changes are to be made to tables, a new migration is made. However, it is possible to roll back a migration using ```rake db:rollback STEP=2``` to rollback the last ‘step’ migrations and make it possbile to run a modifed version again.

DB Migrations - Update

Once a model is created and schema already in place, when it comes to updating a model and adding a new set of data, rails makes this particulary easy. Follow the process below:

* Add new dataset (column) to model
* use command:

```rails generate migration Add<Column>To<Model> <column-name>:<type>

DB Migrations - Referential Integrity

Active Record claims that intelligence belongs in the models of the application, not in the database. Therefore features such as triggers and foreign key constrains, which puch some intelligence back into the database, are not partically used. However, this can be argued against.

Referential Integrity can be ensured using:

* validates\_uniqueness\_of
* :dependent => destorys children is parent is destroyed

Starting Rails App

Rails apps have a built in server ‘puma’ and can be started on the local host using:

rails server --binding=127.0.0.1

Generator

Commands

rails routes - lists all routes in application

Nested Resources

To display nested items,

* ­­has\_many and belongs\_to relationships must have been set up
* params[:parent\_id] must be passed with the request
* parent and nested object must be passed into the form to allow for the correct directory

To show only specifiic routes to a resource in routing use:

``` resource :item, only: [:<route1>, <route2>]```

Secrets

In applications there are various secrets and keys which are required to access resources and ensure the security of the program. Normally secrets must be passed over some form of communication since secrets cannot be checked into version control in plain text. To make this easier rails has a method of encrypting secrets so only one ‘master.key’ is required to unlock the secrets.

The rails master key can be placed in ```config/master.key``` or stored as the RAILS\_MASTER\_KEY environment variable. The master key then unlocks the global ```config/credentials.yml.enc``` secrets file. Global secrets are those which should be available to all members of a team, such as the development variables:

EDITOR=vim rails credentials:edit

db:

name: development-db

EDITOR=vim rails credentials:edit --environment test

db:

name: test-db

EDITOR=vim rails credentials:edit --environment production

db:

name: production-db-1241f

aws:

access\_key\_id: 1f3649fe-ebbd-11e9-81b4-2a2ae2dbcce4

secret\_access\_key: 203060d3a5456fa6cd2da3c958001440

The relevent secrets can then be accessed in the application through the rails helper method, so long as the correct environment variable has been set:

RAILS\_ENV=development

Rails.application.credentials.db[:name]

= > development-db

RAILS\_ENV=production

Rails.application.credentials.db[:name]

= > production-db-1241f

Note: YAML files will not accept special charaters in strings unless the string is enclosed in quotes. Double quoted strings act the same as they do in JSON.

RSpec

RSpec replaces rails default testing suite (mini-test) with a more comprehensive testing suite, which allows for spec to be written describing how the application is susposed to behave.

Install to Rails by adding the ‘rpsec-rails’ gem to the testing and development sections of the gemfile. Then run the following to install and initalise:

bundle install

rails generate rspec:install

Boilerplate specs will then be auto generated when resources are created using rails generate commands.

JBuilder

Generate JSONs with ease.

General attribute names can be created using methods:

json.<attr-name>(<value>)

# => {"attr-name": value}

General structure names can be created using methods:

json.<struct-name> do

json.<attr-name1>(<value1>)

json.<attr-name2>(<value2>)

end

# => {"attr-name": {"attr-name1": value1, "attr-name2": value2 }}

Structure and attribute names can be dynamically created using the set method:

json.set! :author do

json.set! :name, 'David'

end

# => {"author": { "name": "David" }}

Hashes can be merged into data structures using the merge method:

hash = { ‘name’, ‘chris’ }

json.post do

json.title "Merge"

json.merge! hash

end

# => {"post": { "title": "Merge", ‘name’: ‘chris’ }}

Navbars

A navbar is a permanent section of a website which is generally displayed on everypage and allows a user to easily navigate sections of the website.

Since navbars are generally on everypage, they can be added to the view/layouts section and then rendered in the application html:

Navbar:

app/views/layouts/navbar.html.erb

Application:

# app/views/layouts/application.html.erb

…

<body>

<header>

<%= render 'layouts/navigation' %>

</header>

</body

Fonts

Fonts must be added to the asset pipeline in Rails for them to be assessible from browsers. To do this add a fonts folder to the app/assets dir, then add the following to the assets pipeline configuration in the Application class in ‘config/initalizers/assets.rb’:

Rails.application.config.assets.paths << Rails.root.join("app", "assets", "fonts")

Fonts can then be referenced using font-url(‘<url>’) in scss. For example:

@font-face {

font-family: 'RobotoTitle';

src: font-url('RobotoSlab-Medium.ttf');

src: font-url('RobotoSlab-Medium.ttf') format('truetype');

}

Forms

Rails provides a series of helper to generate html form elements, such as checkboxes, in erb files. Helpers:

Simple form: Creates standard html form

<%= form\_with(url: "/search", method: "get") do %>

<%= label\_tag(:q, "Search for:") %>

<%= text\_field\_tag(:q) %>

<%= submit\_tag("Search") %>

<% end %>

Tag helpers:

<%= radio\_button\_tag(:age, "child") %>

<%= label\_tag(:age\_child, "I am younger than 21") %>

<%= radio\_button\_tag(:age, "adult") %>

<%= label\_tag(:age\_adult, "I am over 21") %>

<%= text\_area\_tag(:message, "Hi, nice site", size: "24x6") %>

<%= password\_field\_tag(:password) %>

<%= hidden\_field\_tag(:parent\_id, "5") %>

<%= search\_field(:user, :name) %>

<%= telephone\_field(:user, :phone) %>

<%= date\_field(:user, :born\_on) %>

<%= datetime\_local\_field(:user, :graduation\_day) %>

<%= month\_field(:user, :birthday\_month) %>

<%= week\_field(:user, :birthday\_week) %>

<%= url\_field(:user, :homepage) %>

<%= email\_field(:user, :address) %>

<%= color\_field(:user, :favorite\_color) %>

<%= time\_field(:task, :started\_at) %>

<%= number\_field(:product, :price, in: 1.0..20.0, step: 0.5) %>

<%= range\_field(:product, :discount, in: 1..100) %>

Attaching models to helpers, the following form will action to the post model route:

form\_with model: @<model>, class: ‘<class>’ do |f|

Record identification can be performed if the resource being modified is a resource:

form\_with(model: @article, url: articles\_path)

form\_with(model: @article, url: article\_path(@article), method: "patch")

HTTP Requests

It is often required to authenticate user tokens and make external http requests on behalf of users from the rails application server. There are various different clients to make HTTP requests with, for this example Faraday is used.

To start the http library is required in the controller which needs to make the request:

require 'net/http'

Assuming the API is json based, then a separate method can be made for making the request including the following code and customised as required:

def http\_request(form\_data)

response = Faraday.post("https://accounts.spotify.com/api/token") do |request|

request.headers['Content-Type'] = 'application/x-www-form-urlencoded'

request.body = form\_data

end

JSON.parse(response.body)

end

Where the JSON data is returned as a ruby hash.