Ruby® on Rails

Notes for Professionals

Chapter 6: Rails Best Practices

Section 6.1: Fat Model, Skinny Controller

"Fat Model, Skinny Controller" refers to how the M and C parts of MVC ideally work together. N response-related logic should go in the model, ideally in a nice, testable method. Meanwhile, is simply a nice interface between the view and model.

In practice, this can require a range of different types of refactoring, but it all comes down to any logic that isn't about the response to the model (instead of the controller), not only have where possible but you've also made it possible to test your code outside of the context of

Let's look at a simple example. Say you have code like this:

```
def index
  @published_posts = Post.where('published_at <= ?', Time.now)
  @unpublished_posts = Post.where('published_at IS NULL OR published_at > ?', ...
end
```

You can change it to this:

```
def index
  @published_posts = Post.published
  @unpublished_posts = Post.unpublished
end
```

Then, you can move the logic to your post model, where it might look like this:

```
scope :published, ->(timestamp = Time.now) { where('published_at <= ?', ...
scope :unpublished, ->(timestamp = Time.now) { where('published_at IS ...
timestamp) }
```

Section 6.2: Domain Objects (No More Fat M

"Fat Model, Skinny Controller" is a very good first step, but it doesn't scale well

Let's think on the Single Responsibility of models. What is the single responsibili logic? Is it to hold non-response-related logic?

No. Its responsibility is to handle the persistence layer and its abstraction.

Business logic, as well as any non-response-related logic and non-persisten objects.

Domain objects are classes designed to have only one responsibility in th "Scream Their Architecture" for the problems they solve.

In practice, you should strive towards skinny models, skinny views and s solution shouldn't be influenced by the framework you're choosing.

For example

Let's say you're a marketplace which charges a fixed 15% commission to your customers via Stripe. If you charge a fixed 15% commission, that means that your commission changes depending on the order's amount because Stripe

Chapter 12: ActiveRecord Validations

Section 12.1: Validating length of an attribute

```
class Person < ApplicationRecord
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

The possible length constraint options are:

- **:minimum** - The attribute cannot have less than the specified length.
- **:maximum** - The attribute cannot have more than the specified length.
- **:in** (or **:within**) - The attribute length must be included in a given interval. The value for this option must be a range.
- **:is** - The attribute length must be equal to the given value.

Section 12.2: Validates format of an attribute

Validate that an attribute's value matches a regular expression using `format` and the with option.

```
class User < ApplicationRecord
  validates :name, format: { with: /\A\w{6,18}\z/ }
end
```

You can also define a constant and set its value to a regular expression and pass it to the with: option. This might be more convenient for really complex regular expressions

```
PHONE_REGEX = /\A[\d]{3}\)\d{3}-\d{4}\z/
validates :phone, format: { with: PHONE_REGEX }
```

The default error message is is invalid. This can be changed with the :message option.

```
validates :bio, format: { with: /\A\D+\z/, message: "Numbers are not allowed" }
```

The reverse also replies, and you can specify that a value should not match a regular expression with the without option

Section 12.3: Validating presence of an attribute

This helper validates that the specified attributes are not empty.

```
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: 'John').valid? # => true
Person.create(name: nil).valid? # => false
```

You can use the absence helper to validate that the specified attributes are absent. It uses the present? check for nil or empty values.

Chapter 42: Decorator pattern

Section 42.1: Decorating a Model using Draper

Draper automatically matches up models with their decorators by convention.

```
# app/decorators/user_decorator.rb
class UserDecorator < Draper::Decorator
  def full_name
    "#{object.first_name} #{object.last_name}"
  end

  def created_at
    Time.use_zone(h.current_user.timezone) do
      object.created_at.strftime('%A, %d %b %Y %l:%M %p')
    end
  end
end
```

Given a Draper variable containing an ActiveRecord object, you can access your decorator by calling #decorate on the #user, or by specifying the Draper class if you want to be specific.

```
<% user = @user.decorate %><!-- OR -->
<% user = UserDecorator.decorate(@user) %>
<h1><%= user.full_name.decorate(@user) %></h1>
<h3>joined: <%= user.created_at %></h3>
```

Section 42.2: Decorating a Model using SimpleDelegator

Most Rails developers start by modifying their model information within the template itself:

```
<h1><%= @user.first_name %> (@user.last_name )" %></h1>
<h3>joined: <%= @user.created_at.in_time_zone(current_user.timezone).strftime('%A, %d %b %Y %l:%M
%p') %></h3>
```

For models with a lot of data, this can quickly become cumbersome and lead to copy-pasting logic from one template to another.

This example uses SimpleDelegator from the stdlib.

All requests to a SimpleDelegator object are passed to the parent object by default. You can override any method with presentation logic, or you can add new methods that are specific to this view.

SimpleDelegator provides two methods: __setobj__ to set what object is being delegated to, and __getobj__ to get that object.

```
class UserDecorator < SimpleDelegator
  attr_reader :view
  def initialize(user, view)
    __setobj__ :user
    @view = view
  end

  # new methods can call methods on the parent implicitly
  def full_name
    "#{ first_name } #{ last_name }"
```

200+ pages
of professional hints and tricks

Contents

About

Chapter 1: Getting started with Ruby on Rails

Version	Release Date
5.1.5	2018-02-14
5.1.2	2017-06-26
5.0	2016-06-30
4.2	2014-12-19
4.1	2014-04-08
4.0	2013-06-25
3.2	2012-01-20
3.1	2011-08-31
3.0	2010-08-29
2.3	2009-03-16
2.0	2007-12-07
1.2	2007-01-19
1.1	2006-03-28
1.0	2005-12-13

Section 1.1: Creating a Ruby on Rails Application

This example assumes *Ruby* and *Ruby on Rails* have already been installed properly. If not, you can find how to do it in Section 1.4: Installing Rails.

Open up a command line or terminal. To generate a new rails application, use rails new command followed by the name of your application:

```
$ rails new my_app
```

If you want to create your Rails application with a specific Rails version then you can specify it at the time of generating the application. To do that, use `rails _version_ new` followed by the application name:

```
$ rails _4.2.0_ new my_app
```

This will create a Rails application called `MyApp` in a `my_app` directory and install the gem dependencies that are already mentioned in `Gemfile` using `bundle install`.

To switch to your newly created app's directory, use the `cd` command, which stands for `change directory`.

```
$ cd my_app
```

The `my_app` directory has a number of auto-generated files and folders that make up the structure of a Rails application. Following is a list of files and folders that are created by default:

File/Folder	Purpose
app/	Contains the controllers, models, views, helpers, mailers and assets for your application.
bin/	Contains the rails script that starts your app and can contain other scripts you use to setup, update, deploy or run your application.
config/	Configure your application's routes, database, and more.
config.ru	Rack configuration for Rack based servers used to start the application.
db/	Contains your current database schema, as well as the database migrations.
Gemfile	These files allow you to specify what gem dependencies are needed for your Rails application.
Gemfile.lock	These files are used by the Bundler gem.

lib/	Extended modules for your application.
log/	Application log files.
public/	The only folder seen by the world as-is. Contains static files and compiled assets.
Rakefile	This file locates and loads tasks that can be run from the command line. The task definitions are defined throughout the components of Rails.
README.md	This is a brief instruction manual for your application. You should edit this file to tell others what your application does, how to set it up etc
test/	Unit tests, fixtures, and other test apparatus.
temp/	Temporary files (like cache and pid files).
vendor/	A place for all third-party code. In a typical Rails application this includes vendored gems.

Now you need to create a database from your `database.yml` file:

Version ≥ 5.0

```
rake db:create
# OR
rails db:create
```

Version < 5.0

```
rake db:create
```

Now that we've created the database, we need to run migrations to set up the tables:

Version ≥ 5.0

```
rake db:migrate
# OR
rails db:migrate
```

Version < 5.0

```
rake db:migrate
```

To start the application, we need to fire up the server:

```
$ rails server
# OR
$ rails s
```

By default, rails will start the application at port 3000. To start the application with different port number, we need to fire up the server like,

```
$ rails s -p 3010
```

If you navigate to http://localhost:3000 in your browser, you will see a Rails welcome page, showing that your application is now running.

If it throws an error, there may be several possible problems:

- There is a problem with the `config/database.yml`
- You have dependencies in your `Gemfile` that have not been installed.
- You have pending migrations. Run `rails db:migrate`
- In case you move to the previous migration `rails db:rollback`

If that still throws an error, then you should check your `config/database.yml`

Section 1.2: Create a new Rails app with your choice of

database and including the RSpec Testing Tool

Rails uses `sqlite3` as the default database, but you can generate a new rails application with a database of your choice. Just add the `-d` option followed by the name of the database.

```
$ rails new MyApp -T -d postgresql
```

This is a (non-exhaustive) list of available database options:

- mysql
- oracle
- postgresql
- sqlite3
- frontbase
- ibm_db
- sqlserver
- jdbcmysql
- jdbcsqlite3
- jdbcpostgresql
- jdbc

The -T command indicate to skip the installation of minitest. To install an alternative test suite like RSpec, edit the Gemfile and add

```
group :development, :test do
  gem 'rspec-rails',
end
```

Then launch the following command from the console:

```
rails generate rspec:install
```

Section 1.3: Generating A Controller

To generate a controller (for example `Posts`), navigate to your project directory from a command line or terminal, and run:

```
$ rails generate controller Posts
```

You can shorten this code by replacing `generate` with `g`, for example:

```
$ rails g controller Posts
```

If you open up the newly generated app/controllers/**posts_controller.rb** you'll see a controller with no actions:

```
class PostsController < ApplicationController
    # empty
end
```

It's possible to create default methods for the controller by passing in controller name arguments.

```
$ rails g controller ControllerName method1 method2
```

To create a controller within a module, specify the controller name as a path like `parent_module/controller_name`. For example:

```
$ rails generate controller CreditCards open debit credit close
# OR
$ rails g controller CreditCards open debit credit close
```

This will generate the following files:

```
Controller: app/controllers/credit_cards_controller.rb
Test:       test/controllers/credit_cards_controller_test.rb
Views:      app/views/credit_cards/debit.html.erb [...etc]
Helper:     app/helpers/credit_cards_helper.rb
```

A controller is simply a class that is defined to inherit from `ApplicationController`.

It's inside this class that you'll define methods that will become the actions for this controller.

Section 1.4: Installing Rails

Installing Rails on Ubuntu

On a clean ubuntu, installation of Rails should be straight forward

Upgrading ubuntu packages

```
sudo apt-get update
sudo apt-get upgrade
```

Install Ruby and Rails dependecies

```
sudo apt-get install git-core curl zlib1g-dev build-essential libssl-dev libreadline-dev libyaml-dev libsqlite3-dev sqlite3 libxml2-dev libxslt1-dev libcurl4-openssl-dev python-software-properties libffi-dev
```

Installing ruby version manager. In this case the easy one is using rbenv

```
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
```

Installing Ruby Build

```
git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.bashrc
```

Restart Shell

```
exec $SHELL
```

Install ruby

```
rbenv install 2.3.1
rbenv global 2.3.1
rbenv rehash
```

Installing rails

```
gem install rails
```

Installing Rails on Windows

Step 1: ***Installing Ruby***

We need Ruby programming language installed. We can use a precompiled version of Ruby called RubyInstaller.

- Download and run Ruby Installer from rubyinstaller.org.
- Run the installer. Check "Add Ruby executables to your PATH", then install.
- To access Ruby, go to the Windows menu, click All Programs, scroll down to Ruby, and click "Start Command Prompt with Ruby". A command prompt terminal will open. If you type `ruby -v` and press Enter, you should see the Ruby version number that you installed.

Step 2: ***Ruby Development Kit***

After installing Ruby, we can try to install Rails. But some of the libraries Rails depends on need some build tools in order to be compiled, and Windows lacks those tools by default. You can identify this if you see an error while attempting to install Rails `Gem::InstallError`: The '`[gem name]`' `native gem requires installed build tools.` To fix this, we need to install the Ruby Development Kit.

- Download the DevKit
- Run the installer.
- We need to specify a folder where we're going to permanently install the DevKit. I recommend installing it in the root of your hard drive, at `C:\RubyDevKit`. (Don't use spaces in the directory name.)

Now we need to make the DevKit tools available to Ruby.

- In your command prompt, change to the DevKit directory. `cd C:\RubyDevKit` or whatever directory you installed it in.
- We need to run a Ruby script to initialize the DevKit setup. Type `ruby dk.rb init`. Now we'll tell that same script to add the DevKit to our Ruby installation. Type `ruby dk.rb install`.

The DevKit should now be available for your Ruby tools to use when installing new libraries.

Step 3: ***Rails***

Now we can install Rails. Rails comes as a Ruby gem. In your command prompt, type:

```
gem install rails
```

Once you press Enter, the `gem` program will download and install that version of the Rails gem, along with all the other gems Rails depends on.

Step 4: ***Node.js***

Some libraries that Rails depends on require a JavaScript runtime to be installed. Let's install Node.js so that those libraries work properly.

- Download the Node.js installer from **here**.
- When the download completes, visit your downloads folder, and run the `node-v4.4.7.pkg` installer.
- Read the full license agreement, accept the terms, and click Next through the rest of the wizard, leaving everything at the default.

- A window may pop up asking if you want to allow the app to make changes to your computer. Click "Yes".
- When the installation is complete, you'll need to restart your computer so Rails can access Node.js.

Once your computer restarts, don't forget to go to the Windows menu, click "All Programs", scroll down to Ruby, and click "Start Command Prompt with Ruby".

Section 1.5: Create a new Rails app with a non-standard database adapter

Rails is shipped by default with `ActiveRecord`, an ORM (Object Relational Mapping) derived from the pattern with the [same name](#).

As an ORM, it is built to handle relational-mapping, and more precisely by handling SQL requests for you, hence the limitation to SQL databases only.

However, you can still create a Rails app with another database management system:

1. simply create your app without active-record

```
$ rails app new MyApp --skip-active-record
```

2. add your own database management system in `Gemfile`

```
gem 'mongoid', '~> 5.0'
```

3. `bundle install` and follow the installation steps from the desired database.

In this example, `mongoid` is an object mapping for `MongoDB` and - as many other database gems built for rails - it also inherits from `ActiveModel` the same way as `ActiveRecord`, which provides a common interface for many features such as validations, callbacks, translations, etc.

Other database adapters include, but are not limited to :

- datamapper
- sequel-rails

Section 1.6: Creating Rails APIs in JSON

This example assumes that you have experience in creating Rails applications.

To create an API-only app in Rails 5, run

```
rails new name-of-app --api
```

Add active_model_serializers in Gemfile

```
gem 'active_model_serializers'
```

install bundle in terminal

```
bundle install
```

Set the `ActiveModelSerializer` adapter to use `:json_api`

```
# config/initializers/active_model_serializer.rb
ActiveModelSerializers.config.adapter = :json_api
Mime::Type.register "application/json", :json, %w( text/x-json application/jsonrequest
application/vnd.api+json )
```

Generate a new scaffold for your resource

```
rails generate scaffold Task name:string description:text
```

This will generate the following files:

Controller: app/controllers/tasks_controller.rb

```
Test:       test/models/task_test.rb
            test/controllers/tasks_controller_test.rb
Routes:     resources :tasks added in routes.rb
Migration:  db/migrate/_create_tasks.rb
Model:      app/models/task.rb
Serializer: app/serializers/task_serializer.rb
Controller: app/controllers/tasks_controller.rb
```

Section 1.7: Generate a Resource with Scaffolds

From guides.rubyonrails.org:

> Instead of generating a model directly . . . let's set up a scaffold. A scaffold in Rails is a full set of model,
> database migration for that model, controller to manipulate it, views to view and manipulate the data,
> and a test suite for each of the above.

Here's an example of scaffolding a resource called Task with a string name and a text description:

```
rails generate scaffold Task name:string description:text
```

This will generate the following files:

```
Controller: app/controllers/tasks_controller.rb
Test:       test/models/task_test.rb
            test/controllers/tasks_controller_test.rb
Routes:     resources :tasks added in routes.rb
Views:      app/views/tasks
            app/views/tasks/index.html.erb
            app/views/tasks/edit.html.erb
            app/views/tasks/show.html.erb
            app/views/tasks/new.html.erb
            app/views/tasks/_form.html.erb
Helper:     app/helpers/tasks_helper.rb
JS:         app/assets/javascripts/tasks.coffee
CSS:        app/assets/stylesheets/tasks.scss
            app/assets/stylesheets/scaffolds.scss
```

example to delete files generated by scaffold for the resource called Task

```
rails destroy scaffold Task
```

Chapter 2: Routing

The Rails router recognizes URLs and dispatches them to a controller's action. It can also generate paths and URLs, avoiding the need to hardcode strings in your views.

Section 2.1: Resource Routing (Basic)

Routes are defined in `config/routes.rb`. They are often defined as a group of related routes, using the <u>resources</u> or <u>resource</u> methods.

`resources :users` creates the following seven routes, all mapping to actions of `UsersController`:

```
get        '/users',        to: 'users#index'
post       '/users',        to: 'users#create'
get        '/users/new',    to: 'users#new'
get        '/users/:id/edit', to: 'users#edit'
get        '/users/:id',    to: 'users#show'
patch/put  '/users/:id',    to: 'users#update'
delete     '/users/:id',    to: 'users#destroy'
```

Action names are shown after the # in the `to` parameter above. Methods with those same names must be defined in `app/controllers/users_controller.rb` as follows:

```
class UsersController < ApplicationController
  def index
  end

  def create
  end

  # continue with all the other methods...
end
```

You can limit the actions that gets generated with `only` or `except`:

```
resources :users, only:   [:show]
resources :users, except: [:show, :index]
```

You can view all the routes of your application at any given time by running:

Version < 5.0

```
$ rake routes
```

Version ≥ 5.0

```
$ rake routes
# OR
$ rails routes
```

```
users      GET    /users(.:format)          users#index
           POST   /users(.:format)          users#create
new_user   GET    /users/new(.:format)      users#new
edit_user  GET    /users/:id/edit(.:format) users#edit
user       GET    /users/:id(.:format)      users#show
           PATCH  /users/:id(.:format)      users#update
           PUT    /users/:id(.:format)      users#update
           DELETE /users/:id(.:format)      users#destroy
```

To see only the routes that map to a particular controller:

```
Version < 5.0
$ rake routes -c static_pages
static_pages_home    GET    /static_pages/home(.:format)    static_pages#home
static_pages_help    GET    /static_pages/help(.:format)    static_pages#help
Version ≥ 5.0
$ rake routes -c static_pages
static_pages_home    GET    /static_pages/home(.:format)    static_pages#home
static_pages_help    GET    /static_pages/help(.:format)    static_pages#help

# OR

$ rails routes -c static_pages
static_pages_home    GET    /static_pages/home(.:format)    static_pages#home
static_pages_help    GET    /static_pages/help(.:format)    static_pages#help
```

You can search through routes using the `-g` option. This shows any route that partially matches the helper method name, the URL path or the HTTP verb:

```
Version < 5.0
$ rake routes -g new_user      # Matches helper method
$ rake routes -g POST          # Matches HTTP Verb POST
Version ≥ 5.0
$ rake routes -g new_user      # Matches helper method
$ rake routes -g POST          # Matches HTTP Verb POST
# OR
$ rails routes -g new_user     # Matches helper method
$ rails routes -g POST         # Matches HTTP Verb POST
```

Additionally, when running `rails` server in development mode, you can access a web page that shows all your routes with a search filter, matched in priority from top to bottom, at **<hostname>**`/rails/info/routes`. It will look like this:

Helper	HTTP Verb	Path	Controller#Action
Path / Url		[Path Match]	
users_path	GET	/users(.:format)	users#index
	POST	/users(.:format)	users#create
new_user_path	GET	/users/new(.:format)	users#new
edit_user_path	GET	/users/:id/edit(.:format)	users#edit
user_path	GET	/users/:id(.:format)	users#show
	PATCH	/users/:id(.:format)	users#update
	PUT	/users/:id(.:format)	users#update
	DELETE	/users/:id(.:format)	users#destroy

Routes can be declared available for only members (not collections) using the method `resource` instead of `resources` in `routes.rb`. With `resource`, an `index` route is not created by default, but only when explicitly asking for one like this:

```
resource :orders, only: [:index, :create, :show]
```

Section 2.2: Constraints

You can filter what routes are available using constraints.

There are several ways to use constraints including:

- [segment constraints](#),
- [request based constraints](#)
- [advanced constraints](#)

For example, a requested based constraint to only allow a specific IP address to access a route:

```ruby
constraints(ip: /127\.0\.0\.1$/) do
  get 'route', to: "controller#action"
end
```

[See other similar examples ActionDispatch::Routing::Mapper::Scoping](#).

If you want to do something more complex you can use more advanced constraints and create a class to wrap the logic:

```ruby
# lib/api_version_constraint.rb
class ApiVersionConstraint
  def initialize(version:, default:)
    @version = version
    @default = default
  end

  def version_header
    "application/vnd.my-app.v#{@version}"
  end

  def matches?(request)
    @default || request.headers["Accept"].include?(version_header)
  end
end

# config/routes.rb
require "api_version_constraint"

Rails.application.routes.draw do
  namespace :v1, constraints: ApiVersionConstraint.new(version: 1, default: true) do
    resources :users # Will route to app/controllers/v1/users_controller.rb
  end

  namespace :v2, constraints: ApiVersionConstraint.new(version: 2) do
    resources :users # Will route to app/controllers/v2/users_controller.rb
  end
end
```

One form, several submit buttons

You can also use the value of the submit tags of a form as a constraint to route to a different action. If you have a form with multiple submit buttons (eg "preview" and "submit"), you could capture this constraint directly in your `routes.rb`, instead of writing javascript to change the form destination URL. For example with the [commit_param_routing](#) gem you can take advantage of rails `submit_tag`

Rails `submit_tag` first parameter lets you change the value of your form commit parameter

```erb
# app/views/orders/mass_order.html.erb
<%= form_for(@orders, url: mass_create_order_path do |f| %>
    <!-- Big form here -->
  <%= submit_tag "Preview" %>
  <%= submit_tag "Submit" %>
```

```
  # => <input name="commit" type="submit" value="Preview" />
  # => <input name="commit" type="submit" value="Submit" />
  ...
<% end %>


# config/routes.rb
resources :orders do
  # Both routes below describe the same POST URL, but route to different actions
  post 'mass_order', on: :collection, as: 'mass_order',
    constraints: CommitParamRouting.new('Submit'), action: 'mass_create' # when the user presses
"submit"
  post 'mass_order', on: :collection,
    constraints: CommitParamRouting.new('Preview'), action: 'mass_create_preview' # when the user
presses "preview"
  # Note the `as:` is defined only once, since the path helper is mass_create_order_path for the form
url
  # CommitParamRouting is just a class like ApiVersionContraint
end
```

Section 2.3: Scoping routes

Rails provides several ways to organize your routes.

Scope by URL:

```
scope 'admin' do
  get 'dashboard', to: 'administration#dashboard'
  resources 'employees'
end
```

This generates the following routes

```
get       '/admin/dashboard',       to: 'administration#dashboard'
post      '/admin/employees',       to: 'employees#create'
get       '/admin/employees/new',   to: 'employees#new'
get       '/admin/employees/:id/edit', to: 'employees#edit'
get       '/admin/employees/:id',   to: 'employees#show'
patch/put '/admin/employees/:id',   to: 'employees#update'
delete    '/admin/employees/:id',   to: 'employees#destroy'
```

It may make more sense, on the server side, to keep some views in a different subfolder, to separate admin views from user views.

Scope by module

```
scope module: :admin do
  get 'dashboard', to: 'administration#dashboard'
end
```

`module` looks for the controller files under the subfolder of the given name

```
get       '/dashboard',             to: 'admin/administration#dashboard'
```

You can rename the path helpers prefix by adding an as parameter

```
scope 'admin', as: :administration do
  get 'dashboard'
end
```

```
# => administration_dashboard_path
```

Rails provides a convenient way to do all the above, using the `namespace` method. The following declarations are equivalent

```
namespace :admin do
end

scope 'admin', module: :admin, as: :admin
```

Scope by controller

```
scope controller: :management do
  get 'dashboard'
  get 'performance'
end
```

This generate these routes

```
get        '/dashboard',         to: 'management#dashboard'
get        '/performance',       to: 'management#performance'
```

Shallow Nesting

Resource routes accept a `:shallow` option that helps to shorten URLs where possible. Resources shouldn't be nested more than one level deep. One way to avoid this is by creating shallow routes. The goal is to leave off parent collection URL segments where they are not needed. The end result is that the only nested routes generated are for the `:index`, `:create`, and `:new` actions. The rest are kept in their own shallow URL context. There are two options for scope to custom shallow routes:

- **:shallow_path**: Prefixes member paths with a specified parameter

  ```
  scope shallow_path: "sekret" do
    resources :articles do
      resources :comments, shallow: true
    end
  end
  ```

- **:shallow_prefix**: Add specified parameters to named helpers

  ```
  scope shallow_prefix: "sekret" do
    resources :articles do
      resources :comments, shallow: true
    end
  end
  ```

We can also illustrate `shallow` routes more by:

```
resources :auctions, shallow: true do
  resources :bids do
   resources :comments
  end
end
```

alternatively coded as follows (if you're block-happy):

```
resources :auctions do
  shallow do
    resources :bids do
      resources :comments
    end
  end
end
```

The resulting routes are:

Prefix	Verb	URI Pattern
bid_comments	GET	/bids/:bid_id/comments(.:format)
	POST	/bids/:bid_id/comments(.:format)
new_bid_comment	GET	/bids/:bid_id/comments/new(.:format)
edit_comment	GET	/comments/:id/edit(.:format)
comment	GET	/comments/:id(.:format)
	PATCH	/comments/:id(.:format)
	PUT	/comments/:id(.:format)
	DELETE	/comments/:id(.:format)
auction_bids	GET	/auctions/:auction_id/bids(.:format)
	POST	/auctions/:auction_id/bids(.:format)
new_auction_bid	GET	/auctions/:auction_id/bids/new(.:format)
edit_bid	GET	/bids/:id/edit(.:format)
bid	GET	/bids/:id(.:format)
	PATCH	/bids/:id(.:format)
	PUT	/bids/:id(.:format)
	DELETE	/bids/:id(.:format)
auctions	GET	/auctions(.:format)
	POST	/auctions(.:format)
new_auction	GET	/auctions/new(.:format)
edit_auction	GET	/auctions/:id/edit(.:format)
auction	GET	/auctions/:id(.:format)
	PATCH	/auctions/:id(.:format)
	PUT	/auctions/:id(.:format)
	DELETE	/auctions/:id(.:format)

If you analyze the routes generated carefully, you'll notice that the nested parts of the URL are only included when they are needed to determine what data to display.

Section 2.4: Concerns

To avoid repetition in nested routes, concerns provide a great way of sharing common resources that are reusable. To create a concern use the method `concern` within the `routes.rb` file. The method expects a symbol and block:

```
concern :commentable do
  resources :comments
end
```

While not creating any routes itself, this code allows using the `:concerns` attribute on a resource. The simplest example would be:

```
resource :page, concerns: :commentable
```

The equivalent nested resource would look like this:

```
resource :page do
  resource :comments
end
```

This would build, for example, the following routes:

```
/pages/#{page_id}/comments
/pages/#{page_id}/comments/#{comment_id}
```

For concerns to be meaningful, there must be multiple resources that utilize the concern. Additional resources could use any of the following syntax to call the concern:

```
resource :post, concerns: %i(commentable)
resource :blog do
  concerns :commentable
end
```

Section 2.5: Root route

You can add a home page route to your app with the `root` method.

```
# config/routes.rb
Rails.application.routes.draw do
  root "application#index"
  # equivalent to:
  # get "/", "application#index"
end

# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  def index
    render "homepage"
  end
end
```

And in terminal, `rake routes` (`rails routes` in Rails 5) will produce:

```
root     GET     /             application#index
```

Because the homepage is usually the most important route, and routes are prioritized in the order they appear, the `root` route should usually be the first in your routes file.

Section 2.6: Split routes into multiple files

If your routes file is overwhelmingly big, you can put your routes in multiple files and include each of the files with Ruby's `require_relative` method:

`config/routes.rb`:
```
YourAppName::Application.routes.draw do
  require_relative 'routes/admin_routes'
  require_relative 'routes/sidekiq_routes'
  require_relative 'routes/api_routes'
  require_relative 'routes/your_app_routes'
end
```

`config/routes/api_routes.rb`:
```
YourAppName::Application.routes.draw do
```

```ruby
  namespace :api do
    # ...
  end
end
```

Section 2.7: Additional RESTful actions

resources :photos do member do get 'preview' end collection do get 'dashboard' end end

This creates the following routes **in addition to default 7 RESTful routes**:

```ruby
get         '/photos/:id/preview',          to: 'photos#preview'
get         '/photos/dashboards',           to: 'photos#dashboard'
```

If you want to do this for single lines, you can use:

resources :photos do get 'preview', on: :member get 'dashboard', on: :collection end

You can also add an action to the **/**new path:

resources :photos do get 'preview', on: :new end

Which will create:

```ruby
get         '/photos/new/preview',          to: 'photos#preview'
```

Be mindful when adding actions to your RESTful routes, probably you are missing another resource!

Section 2.8: Member and Collection Routes

Defining a member block inside a resource creates a route that can act on an individual member of that resource-based route:

```ruby
resources :posts do
  member do
    get 'preview'
  end
end
```

This generates the following member route:

```ruby
get '/posts/:id/preview', to: 'posts#preview'
# preview_post_path
```

Collection routes allow for creating routes that can act on a collection of resource objects:

```ruby
resources :posts do
  collection do
    get 'search'
  end
end
```

This generates the following collection route:

```ruby
get '/posts/search', to: 'posts#search'
```

```
# search_posts_path
```

An alternate syntax:

```
resources :posts do
  get 'preview', on: :member
  get 'search',  on: :collection
end
```

Section 2.9: Mount another application

mount is used to mount another application (basically rack application) or rails engines to be used within the current application

syntax:

```
mount SomeRackApp, at: "some_route"
```

Now you can access above mounted application using route helper some_rack_app_path or some_rack_app_url.

But if you want to rename this helper name you can do it as:

```
mount SomeRackApp, at: "some_route", as: :myapp
```

This will generate the myapp_path and myapp_url helpers which can be used to navigate to this mounted app.

Section 2.10: Nested Routes

If you want to add nested routes you can write the following code in routes.rb file.

```
resources :admins do
  resources :employees
end
```

This will generate following routes:

```
     admin_employees GET     /admins/:admin_id/employees(.:format)          employees#index
                     POST    /admins/:admin_id/employees(.:format)          employees#create
  new_admin_employee GET     /admins/:admin_id/employees/new(.:format)      employees#new
 edit_admin_employee GET     /admins/:admin_id/employees/:id/edit(.:format) employees#edit
      admin_employee GET     /admins/:admin_id/employees/:id(.:format)      employees#show
                     PATCH   /admins/:admin_id/employees/:id(.:format)      employees#update
                     PUT     /admins/:admin_id/employees/:id(.:format)      employees#update
                     DELETE  /admins/:admin_id/employees/:id(.:format)      employees#destroy
              admins GET     /admins(.:format)                              admins#index
                     POST    /admins(.:format)                              admins#create
           new_admin GET     /admins/new(.:format)                          admins#new
          edit_admin GET     /admins/:id/edit(.:format)                     admins#edit
               admin GET     /admins/:id(.:format)                          admins#show
                     PATCH   /admins/:id(.:format)                          admins#update
                     PUT     /admins/:id(.:format)                          admins#update
                     DELETE  /admins/:id(.:format)                          admins#destroy
```

Section 2.11: Redirection

You can perform redirection in Rails routes as follows:

Version ≥ 4.0

```
get '/stories', to: redirect('/posts')
```

Version < 4.0

```
match "/abc" => redirect("http://example.com/abc")
```

You can also redirect all unknown routes to a given path:

Version ≥ 4.0

```
match '*path' => redirect('/'), via: :get
# or
get '*path' => redirect('/')
```

Version < 4.0

```
match '*path' => redirect('/')
```

Section 2.12: Redirects and Wildcard Routes

If you want to provide a URL out of convenience for your user but map it directly to another one you're already using. Use a redirect:

```
# config/routes.rb
TestApp::Application.routes.draw do
  get 'courses/:course_name' => redirect('/courses/%{course_name}/lessons'), :as => "course"
end
```

Well, that got interesting fast. The basic principle here is to just use the *#redirect* method to send one route to another route. If your route is quite simple, it's a really straightforward method. But if you want to also send the original parameters, you need to do a bit of gymnastics by capturing the parameter inside **%{**here**}**. Note the single quotes around everything.

In the example above, we've also renamed the route for convenience by using an alias with the :as parameter. This lets us use that name in methods like the #_path helpers. Again, test out your $ rake routes with questions.

Section 2.13: Scope available locales

If your application is available in different languages, you usually show the current locale in the URL.

```
scope '/(:locale)', locale: /#{I18n.available_locales.join('|')}/ do
    root 'example#root'
    # other routes
end
```

Your root will be accessible via the locales defined in I18n.available_locales.

Section 2.14: URL params with a period

If you want to support a url parameter more complex than an id number, you may run into trouble with the parser if the value contains a period. Anything following a period will be assumed to be a format (i.e. json, xml).

You can work around this limitation by using a constraint to *broaden* the accepted input.

For example, if you want to reference a user record by email address in the url:

```
resources :users, constraints: { id: /.*/ }
```

Chapter 3: ActiveRecord

Section 3.1: Creating a Model via generator

Ruby on Rails provides a `model` generator you can use to create ActiveRecord models. Simply use `rails generate model` and provide the model name.

```
$ rails g model user
```

In addition to the model file in `app/models`, the generator will also create:

- the Test in `test/models/user_test.rb`
- the Fixtures in `test/fixtures/users.yml`
- the database Migration in `db/migrate/XXX_create_users.rb`

You can also generate some fields for the model when generating it.

```
$ rails g model user email:string sign_in_count:integer birthday:date
```

This will create the columns email, sign_in_count and birthday in your database, with the appropriate types.

Section 3.2: Introduction to Callbacks

A callback is a method that gets called at specific moments of an object's lifecycle (right before or after creation, deletion, update, validation, saving or loading from the database).

For instance, say you have a listing that expires within 30 days of creation.

One way to do that is like this:

```ruby
class Listing < ApplicationRecord
  after_create :set_expiry_date

  private

  def set_expiry_date
    expiry_date = Date.today + 30.days
    self.update_column(:expires_on, expiry_date)
  end
end
```

All of the available methods for callbacks are as follows, in the same order that they are called during the operation of each object:

Creating an Object

- before_validation
- after_validation
- before_save
- around_save
- before_create
- around_create
- after_create
- after_save

- after_commit/after_rollback

Updating an Object

- before_validation
- after_validation
- before_save
- around_save
- before_update
- around_update
- after_update
- after_save
- after_commit/after_rollback

Destroying an Object

- before_destroy
- around_destroy
- after_destroy
- after_commit/after_rollback

NOTE: *after_save runs both on create and update, but always after the more specific callbacks after_create and after_update, no matter the order in which the macro calls were executed.*

Section 3.3: Creating a Model manually

While using scaffolding is a fast and easy if you are new to Rails or you are creating a new application, later it can be useful just to do it on your own ato avoid the need to go through the scaffold-generated code to slim it down (remove unused parts, etc.).

Creating a model can be as simple as creating a file under `app/models`.

The most simple model, in `ActiveRecord`, is a class that extends **`ActiveRecord::Base`**.

```
class User < ActiveRecord::Base
end
```

Model files are stored in `app/models/`, and the file name corresponds to the singular name of the class:

```
# user
app/models/user.rb

# SomeModel
app/models/some_model.rb
```

The class will inherit all the ActiveRecord features: query methods, validations, callbacks, etc.

```
# Searches the User with ID 1
User.find(1)
```

Note: Make sure that the table for the corresponding model exists. If not, you can create the table by creating a Migration

You can generate a model and it's migration by terminal from the following command

```
rails g model column_name1:data_type1, column_name2:data_type2, ...
```

and can also assign foreign key(relationship) to the model by following command

```
rails g model column_name:data_type, model_name:references
```

Section 3.4: Manually Testing Your Models

Testing your Active Record models through your command line interface is simple. Navigate to the app directory in your terminal and type in `rails console` to start the Rails console. From here, you can run active record methods on your database.

For example, if you had a database schema with a Users table having a `name:`**`string`** column and `email:`**`string`**, you could run:

```
User.create name: "John", email: "john@example.com"
```

Then, to show that record, you could run:

```
User.find_by email: "john@example.com"
```

Or if this is your first or only record, you could simply get the first record by running:

```
User.first
```

Section 3.5: Creating A Migration

Add/remove fields in existing tables

Create a migration by running:

```
rails generate migration AddTitleToCategories title:string
```

This will create a migration that adds a `title` column to a `categories` table:

```ruby
class AddTitleToCategories < ActiveRecord::Migration[5.0]
  def change
    add_column :categories, :title, :string
  end
end
```

Similarly, you can generate a migration to remove a column: `rails generate migration RemoveTitleFromCategories title:string`

This will create a migration that removes a `title` column from the `categories` table:

```ruby
class RemoveTitleFromCategories < ActiveRecord::Migration[5.0]
  def change
    remove_column :categories, :title, :string
  end
end
```

While, strictly speaking, specifying **type** (:**`string`** in this case) is **not necessary** for removing a column, **it's helpful**, since it provides the information necessary for **rolling it back**.

Create a table

Create a migration by running:

```
rails g CreateUsers name bio
```

Rails recognizes the intent to create a table from the `Create` prefix, the rest of the migration name will be used as a table name. The given example generates the following:

```ruby
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :bio
    end
  end
end
```

Notice that the creation command didn't specify types of columns and the default `string` was used.

Create a join table

Create a migration by running:

```
rails g CreateJoinTableParticipation user:references group:references
```

Rails detects the intent to create a join table by finding `JoinTable` in migration name. Everything else is determined from the names of the fields you give after the name.

```ruby
class CreateJoinTableParticipation < ActiveRecord::Migration
  def change
    create_join_table :users, :groups do |t|
      # t.index [:user_id, :group_id]
      # t.index [:group_id, :user_id]
    end
  end
end
```

Uncomment the necessary `index` statements and delete the rest.

Precedence

Notice that the example migration name `CreateJoinTableParticipation` matches the rule for table creation: it has a `Create` prefix. But it did not generate a simple `create_table`. This is because migration generator ([source code](#)) uses a **first match** of the following list:

- `(Add|Remove)<ignored>(To|From)<table_name>`

- `<ignored>JoinTable<ignored>`

- `Create<table_name>`

Section 3.6: Create a Join Table using Migrations

Specially useful for `has_and_belongs_to_many` relation, you can manually create a join table using the `create_table` method. Suppose you have two models `Tags` and `Proyects`, and you'd like to associate them using a

has_and_belongs_to_many relation. You need a join table to associate instances of both classes.

```ruby
class CreateProjectsTagsJoinTableMigration < ActiveRecord::Migration
  def change
    create_table :projects_tags, id: false do |t|
      t.integer :project_id
      t.integer :tag_id
    end
  end
end
```

The actual name of the table needs to follow this convention: the model which alphabetically precedes the other must go first. **P**roject preceds **T**ags so the name of the table is projects_tags.

Also since the purpose of this table is to route the association between the instances of two models, the actual id of every record in this table is not necessary. You specify this by passing id: **false**

Finally, as is convention in Rails, the table name must be the compound plural form of the individual models, but the column of the table must be in singular form.

Section 3.7: Using a model instance to update a row

Let's say you have a User model

```ruby
class User < ActiveRecord::Base
end
```

Now to update the first_name and last_name of a user with id = 1, you can write the following code.

```ruby
user = User.find(1)
user.update(first_name: 'Kashif', last_name: 'Liaqat')
```

Calling update will attempt to update the given attributes in a single transaction, returning **true** if successful and **false** if not.

Chapter 4: Views

Section 4.1: Structure

As Rails follows the M**V**C pattern `Views` are where your "templates" are for your actions.

Let's say you have a controller `articles_controller.rb`. For this controller you would have a folder in views called `app/views/articles`:

```
app
|-- controllers
|    '-- articles_controller.rb
|
'-- views
'-- articles
|    |- index.html.erb
|    |- edit.html.erb
|    |- show.html.erb
|    |- new.html.erb
|    '- _partial_view.html.erb
|
'-- [...]
```

This structure allows you to have a folder for each controller. When calling an action in your controller the appropriate view will be rendered automatically.

```
// articles_controller.rb
class ArticlesController < ActionController::Base
  def show
  end
end


// show.html.erb
<h1>My show view</h1>
```

Section 4.2: Partials

Partial templates (partials) are a way of breaking the rendering process into more manageable chunks. Partials allow you to extract pieces of code from your templates to separate files and also reuse them throughout your templates.

To *create* a partial, create a new file that begins with an underscore: `_form.html.erb`

To *render* a partial as part of a view, use the render method within the view: `<%= render "form" %>`

- Note, the underscore is left out when rendering
- A partial has to be rendered using its path if located in a different folder

To *pass* a variable into the partial as a local variable, use this notation:

```
<%= render :partial => 'form', locals: { post: @post } %>
```

Partials are also useful when you need to *reuse* exactly the same code (**DRY philosophy**).

For example, to reuse **\<head\>** code, create a partial named `_html_header.html.erb`, enter your **\<head\>** code to be

reused, and render the partial whenever needed by: `<%= render 'html_header' %>`.

Object Partials

Objects that respond to `to_partial_path` can also be rendered, as in `<%= render @post %>`. By default, for ActiveRecord models, this will be something like `posts/post`, so by actually rendering **@post**, the file `views/posts/_post.html.erb` will be rendered.

A local named `post` will be automatically assigned. In the end, `<%= render @post %>` is a short hand for `<%= render 'posts/post', post: @post %>`.

Collections of objects that respond to `to_partial_path` can also be provided, such as `<%= render @posts %>`. Each item will be rendered consecutively.

Global Partials

To create a global partial that can be used anywhere without referencing its exact path, the partial has to be located in the `views/application` path. The previous example has been modified below to illustrate this feature.

For example, this is a path to a global partial app`/`views`/`application`/_`html_header`.`html`.`erb:

To render this global partial anywhere, use `<%= render 'html_header' %>`

Section 4.3: AssetTagHelper

To let rails automatically and correctly link assets (css/js/images) in most cases you want to use built in helpers. ([Official documentation](#))

Image helpers
image_path

This returns the path to an image asset in app`/`assets`/i`mages.

```
image_path("edit.png") # => /assets/edit.png
```

image_url

This returns the full URL to an image asset in app`/`assets`/i`mages.

```
image_url("edit.png") # => http://www.example.com/assets/edit.png
```

image_tag

Use this helper if you want to include an `<img src="" />`-tag with the source set.

```
image_tag("icon.png") # => <img src="/assets/icon.png" alt="Icon" />
```
JavaScript helpers
javascript_include_tag

If you want to include a JavaScript-file in your view.

```
javascript_include_tag "application" # => <script src="/assets/application.js"></script>
```

javascript_path

This returns the path of your JavaScript-file.

```
javascript_path "application" # => /assets/application.js
```

javascript_url

This returns the full URL of your JavaScript-file.

```
javascript_url "application" # => http://www.example.com/assets/application.js
```

Stylesheet helpers

stylesheet_link_tag

If you want to include a CSS-file in your view.

```
stylesheet_link_tag "application" # => <link href="/assets/application.css" media="screen"
rel="stylesheet" />
```

stylesheet_path

This returns the path of you stylesheet asset.

```
stylesheet_path "application" # => /assets/application.css
```

stylesheet_url

This returns the full URL of you stylesheet asset.

```
stylesheet_url "application" # => http://www.example.com/assets/application.css
```

Example usage

When creating a new rails app you will automatically have two of these helpers in
app**/**views**/**layouts**/**application.html.erb

```
<%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track': 'reload' %>
<%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
```

This outputs:

```
// CSS
<link rel="stylesheet" media="all" href="/assets/application.self-
e19d4b856cacba4f6fb0e5aa82a1ba9aa4ad616f0213a1259650b281d9cf6b20.css?body=1" data-turbolinks-
track="reload" />
// JavaScript
<script
src="/assets/application.self-619d9bf310b8eb258c67de7af745cafbf2a98f6d4c7bb6db8e1b00aed89eb0b1.js?b
ody=1" data-turbolinks-track="reload"></script>
```

Section 4.4: Replace HTML code in Views

If you ever wanted to determine the html content to be printed on a page during run time then, rails has a very
good solution for that. It has something called the **content_for** which allows us to pass a block to a rails view.
Please check the below example,

Declare content_for

```
<div>
  <%= yield :header %>
</div>

<% content_for :header do %>
 <ul>
   <li>Line Item 1</li>
   <li>Line Item 2</li>
 </ul>
<% end %>
```

Section 4.5: HAML - an alternative way to use in your views

HAML (HTML abstraction markup language) is a beautiful and elegant way to describe and design the HTML of your views. Instead of opening- and closing tags, HAML uses indentation for the structure of your pages. Basically, if something should be placed within another element, you just indent it by using one tab stop. Tabs and white space are important in HAML, so be sure that you always use the same amount of tabs.

Examples:

```
#myview.html.erb
<h1><%= @the_title %></h1>
<p>This is my form</p>
<%= render "form" %>
```

And in HAML:

```
#myview.html.haml
%h1= @the_title
%p
    This is my form
= render 'form'
```

You see, the structure of the layout is much clearer than using HTML and ERB.

Installation

Just install the gem using

```
gem install haml
```

and add the gem to the Gemfile

```
gem "haml"
```

For using HAML instead of HTML/ERB, just replace the file extensions of your views from something.html.erb to something.html.haml.

Quick tipps

Common elements like divs can be written in a short way

HTML

```
<div class="myclass">My Text</div>
```

HAML

```
%div.myclass
```

HAML, shorthand

```
.myclass
```

Attributes

HTML

```
<p class="myclass" id="myid">My paragraph</p>
```

HAML

```
%p{:class => "myclass", :id => "myid"} My paragraph
```

Inserting ruby code

You can insert ruby code with the = and - signs.

```
= link_to "Home", home_path
```

Code starting with **=** will be executed and embedded into the document.

Code starting with **-** will be executed, but not inserted into the document.

Full documentation

HAML is very easy to start with, but is also very complex, so that I'll recommend reading the documentation.

Chapter 5: ActiveRecord Migrations

Column type	Description
:primary_key	Primary key
:string	Shorter string datatype. Allows `limit` option for maximum number of characters.
:text	Longer amount of text. Allows `limit` option for maximum number of bytes.
:integer	Integer. Allows `limit` option for maximum number of bytes.
:bigint	Larger integer
:float	Float
:decimal	Decimal number with variable precision. Allows `precision` and `scale` options.
:numeric	Allows `precision` and `scale` options.
:datetime	DateTime object for dates/times.
:time	Time object for times.
:date	Date object for dates.
:binary	Binary data. Allows `limit` option for maximum number of bytes.
:boolean	Boolean

Section 5.1: Adding multiple columns to a table

To add multiple columns to a table, separate `field:type` pairs with spaces when using `rails generate migration` command.

The general syntax is:

```
rails generate migration NAME [field[:type][:index] field[:type][:index]] [options]
```

For example, the following will add `name`, `salary` and `email` fields to the `users` table:

```
rails generate migration AddDetailsToUsers name:string salary:decimal email:string
```

Which generates the following migration:

```ruby
class AddDetailsToUsers < ActiveRecord::Migration[5.0]
  def change
    add_column :users, :name, :string
    add_column :users, :salary, :decimal
    add_column :users, :email, :string
  end
end
```

Section 5.2: Add a reference column to a table

To add a reference to a `team` to the `users` table, run this command:

```
$ rails generate migration AddTeamRefToUsers team:references
```

This generates the following migration:

```ruby
class AddTeamRefToUsers < ActiveRecord::Migration[5.0]
  def change
    add_reference :users, :team, foreign_key: true
  end
end
```

That migration will create a `team_id` column in the `users` table.

If you want to add an appropriate `index` and `foreign_key` on the added column, change the command to `rails generate migration AddTeamRefToUsers team:references:index`. This will generate the following migration:

```ruby
class AddTeamRefToUsers < ActiveRecord::Migration
  def change
    add_reference :users, :team, index: true
    add_foreign_key :users, :teams
  end
end
```

If you want to name your reference column other than what Rails auto generates, add the following to your migration: (E.g.: You might want to call the `User` who created the `Post` as `Author` in the `Post` table)

```ruby
class AddAuthorRefToPosts < ActiveRecord::Migration
  def change
    add_reference :posts, :author, references: :users, index: true
  end
end
```

Section 5.3: Rollback migrations

To `rollback` the latest migration, either by reverting the `change` method or by running the `down` method. Run command:

Version < 5.0
```
rake db:rollback
```
Version ≥ 5.0
```
rails db:rollback
```
Rollback the last 3 migrations
Version < 5.0
```
rake db:rollback STEP=3
```
Version ≥ 5.0
```
rails db:rollback STEP=3
```

STEP provide the number of migrations to revert.

Rollback all migrations
Version < 5.0
```
rake db:rollback VERSION=0
```
Version ≥ 5.0
```
rails db:rollback VERSION=0
```

Section 5.4: Add a new column with an index

To add a new *indexed* column `email` to the `users` table, run the command:

```
rails generate migration AddEmailToUsers email:string:index
```

This will generate the following migration:

```ruby
class AddEmailToUsers < ActiveRecord::Migration[5.0]
  def change
```

```
        add_column :users, :email, :string
        add_index :users, :email
    end
end
```

Section 5.5: Run specific migration

To run a specific migration up or down, use `db:migrate:up` or `db:migrate:down`.

Up a specific migration:

Version < 5.0
```
rake db:migrate:up VERSION=20090408054555
```
Version ≥ 5.0
```
rails db:migrate:up VERSION=20090408054555
```

Down a specific migration:

Version < 5.0
```
rake db:migrate:down VERSION=20090408054555
```
Version ≥ 5.0
```
rails db:migrate:down VERSION=20090408054555
```

The version number in the above commands is the numeric prefix in the migration's filename. For example, to migrate to the migration `20160515085959_add_name_to_users.rb`, you would use `20160515085959` as the version number.

Section 5.6: Redo migrations

You can rollback and then migrate again using the **redo** command. This is basically a shortcut that combines `rollback` and `migrate` tasks.

Run command:

Version < 5.0
```
rake db:migrate:redo
```
Version ≥ 5.0
```
rails db:migrate:redo
```

You can use the `STEP` parameter to go back more than one version.

For example, to go back 3 migrations:

Version < 5.0
```
rake db:migrate:redo STEP=3
```
Version ≥ 5.0
```
rails db:migrate:redo STEP=3
```

Section 5.7: Add a new column to a table

To add a new column `name` to the `users` table, run the command:

```
rails generate migration AddNameToUsers name
```

This generates the following migration:

```
class AddNameToUsers < ActiveRecord::Migration[5.0]
  def change
    add_column :users, :name, :string
  end
end
```

When the migration name is of the form `AddXXXToTABLE_NAME` followed by list of columns with data types, the generated migration will contain the appropriate `add_column` statements.

Section 5.8: Remove an existing column from a table

To remove existing column `name` from `users` table, run the command:

```
rails generate migration RemoveNameFromUsers name:string
```

This will generate the following migration:

```
class RemoveNameFromUsers < ActiveRecord::Migration[5.0]
  def change
    remove_column :users, :name, :string
  end
end
```

When the migration name is of the form `RemoveXXXFromYYY` followed by list of columns with data types then the generated migration will contain the appropriate `remove_column` statements.

While it's not required to specify the data type (e.g. `:string`) as a parameter to `remove_column`, it is highly recommended. If the data type is *not* specified, then the migration will not be reversible.

Section 5.9: Add column with default value

The following example adds a column `admin` to the `users` table, and gives that column the default value `false`.

```
class AddDetailsToUsers < ActiveRecord::Migration[5.0]
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

Migrations with defaults might take a long time in large tables with for example PostgreSQL. This is because each row will have to be updated with the default value for the newly added column. To circumvent this (and reduce downtime during deployments), you can split your migration into three steps:

1. Add a `add_column`-migration similar to the one above, but set no default
2. Deploy and update the column in a rake task or on the console while your app is running. Make sure your application already writes data to that colum for new/updated rows.
3. Add another `change_column` migration, which then changes the default of that column to the desired default value

Section 5.10: Running migrations in different environments

To run migrations in the `test` environment, run this shell command:

```
rake db:migrate RAILS_ENV=test
```

Version ≥ 5.0

Starting in Rails 5.0, you can use `rails` instead of `rake`:

```
rails db:migrate RAILS_ENV=test
```

Section 5.11: Create a new table

To create a new `users` table with the columns `name` and `salary`, run the command:

```
rails generate migration CreateUsers name:string salary:decimal
```

This will generate the following migration:

```ruby
class CreateUsers < ActiveRecord::Migration[5.0]
  def change
    create_table :users do |t|
      t.string :name
      t.decimal :salary
    end
  end
end
```

When the migration name is of the form `CreateXXX` followed by list of columns with data types, then a migration will be generated that creates the table XXX with the listed columns.

Section 5.12: Running migrations

Run command:

Version < 5.0
```
rake db:migrate
```
Version ≥ 5.0
```
rails db:migrate
```

Specifying target version will run the required migrations (up, down, change) until it has reached the specified version. Here, `version number` is the numerical prefix on the migration's filename.

Version < 5.0
```
rake db:migrate VERSION=20080906120000
```
Version ≥ 5.0
```
rails db:migrate VERSION=20080906120000
```

Section 5.13: Change an existing column's type

To modify an existing column in Rails with a migration, run the following command:

```
rails g migration change_column_in_table
```

This will create a new migration file in db/migration directory (if it doesn't exist already), which will contain the file prefixed with timestamp and migration file name which contains the below content:

```ruby
def change
```

```
    change_column(:table_name, :column_name, :new_type)
  end
```

Rails Guide – Changing Columns

A longer but safer method

The above code prevents the user from ever rolling back the migration. You can avoid this problem by splitting the change method into separate up and down methods:

```
def up
  change_column :my_table, :my_column, :new_type
end

def down
  change_column :my_table, :my_column, :old_type
end
```

Section 5.14: Create a hstore column

Hstore columns can be useful to store settings. They are available in PostgreSQL databases after you enabled the extension.

```
class CreatePages < ActiveRecord::Migration[5.0]
  def change
    create_table :pages do |t|
      enable_extension 'hstore' unless extension_enabled?('hstore')
      t.hstore :settings
      t.timestamps
    end
  end
end
```

Section 5.15: Create a join table

To create a join table between students and courses, run the command:

```
$ rails g migration CreateJoinTableStudentCourse student course
```

This will generate the following migration:

```
class CreateJoinTableStudentCourse < ActiveRecord::Migration[5.0]
  def change
    create_join_table :students, :courses do |t|
      # t.index [:student_id, :course_id]
      # t.index [:course_id, :student_id]
    end
  end
end
```

Section 5.16: Add a self reference

A self reference can be useful to build a hierarchical tree. This can be achieved with add_reference in a migration.

```
class AddParentPages < ActiveRecord::Migration[5.0]
  def change
```

```
        add_reference :pages, :pages
    end
end
```

The foreign key column will be pages_id. If you want to decide about the foreign key column name, you have to create the column first and add the reference after.

```
class AddParentPages < ActiveRecord::Migration[5.0]
  def change
    add_column :pages, :parent_id, :integer, null: true, index: true
    add_foreign_key :pages, :pages, column: :parent_id
  end
end
```

Section 5.17: Create an array column

An **array** column is supported by PostgreSQL. Rails will automatically convert a PostgreSQL array to a Ruby array, and vice-versa.

Create a table with an **array** column:

```
create_table :products do |t|
  t.string :name
  t.text :colors, array: true, default: []
end
```

Add an **array** column to an existing table:

```
add_column :products, :colors, array: true, default: []
```

Add an index for an **array** column:

```
add_index :products, :colors, using: 'gin'
```

Section 5.18: Add an unique column to a table

To add a new *unique* column email to users, run the following command:

```
rails generate migration AddEmailToUsers email:string:uniq
```

This will create the following migration:

```
class AddEmailToUsers < ActiveRecord::Migration[5.0]
  def change
    add_column :users, :email, :string
    add_index :users, :email, unique: true
  end
end
```

Section 5.19: Checking migration status

We can check the status of migrations by running

```
Version ≥ 3.0 Version < 5.0
```
```
rake db:migrate:status
```

```
rails db:migrate:status
```

The output will look like this:

```
Status    Migration ID     Migration Name
--------------------------------------------------
up       20140711185212   Create documentation pages
up       20140724111844   Create nifty attachments table
up       20140724114255   Create documentation screenshots
up       20160213170731   Create owners
up       20160218214551   Create users
up       20160221162159   ********** NO FILE **********
up       20160222231219   ********** NO FILE **********
```

Under the status field, up means the migration has been run and down means that we need to run the migration.

Section 5.20: Changing Tables

If you have created a table with some wrong schema, then the easiest way to change the columns and their properties is change_table. Review the following example:

```
change_table :orders do |t|
  t.remove :ordered_at # removes column ordered_at
  t.string :skew_number # adds a new column
  t.index  :skew_number #creates an index
  t.rename :location, :state #renames location column to state
end
```

The above migration changes a table orders. Here is a line-by-line description of the changes:

1. t.remove :ordered_at removes the column ordered_at from the table orders.
2. t.string :skew_number adds a new string-type column named skew_number in the orders table.
3. t.index :skew_number adds an index on the skew_number column in the orders table.
4. t.rename :location, :state renames the location column in the orders table to state.

Section 5.21: Adding a NOT NULL constraint to existing data

Say you want to add a foreign key company_id to the users table, and you want to have a NOT NULL constraint on it. If you already have data in users, you will have to do this in multiple steps.

```
class AddCompanyIdToUsers < ActiveRecord::Migration
  def up
    # add the column with NULL allowed
    add_column :users, :company_id, :integer

    # make sure every row has a value
    User.find_each do |user|
      # find the appropriate company record for the user
      # according to your business logic
      company = Company.first
      user.update!(company_id: company.id)
    end

    # add NOT NULL constraint
    change_column_null :users, :company_id, false
```

```
  end

  # Migrations that manipulate data must use up/down instead of change
  def down
    remove_column :users, :company_id
  end
end
```

Section 5.22: Forbid null values

To forbid null values in your table columns, add the :null parameter to your migration, like this:

```
class AddPriceToProducts < ActiveRecord::Migration
  def change
    add_column :products, :float, null: false
  end
end
```

Chapter 6: Rails Best Practices

Section 6.1: Fat Model, Skinny Controller

"Fat Model, Skinny Controller" refers to how the M and C parts of MVC ideally work together. Namely, any non-response-related logic should go in the model, ideally in a nice, testable method. Meanwhile, the "skinny" controller is simply a nice interface between the view and model.

In practice, this can require a range of different types of refactoring, but it all comes down to one idea: by moving any logic that isn't about the response to the model (instead of the controller), not only have you promoted reuse where possible but you've also made it possible to test your code outside of the context of a request.

Let's look at a simple example. Say you have code like this:

```ruby
def index
  @published_posts = Post.where('published_at <= ?', Time.now)
  @unpublished_posts = Post.where('published_at IS NULL OR published_at > ?', Time.now)
end
```

You can change it to this:

```ruby
def index
  @published_posts = Post.published
  @unpublished_posts = Post.unpublished
end
```

Then, you can move the logic to your post model, where it might look like this:

```ruby
scope :published, ->(timestamp = Time.now) { where('published_at <= ?', timestamp) }
scope :unpublished, ->(timestamp = Time.now) { where('published_at IS NULL OR published_at > ?', timestamp) }
```

Section 6.2: Domain Objects (No More Fat Models)

"Fat Model, Skinny Controller" is a very good first step, but it doesn't scale well once your codebase starts to grow.

Let's think on the Single Responsibility of models. What is the single responsibility of models? Is it to hold business logic? Is it to hold non-response-related logic?

No. Its responsibility is to handle the persistence layer and its abstraction.

Business logic, as well as any non-response-related logic and non-persistence-related logic, should go in domain objects.

Domain objects are classes designed to have only one responsibility in the domain of the problem. Let your classes "Scream Their Architecture" for the problems they solve.

In practice, you should strive towards skinny models, skinny views and skinny controllers. The architecture of your solution shouldn't be influenced by the framework you're choosing.

For example

Let's say you're a marketplace which charges a fixed 15% commission to your customers via Stripe. If you charge a fixed 15% commission, that means that your commission changes depending on the order's amount because Stripe

charges 2.9% + 30¢.

The amount you charge as commission should be: `amount*0.15 - (amount*0.029 + 0.30)`.

Don't write this logic in the model:

```ruby
# app/models/order.rb
class Order < ActiveRecord::Base
  SERVICE_COMMISSION = 0.15
  STRIPE_PERCENTAGE_COMMISSION = 0.029
  STRIPE_FIXED_COMMISSION = 0.30

  ...

  def commission
    amount*SERVICE_COMMISSION - stripe_commission
  end

  private

  def stripe_commission
    amount*STRIPE_PERCENTAGE_COMMISSION + STRIPE_FIXED_COMMISSION
  end
end
```

As soon as you integrate with a new payment method, you won't be able to scale this functionality inside this model.

Also, as soon as you start to integrate more business logic, your `Order` object will start to lose [cohesion](#).

Prefer domain objects, with the calculation of the commission completely abstracted from the responsibility of persisting orders:

```ruby
# app/models/order.rb
class Order < ActiveRecord::Base
  ...
  # No reference to commission calculation
end

# lib/commission.rb
class Commission
  SERVICE_COMMISSION = 0.15

  def self.calculate(payment_method, model)
    model.amount*SERVICE_COMMISSION - payment_commission(payment_method, model)
  end

  private

  def self.payment_commission(payment_method, model)
    # There are better ways to implement a static registry,
    # this is only for illustration purposes.
    Object.const_get("#{payment_method}Commission").calculate(model)
  end
end

# lib/stripe_commission.rb
class StripeCommission
  STRIPE_PERCENTAGE_COMMISSION = 0.029
  STRIPE_FIXED_COMMISSION = 0.30
```

```
  def self.calculate(model)
    model.amount*STRIPE_PERCENTAGE_COMMISSION
      + STRIPE_PERCENTAGE_COMMISSION
  end
end

# app/controllers/orders_controller.rb
class OrdersController < ApplicationController
  def create
    @order = Order.new(order_params)
    @order.commission = Commission.calculate("Stripe", @order)
    ...
  end
end
```

Using domain objects has the following architectural advantages:

- it's extremely easy to unit test, as no fixtures or factories are required to instantiate the objects with the logic.
- works with everything that accepts the message `amount`.
- keeps each domain object small, with clearly defined responsibilities, and with higher cohesion.
- easily scales with new payment methods by [addition, not modification](#).
- stops the tendency to have an ever-growing `User` object in each Ruby on Rails application.

I personally like to put domain objects in `lib`. If you do so, remember to add it to `autoload_paths`:

```
# config/application.rb
config.autoload_paths << Rails.root.join('lib')
```

You may also prefer to create domain objects more action-oriented, following the Command/Query pattern. In such case, putting these objects in `app/commands` might be a better place as all `app` subdirectories are automatically added to the autoload path.

Section 6.3: Beware of default_scope

ActiveRecord includes `default_scope`, to automatically scope a model by default.

```
class Post
  default_scope ->{ where(published: true).order(created_at: :desc) }
end
```

The above code will serve posts which are already published when you perform any query on the model.

```
Post.all # will only list published posts
```

That scope, while innocuous-looking, has multiple hidden side-effect that you may not want.

default_scope and order

Since you declared an `order` in the `default_scope`, calling `order` on `Post` will be added as additional orders instead of overriding the default.

```
Post.order(updated_at: :desc)

SELECT "posts".* FROM "posts" WHERE "posts"."published" = 't' ORDER BY "posts"."created_at" DESC,
"posts"."updated_at" DESC
```

This is probably not the behavior you wanted; you can override this by excluding the `order` from the scope first

```
Post.except(:order).order(updated_at: :desc)
```

```
SELECT "posts".* FROM "posts" WHERE "posts"."published" = 't' ORDER BY "posts"."updated_at" DESC
```

default_scope and model initialization

As with any other **ActiveRecord::Relation**, default_scope will alter the default state of models initialized from it.

In the above example, Post has where(published: **true**) set by default, and so new models from Post will also have it set.

```
Post.new # => <Post published: true>
```

unscoped

default_scope can nominally be cleared by calling unscoped first, but this also has side-effects. Take, for example, an STI model:

```
class Post < Document
  default_scope ->{ where(published: true).order(created_at: :desc) }
end
```

By default, queries against Post will be scoped to type columns containing 'Post'. But unscoped will clear this along with your own default_scope, so if you use unscoped you have to remember to account for this as well.

```
Post.unscoped.where(type: 'Post').order(updated_at: :desc)
```

unscoped and Model Associations

Consider a relationship between Post and User

```
class Post < ApplicationRecord
  belongs_to :user
  default_scope ->{ where(published: true).order(created_at: :desc) }
end

class User < ApplicationRecord
  has_many :posts
end
```

By getting an individual User, you can see the posts related to it:

```
user = User.find(1)
user.posts
```

```
SELECT "posts".* FROM "posts" WHERE "posts"."published" = 't' AND "posts"."user_id" = ? ORDER BY
"posts"."created_at" DESC [["user_id", 1]]
```

But you want to clear the default_scope from the posts relation, so you use unscoped

```
user.posts.unscoped
```

```
SELECT "posts".* FROM "posts"
```

This wipes out the user_id condition as well as the default_scope.

An example use-case for default_scope

Despite all of that, there are situations where using default_scope is justifiable.

Consider a multi-tenant system where multiple subdomains are served from the same application but with isolated data. One way to achieve this isolation is through `default_scope`. The downsides in other cases become upsides here.

```ruby
class ApplicationRecord < ActiveRecord::Base
  def self.inherited(subclass)
    super

    return unless subclass.superclass == self
    return unless subclass.column_names.include? 'tenant_id'

    subclass.class_eval do
      default_scope ->{ where(tenant_id: Tenant.current_id) }
    end
  end
end
```

All you need to do is set `Tenant.current_id` to something early in the request, and any table that contains `tenant_id` will automatically become scoped without any additional code. Instantiating records will automatically inherit the tenant id they were created under.

The important thing about this use-case is that the scope is set once per request, and it doesn't change. The only cases you will need `unscoped` here are special cases like background workers that run outside of a request scope.

Section 6.4: Convention Over Configuration

In Rails, you find yourself looking at *controllers, views, and models* for your database.

To reduce the need for heavy configuration, Rails implements rules to ease up working with the application. You may define your own rules but for the beginning (and for later on) it's a good idea to stick to conventions that Rails offers.

These conventions will speed up development, keep your code concise and readable, and allow you an easy navigation inside your application.

Conventions also lower the barriers to entry for beginners. There are so many conventions in Rails that a beginner doesn't even need to know about, but can just benefit from in ignorance. It's possible to create great applications without knowing why everything is the way it is.

For Example

If you have a database table called `orders` with the primary key `id`, the matching model is called `order` and the controller that handles all the logic is named `orders_controller`. The view is split in different actions: if the controller has a `new` and `edit` action, there is also a `new` and `edit` view.

For Example

To create an app you simply run `rails new app_name`. This will generate roughly 70 files and folders that comprise the infrastructure and foundation for your Rails app.

It includes:

- Folders to hold your models (database layer), controllers, and views
- Folders to hold unit tests for your application
- Folders to hold your web assets like Javascript and CSS files
- Default files for HTTP 400 responses (i.e. file not found)

- Many others

Section 6.5: Don't Repeat Yourself (DRY)

To help to maintain clean code, Rails follows the principle of DRY.

It involves whenever possible, re-using as much code as possible rather than duplicating similar code in multiple places (for example, using partials). This reduces *errors*, keeps your code *clean* and enforces the principle of *writing code once* and then reusing it. It is also easier and more efficient to update code in one place than to update multiple parts of the same code. Thus making your code more modular and robust.

Also *Fat Model, Skinny Controller* is DRY, because you write the code in your model and in the controller only do the call, like:

```ruby
# Post model
scope :unpublished, ->(timestamp = Time.now) { where('published_at IS NULL OR published_at > ?', timestamp) }


# Any controller
def index
    ....
    @unpublished_posts = Post.unpublished
    ....
end

def others
    ...
    @unpublished_posts = Post.unpublished
    ...
end
```

This also helps lead to an API driven structure where internal methods are hidden and changes are achieved through passing parameters in an API fashion.

Section 6.6: You Ain't Gonna Need it (YAGNI)

If you can say "YAGNI" (You ain't gonna need it) about a feature, you better not implement it. There can be a lot of development time saved through focussing onto simplicity. Implementing such features anyway can lead to problems:

Problems
Overengineering

If a product is more complicated than it has to be, it is over engineered. Usually these "not yet used" features will never be used in the intended way they were written and have to be refactored if they ever get used. Premature optimisations, especially performance optimisations, often lead to design decisions which will be proved wrong in the future.

Code Bloat

Code Bloat means unnecessary complicated code. This can occur for example by abstraction, redundancy or incorrect application of design patterns. The code base becomes difficult to understand, confusing and expensive to maintain.

Feature Creep

Feature Creep refers to adding new features that go beyond the core functionality of the product and lead to an unnecessarily high complexity of the product.

Long development time

The time which could be used to develop necessary features is spent to develop unnecessary features. The product takes longer to deliver.

Solutions
KISS - Keep it simple, stupid

According to KISS, most systems work the best if they are designed simple. Simplicity should be a primary design goal to reduce complexity. It can be achieved by following the "Single Responsibility Principle" for example.

YAGNI – You Ain't Gonna Need it

Less is more. Think about every feature, is it really needed? If you can think of any way that it's YAGNI, leave it away. It's better to develop it when it's needed.

Continuous Refactoring

The product is being improved steadily. With refactoring, we can make sure that the product is being done according to best practice and does not degenerate to a patch work.

Chapter 7: Naming Conventions

Section 7.1: Controllers

Controller class names are pluralized. The reason is the controller controls multiple instances of object instance.

For Example: `OrdersController` would be the controller for an `orders` table. Rails will then look for the class definition in a file called `orders_controller.rb` in the `/app/controllers` directory.

For Example: `PostsController` would be the controller for a `posts` table.

If the controller class name has multiple capitalized words, the table name is assumed to have underscores between these words.

For Example: If a controller is named `PendingOrdersController` then assumed file name for this controller will be `pending_orders_controller.rb`.

Section 7.2: Models

The model is named using the class naming convention of unbroken MixedCase and is always the singular of the table name.

For Example: If a table was named `orders`, the associated model would be named `Order`

For Example: If a table was named `posts`, the associated model would be named `Post`

Rails will then look for the class definition in a file called `order.rb` in the `/app/models` directory.

If the model class name has multiple capitalized words, the table name is assumed to have underscores between these words.

For Example: If a model is named `BlogPost` then assumed table name will be `blog_posts`.

Section 7.3: Filenames and autoloading

Rails files - and Ruby files in general - should be named with `lower_snake_case` filenames. E.g.

```
app/controllers/application_controller.rb
```

is the file that contains the `ApplicationController` class definition. Note that while `PascalCase` is used for class and module names, the files in which they reside should still be `lower_snake_case`.

Consistent naming is important since Rails makes use of auto-loading files as needed, and uses "inflection" to transform between different naming styles, such as transforming `application_controller` to `ApplicationController` and back again.

E.g. if Rails sees that the `BlogPost` class doesn't exist (hasn't been loaded yet), it'll look for a file named `blog_post.rb` and attempt to load that file.

It is therefore also important to name files for what they contain, since the autoloader expects file names to match content. If, for instance, the `blog_post.rb` instead contains a class named just `Post`, you'll see a **LoadError**: `Expected [some path]/blog_post.rb to define BlogPost.`

If you add a dir under `app/something/` (e.g. /models/products/), and

- want to namespace modules and classes inside new dir then you don't need to do anything and it'll be loaded itself. For example, in app`/models/products/` `you would need to wrap your` `class in`module Products`.
- don't want to namespace modules and classes inside my new dir then you have to add `config.`autoload_paths `+=` `%W(` `#{config.root}/app/models/products )` to your application.`rb` to autoload.

One more thing to pay attention to (especially if English is not your first language) is the fact that Rails accounts for irregular plural nouns in English. So if you have model named "Foot" the corresponding controller needs to be called "FeetController" rather than "FootsController" if you want rails "magic" routing (and many more such features) to work.

Section 7.4: Views and Layouts

When a controller action is rendered, Rails will attempt to find a matching layout and view based on the name of the controller.

Views and layouts are placed in the app`/`views directory.

Given a request to the PeopleController`#index` action, Rails will search for:

- the layout called people in app`/`views`/`layouts`/` (or application if no match is found)
- a view called index.`html`.`erb` in app`/`views`/`people`/` by default
- if you wish to render other file called index_new.`html`.`erb` you have to write code for that in PeopleController`#index` action like render `'index_new'`
- we can set different layouts for every action by writing render `'index_new'`, layout: `'your_layout_name'`

Section 7.5: Models class from Controller name

You can get a Model class from a Controller name this way (context is Controller class):

```
class MyModelController < ActionController::Base

  # Returns corresponding model class for this controller
  # @return [ActiveRecord::Base]
  def corresponding_model_class
    # ... add some validation
    controller_name.classify.constantize
  end
end
```

Chapter 8: ActionCable

Section 8.1: User Authentication

```ruby
# app/channels/application_cable/connection.rb
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
      self.current_user = find_verified_user
      logger.add_tags 'ActionCable', current_user.id
      # Can replace current_user.id with usernames, ids, emails etc.
    end

    protected

    def find_verified_user
      if verified_user = env['warden'].user
        verified_user
      else
        reject_unauthorized_connection
      end
    end
  end
end
```

Section 8.2: [Basic] Server Side

```ruby
# app/channels/appearance_channel.rb
class NotificationsChannel < ApplicationCable::Channel
  def subscribed
    stream_from "notifications"
  end

  def unsubscribed
  end

  def notify(data)
    ActionCable.server.broadcast "notifications", { title: 'New things!', body: data }
  end
end
```

Section 8.3: [Basic] Client Side (Coffeescript)

app/assets/javascripts/channels/notifications.coffee

```coffeescript
App.notifications = App.cable.subscriptions.create "NotificationsChannel",
  connected: ->
    # Called when the subscription is ready for use on the server
    $(document).on "change", "input", (e)=>
      @notify(e.target.value)

  disconnected: ->
    # Called when the subscription has been terminated by the server
    $(document).off "change", "input"

  received: (data) ->
    # Called when there's incoming data on the websocket for this channel
```

```
      $('body').append(data)

  notify: (data)->
    @perform('notify', data: data)
```

app/assets/javascripts/application.js # usually generated like this

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require_tree .
```

app/assets/javascripts/cable.js # usually generated like this

```
//= require action_cable
//= require_self
//= require_tree ./channels

(function() {
  this.App || (this.App = {});

  App.cable = ActionCable.createConsumer();

}).call(this);
```

Chapter 9: ActiveModel

Section 9.1: Using ActiveModel::Validations

You can validate any object, even plain ruby.

```ruby
class User
  include ActiveModel::Validations

  attr_reader :name, :age

  def initialize(name, age)
    @name = name
    @age  = age
  end

  validates :name, presence: true
  validates :age, numericality: { only_integer: true, greater_than: 12 }
end

User.new('John Smith', 28).valid? #=> true
User.new('Jane Smith', 11).valid? #=> false
User.new(nil, 30).valid?          #=> false
```

Chapter 10: User Authentication in Rails

Devise is a very powerful gem, it allows you to sign up, sign in and sign out options just after installing. Moreover user can add authentications and restrictions to its applications. Devise also come with its own views, if user wants to use. A user can also customize sign up and sign in forms according to its need and requirement. It should be noted that Devise recommends that you implement your own login if you're new to rails.

Section 10.1: Authentication using Devise

Add gem to the Gemfile:

```
gem 'devise'
```

Then run the `bundle install` command.

Use command `$ rails generate devise:install` to generate required configuration file.

Set up the default URL options for the Devise mailer in each environment In development environment add this line:

```
config.action_mailer.default_url_options = { host: 'localhost', port: 3000 }
```

to your `config/environments/development.rb`

similarly in production this edit `config/environments/production.rb` file and add

```
config.action_mailer.default_url_options = { host: 'your-site-url'}
```

Then create a model using:`$ rails generate devise` USER Where USER is the class name for which you want to implement authentication.

Finally, run: `rake db:migrate` and you are all set.

Custom views

If you need to configure your views, you can use the `rails generate devise:views` generator that will copy all views to your application. Then you can edit them as desired.

If you have more than one Devise model in your application (for example User and Admin), you will notice that Devise uses the same views for all models. Devise offers an easy way to customize views. Set `config.scoped_views = true` inside the `config/initializers/devise.rb` file.

You can also use the generator to create scoped views: `rails generate devise:views users`

If you would like to generate only a few sets of views, such as the ones for the registerable and confirmable module use the -v flag: `rails generate devise:views -v registrations confirmations`

Section 10.2: Devise Controller Filters & Helpers

To set up a controller with user authentication using devise, add this before_action: (assuming your devise model is 'User'):

```
before_action :authenticate_user!
```

To verify if a user is signed in, use the following helper:

```
user_signed_in?
```

For the current signed-in user, use this helper:

```
current_user
```

You can access the session for this scope:

```
user_session
```

- Note that if your Devise model is called `Member` instead of `User`, replace `user` above with `member`

Section 10.3: Omniauth

First choose your auth strategy and add it to your `Gemfile`. You can find a list of strategies here:
https://github.com/intridea/omniauth/wiki/List-of-Strategies

```ruby
gem 'omniauth-github', :github => 'intridea/omniauth-github'
gem 'omniauth-openid', :github => 'intridea/omniauth-openid'
```

You can add this to your rails middleware like so:

```ruby
Rails.application.config.middleware.use OmniAuth::Builder do
  require 'openid/store/filesystem'
  provider :github, ENV['GITHUB_KEY'], ENV['GITHUB_SECRET']
  provider :openid, :store => OpenID::Store::Filesystem.new('/tmp')
end
```

By default, OmniAuth will add **/**auth**/**:provider to your routes and you can start by using these paths.

By default, if there is a failure, omniauth will redirect to **/**auth**/**failure

Section 10.4: has_secure_password

Create User Model

```
rails generate model User email:string password_digest:string
```

Add has_secure_password module to User model
class User < ActiveRecord::Base has_secure_password end

Now you can create a new user with password

```ruby
user = User.new email: 'bob@bob.com', password: 'Password1', password_confirmation: 'Password1'
```

Verify password with authenticate method

```ruby
user.authenticate('somepassword')
```

Section 10.5: has_secure_token

Create User Model

```ruby
# Schema: User(token:string, auth_token:string)
```

```
class User < ActiveRecord::Base
  has_secure_token
  has_secure_token :auth_token
end
```

Now when you create a new user a token and auth_token are automatically generated

```
user = User.new
user.save
user.token # => "pX27zsMN2ViQKta1bGfLmVJE"
user.auth_token # => "77TMHrHJFvFDwodq8w7Ev2m7"
```

You can update the tokens using `regenerate_token` and `regenerate_auth_token`

```
user.regenerate_token # => true
user.regenerate_auth_token # => true
```

Chapter 11: ActiveRecord Associations

Section 11.1: Polymorphic association

This type of association allows an ActiveRecord model to belong to more than one kind of model record. Common example:

```
class Human < ActiveRecord::Base
  has_one :address, :as => :addressable
end

class Company < ActiveRecord::Base
  has_one :address, :as => :addressable
end

class Address < ActiveRecord::Base
  belongs_to :addressable, :polymorphic => true
end
```

Without this association, you'd have all these foreign keys in your Address table but you only would ever have a value for one of them because an address, in this scenario, can only belong to one entity (Human or Company). Here is what it would look like:

```
class Address < ActiveRecord::Base
  belongs_to :human
  belongs_to :company
end
```

Section 11.2: Self-Referential Association

Self-referential association is used to associate a model with itself. The most frequent example would be, to manage association between a friend and his follower.

ex.

```
rails g model friendship user_id:references friend_id:integer
```

now you can associate models like;

```
class User < ActiveRecord::Base
  has_many :friendships
  has_many :friends, :through => :friendships
  has_many :inverse_friendships, :class_name => "Friendship", :foreign_key => "friend_id"
  has_many :inverse_friends, :through => :inverse_friendships, :source => :user
end
```

and the other model will look like;

```
class Friendship < ActiveRecord::Base
  belongs_to :user
  belongs_to :friend, :class_name => "User"
end
```

Section 11.3: belongs_to

A `belongs_to` association sets up a one-to-one connection with another model, so each instance of the declaring model "belongs to" one instance of the other model.

For example, if your application includes users and posts, and each post can be assigned to exactly one user, you'd declare the post model this way:

```ruby
class Post < ApplicationRecord
  belongs_to :user
end
```

In your table structure you might then have

```ruby
create_table "posts", force: :cascade do |t|
  t.integer  "user_id",  limit: 4
end
```

Section 11.4: has_one

A `has_one` association sets up a one-to-one connection with another model, but with different semantics. This association indicates that each instance of a model contains or possesses one instance of another model.

For example, if each user in your application has only one account, you'd declare the user model like this:

```ruby
class User < ApplicationRecord
  has_one :account
end
```

In Active Record, when you have a `has_one` relation, active record ensures that the only one record exists with the foreign key.

Here in our example: In accounts table, there can only be one record with a particular user_id. If you try to associate one more account for the same user, it makes the previous entry's foreign key as null(making it orphan) and creates a new one automatically. It makes the previous entry null even if the save fails for the new entry to maintain consistency.

```ruby
user = User.first
user.build_account(name: "sample")
user.save   [Saves it successfully, and creates an entry in accounts table with user_id 1]
user.build_account(name: "sample1")  [automatically makes the previous entry's foreign key null]
user.save  [creates the new account with name sample 1 and user_id 1]
```

Section 11.5: has_many

A `has_many` association indicates a one-to-many connection with another model. This association generally is located on the other side of a belongs_to association.

This association indicates that each instance of the model has zero or more instances of another model.

For example, in an application containing users and posts, the user model could be declared like this:

```ruby
class User < ApplicationRecord
  has_many :posts
```

```
  end
```

The table structure of `Post` would remain the same as in the `belongs_to` example; in contrast, `User` would not require any schema changes.

If you want to get the list of all the published posts for the `User`, then you can add the following (i.e. you can add scopes to your association objects):

```
class User < ApplicationRecord
  has_many :published_posts, -> { where("posts.published IS TRUE") }, class_name: "Post"
end
```

Section 11.6: The has_many :through association

A `has_many :through` association is often used to set up a `many-to-many` connection with another model. This association indicates that the declaring model can be matched with zero or more instances of another model by proceeding through a third model.

For example, consider a medical practice where patients make appointments to see physicians. The relevant association declarations could look like this:

```
class Physician < ApplicationRecord
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ApplicationRecord
  belongs_to :physician
  belongs_to :patient
end

class Patient < ApplicationRecord
  has_many :appointments
  has_many :physicians, through: :appointments
end
```

Section 11.7: The has_one :through association

A `has_one :through` association sets up a `one-to-one` connection with another model. This association indicates that the declaring model can be matched with one instance of another model by proceeding through a third model.

For example, if each `supplier` has one `account`, and each account is associated with one account history, then the supplier model could look like this:

```
class Supplier < ApplicationRecord
  has_one :account
  has_one :account_history, through: :account
end

class Account < ApplicationRecord
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ApplicationRecord
  belongs_to :account
```

Section 11.8: The has_and_belongs_to_many association

A `has_and_belongs_to_many` association creates a direct `many`-`to`-`many` connection with another model, with no intervening model.

For example, if your application includes `assemblies` and `parts`, with each assembly having many parts and each part appearing in many assemblies, you could declare the models this way:

```ruby
class Assembly < ApplicationRecord
  has_and_belongs_to_many :parts
end

class Part < ApplicationRecord
  has_and_belongs_to_many :assemblies
end
```

Chapter 12: ActiveRecord Validations

Section 12.1: Validating length of an attribute

```ruby
class Person < ApplicationRecord
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

The possible length constraint options are:

- `:minimum` - The attribute cannot have less than the specified length.
- `:maximum` - The attribute cannot have more than the specified length.
- `:in` (or `:within`) - The attribute length must be included in a given interval. The value for this option must be a range.
- `:is` - The attribute length must be equal to the given value.

Section 12.2: Validates format of an attribute

Validate that an attribute's value matches a regular expression using `format` and the `with` option.

```ruby
class User < ApplicationRecord
  validates :name, format: { with: /\A\w{6,10}\z/ }
end
```

You can also define a constant and set its value to a regular expression and pass it to the `with:` option. This might be more convenient for really complex regular expressions

```ruby
PHONE_REGEX = /\A\(\d{3}\)\d{3}-\d{4}\z/
validates :phone, format: { with: PHONE_REGEX }
```

The default error message is `is invalid`. This can be changed with the `:message` option.

```ruby
validates :bio, format: { with: /\A\D+\z/, message: "Numbers are not allowed" }
```

The reverse also replies, and you can specify that a value should *not* match a regular expression with the `without:` option

Section 12.3: Validating presence of an attribute

This helper validates that the specified attributes are not empty.

```ruby
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: "John").valid? # => true
Person.create(name: nil).valid? # => false
```

You can use the absence helper to validate that the specified attributes are absent. It uses the `present?` method to check for nil or empty values.

```
class Person < ApplicationRecord
  validates :name, :login, :email, absence: true
end
```

Note: In case the attribute is a `boolean` one, you cannot make use of the usual presence validation (the attribute would not be valid for a `false` value). You can get this done by using an inclusion validation:

```
validates :attribute, inclusion: [true, false]
```

Section 12.4: Custom validations

You can add your own validations adding new classes inheriting from `ActiveModel::Validator` or from `ActiveModel::EachValidator`. Both methods are similar but they work in a slightly different ways:

`ActiveModel::Validator` **and** `validates_with`

Implement the `validate` method which takes a record as an argument and performs the validation on it. Then use `validates_with` with the class on the model.

```
# app/validators/starts_with_a_validator.rb
class StartsWithAValidator < ActiveModel::Validator
  def validate(record)
    unless record.name.starts_with? 'A'
      record.errors[:name] << 'Need a name starting with A please!'
    end
  end
end

class Person < ApplicationRecord
  validates_with StartsWithAValidator
end
```

`ActiveModel::EachValidator` **and** `validate`

If you prefer to use your new validator using the common `validate` method on a single param, create a class inheriting from `ActiveModel::EachValidator` and implement the `validate_each` method which takes three arguments: `record`, `attribute`, and `value`:

```
class EmailValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    unless value =~ /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
      record.errors[attribute] << (options[:message] || 'is not an email')
    end
  end
end

class Person < ApplicationRecord
  validates :email, presence: true, email: true
end
```

More information on the [Rails guides](#).

Section 12.5: Validates inclusion of an attribute

You can check if a value is included in an array using the `inclusion:` helper. The `:in` option and its alias, `:within` show the set of acceptable values.

```
class Country < ApplicationRecord
  validates :continent, inclusion: { in: %w(Africa Antartica Asia Australia
                                            Europe North America South America) }
end
```

To check if a value is not included in an array, use the `exclusion:` helper

```
class User < ApplicationRecord
  validates :name, exclusion: { in: %w(admin administrator owner) }
end
```

Section 12.6: Grouping validation

Sometimes it is useful to have multiple validations use one condition. It can be easily achieved using with_options.

```
class User < ApplicationRecord
  with_options if: :is_admin? do |admin|
    admin.validates :password, length: { minimum: 10 }
    admin.validates :email, presence: true
  end
end
```

All validations inside of the with_options block will have automatically passed the condition if: :is_admin?

Section 12.7: Validating numericality of an attribute

This validation restricts the insertion of only numeric values.

```
class Player < ApplicationRecord
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```

Besides `:only_integer`, this helper also accepts the following options to add constraints to acceptable values:

- `:greater_than` - Specifies the value must be greater than the supplied value. The default error message for this option is "must be greater than %{count}".
- `:greater_than_or_equal_to` - Specifies the value must be greater than or equal to the supplied value. The default error message for this option is "must be greater than or equal to %{count}".
- `:equal_to` - Specifies the value must be equal to the supplied value. The default error message for this option is "must be equal to %{count}".
- `:less_than` - Specifies the value must be less than the supplied value. The default error message for this option is "must be less than %{count}".
- `:less_than_or_equal_to` - Specifies the value must be less than or equal to the supplied value. The default error message for this option is "must be less than or equal to %{count}".
- `:other_than` - Specifies the value must be other than the supplied value. The default error message for this option is "must be other than %{count}".
- `:odd` - Specifies the value must be an odd number if set to true. The default error message for this option is "must be odd".
- `:even` - Specifies the value must be an even number if set to true. The default error message for this option is "must be even".

By default, numericality doesn't allow nil values. You can use allow_nil: true option to permit it.

Section 12.8: Validate uniqueness of an attribute

This helper validates that the attribute's value is unique right before the object gets saved.

```ruby
class Account < ApplicationRecord
  validates :email, uniqueness: true
end
```

There is a `:scope` option that you can use to specify one or more attributes that are used to limit the uniqueness check:

```ruby
class Holiday < ApplicationRecord
  validates :name, uniqueness: { scope: :year,
    message: "should happen once per year" }
end
```

There is also a `:case_sensitive` option that you can use to define whether the uniqueness constraint will be case sensitive or not. This option defaults to `true`.

```ruby
class Person < ApplicationRecord
  validates :name, uniqueness: { case_sensitive: false }
end
```

Section 12.9: Skipping Validations

Use following methods if you want to skip the validations. These methods will save the object to the database even if it is invalid.

- decrement!
- decrement_counter
- increment!
- increment_counter
- toggle!
- touch
- update_all
- update_attribute
- update_column
- update_columns
- update_counters

You can also skip validation while saving by passing `validate` as an argument to `save`

```ruby
User.save(validate: false)
```

Section 12.10: Confirmation of attribute

You should use this when you have two text fields that should receive exactly the same content. For example, you may want to confirm an email address or a password. This validation creates a ***virtual*** attribute whose name is the name of the field that has to be confirmed with `_confirmation` appended.

```ruby
class Person < ApplicationRecord
  validates :email, confirmation: true
end
```

Note This check is performed only if `email_confirmation` is not nil.

To require confirmation, make sure to add a presence check for the confirmation attribute.

```
class Person < ApplicationRecord
  validates :email,        confirmation: true
  validates :email_confirmation, presence: true
end
```

Source

Section 12.11: Using :on option

The `:on` option lets you specify when the validation should happen. The default behavior for all the built-in validation helpers is to be run on save (both when you're creating a new record and when you're updating it).

```
class Person < ApplicationRecord
  # it will be possible to update email with a duplicated value
  validates :email, uniqueness: true, on: :create

  # it will be possible to create the record with a non-numerical age
  validates :age, numericality: true, on: :update

  # the default (validates on both create and update)
  validates :name, presence: true
end
```

Section 12.12: Conditional validation

Sometimes you may need to validate record only under certain conditions.

```
class User < ApplicationRecord
  validates :name, presence: true, if: :admin?

  def admin?
    conditional here that returns boolean value
  end
end
```

If you conditional is really small, you can use a Proc:

```
class User < ApplicationRecord
  validates :first_name, presence: true, if: Proc.new { |user| user.last_name.blank? }
end
```

For negative conditional you can use **unless**:

```
class User < ApplicationRecord
  validates :first_name, presence: true, unless: Proc.new { |user| user.last_name.present? }
end
```

You can also pass a string, which will be executed via `instance_eval`:

```
class User < ApplicationRecord
  validates :first_name, presence: true, if: 'last_name.blank?'
end
```

Chapter 13: ActiveRecord Query Interface

ActiveRecord is the M in MVC which is the layer of the system responsible for representing business data and logic. The technique that connects the rich objects of an application to tables in a relational database management system is **O**bject **R**elational **M**apper(**ORM**).

ActiveRecord will perform queries on the database for you and is compatible with most database systems. Regardless of which database system you're using, the ActiveRecord method format will always be the same.

Section 13.1: .where

The `where` method is available on any `ActiveRecord` model and allows querying the database for a set of records matching the given criteria.

The `where` method accepts a hash where the keys correspond to the column names on the table that the model represents.

As a simple example, we will use the following model:

```
class Person < ActiveRecord::Base
  #attribute :first_name, :string
  #attribute :last_name, :string
end
```

To find all people with the first name of `Sven`:

```
people = Person.where(first_name: 'Sven')
people.to_sql # "SELECT * FROM people WHERE first_name='Sven'"
```

To find all people with the first name of `Sven` and last name of `Schrodinger`:

```
people = Person.where(first_name: 'Sven', last_name: 'Schrodinger')
people.to_sql # "SELECT * FROM people WHERE first_name='Sven' AND last_name='Schrodinger'"
```

In the above example, the sql output shows that records will only be returned if both the `first_name` and the `last_name` match.

query with OR condition

To find records with `first_name == 'Bruce'` OR `last_name == 'Wayne'`

```
User.where('first_name = ? or last_name = ?', 'Bruce', 'Wayne')
# SELECT "users".* FROM "users" WHERE (first_name = 'Bruce' or last_name = 'Wayne')
```

Section 13.2: .where with an array

The `where` method on any ActiveRecord model can be used to generate SQL of the form `WHERE column_name IN (a, b, c, ...)`. This is achieved by passing an array as argument.

As a simple example, we will use the following model:

```
class Person < ActiveRecord::Base
  #attribute :first_name, :string
  #attribute :last_name, :string
```

```ruby
people = Person.where(first_name: ['Mark', 'Mary'])
people.to_sql # "SELECT * FROM people WHERE first_name IN ('Mark', 'Mary')"
```

If the array contains a **nil**, the SQL will be modified to check if the column is null:

```ruby
people = Person.where(first_name: ['Mark', 'Mary', nil])
people.to_sql # "SELECT * FROM people WHERE first_name IN ('Mark', 'Mary') OR first_name IS NULL"
```

Section 13.3: Scopes

Scopes act as predefined filters on `ActiveRecord` models.

A scope is defined using the `scope` class method.

As a simple example, we will use the following model:

```ruby
class Person < ActiveRecord::Base
  #attribute :first_name, :string
  #attribute :last_name, :string
  #attribute :age, :integer

  # define a scope to get all people under 17
  scope :minors, -> { where(age: 0..17) }

  # define a scope to search a person by last name
  scope :with_last_name, ->(name) { where(last_name: name) }

end
```

Scopes can be called directly off the model class:

```ruby
minors = Person.minors
```

Scopes can be chained:

```ruby
peters_children = Person.minors.with_last_name('Peters')
```

The `where` method and other query type methods can also be chained:

```ruby
mary_smith = Person.with_last_name('Smith').where(first_name: 'Mary')
```

Behind the scenes, scopes are simply syntactic sugar for a standard class method. For example, these methods are functionally identical:

```ruby
scope :with_last_name, ->(name) { where(name: name) }

# This ^ is the same as this:

def self.with_last_name(name)
  where(name: name)
end
```

Default Scope

in your model to set a default scope for all operations on the model.

There is one notable difference between the `scope` method and a class method: `scope`-defined scopes will *always* return an **ActiveRecord::Relation**, even if the logic within returns nil. Class methods, however, have no such safety net and can break chainability if they return something else.

Section 13.4: Get first and last record

Rails have very easy way to get `first` and `last` record from database.

To get the `first` record from `users` table we need to type following command:

```
User.first
```

It will generate following `sql` query:

```sql
SELECT  `users`.* FROM `users`  ORDER BY `users`.`id` ASC LIMIT 1
```

And will return following record:

```
#<User:0x007f8a6db09920 id: 1, first_name: foo, created_at: Thu, 16 Jun 2016 21:43:03 UTC +00:00,
updated_at: Thu, 16 Jun 2016 21:43:03 UTC +00:00 >
```

To get the `last` record from `users` table we need to type following command:

```
User.last
```

It will generate following `sql` query:

```sql
SELECT  `users`.* FROM `users`  ORDER BY `users`.`id` DESC LIMIT 1
```

And will return following record:

```
#<User:0x007f8a6db09920 id: 10, first_name: bar, created_at: Thu, 16 Jun 2016 21:43:03 UTC +00:00,
updated_at: Thu, 16 Jun 2016 21:43:03 UTC +00:00 >
```

Passing an integer to **first** and **last** method creates a **LIMIT** query and returns array of objects.

```
User.first(5)
```

It will generate following `sql` query.

```sql
SELECT  "users".* FROM "users"  ORDER BY "users"."id" ASC LIMIT 5
```

And

```
User.last(5)
```

It will generate following `sql` query.

```sql
SELECT  "users".* FROM "users"  ORDER BY "users"."id" DESC LIMIT 5
```

Section 13.5: Ordering

You can order **ActiveRecord** query results by using **.order**:

```
User.order(:created_at)
#=> => [#<User id: 2, created_at: "2015-08-12 21:36:23">, #<User id: 11, created_at: "2015-08-15
10:21:48">]
```

If not specified, ordering will be performed in ascending order. You can specify it by doing:

```
User.order(created_at: :asc)
#=> => [#<User id: 2, created_at: "2015-08-12 21:36:23">, #<User id: 11, created_at: "2015-08-15
10:21:48">]

User.order(created_at: :desc)
#=> [#<User id: 7585, created_at: "2016-07-13 17:15:27">, #<User id: 7583, created_at: "2016-07-13
16:51:18">]
```

.order also accepts a string, so you could also do

```
User.order("created_at DESC")
#=> [#<User id: 7585, created_at: "2016-07-13 17:15:27">, #<User id: 7583, created_at: "2016-07-13
16:51:18">]
```

As the string is raw SQL, you can also specify a table and not only an attribute. Assuming you want to order users according to their role name, you can do this:

```
Class User < ActiveRecord::Base
    belongs_to :role
end

Class Role < ActiveRecord::Base
  has_many :users
end

User.includes(:role).order("roles.name ASC")
```

The order scope can also accept an Arel node:

```
User.includes(:role).order(User.arel_table[:name].asc)
```

Section 13.6: where.not

where clauses can be negated using the where.**not** syntax:

```
class Person < ApplicationRecord
  #attribute :first_name, :string
end

people = Person.where.not(first_name: ['Mark', 'Mary'])
# => SELECT "people".* FROM "people" WHERE "people"."first_name" NOT IN ('Mark', 'Mary')
```

Supported by ActiveRecord 4.0 and later.

Section 13.7: Includes

ActiveRecord with `includes` ensures that all of the specified associations are loaded using the minimum possible number of queries. So when querying a table for data with an associated table, both tables are loaded into memory.

```ruby
@authors = Author.includes(:books).where(books: { bestseller: true } )

# this will print  results without additional db hitting
@authors.each do |author|
  author.books.each do |book|
    puts book.title
  end
end
```

`Author.joins(:books).where(books: { bestseller: true } )` will load only **authors** with conditions into memory **without loading books**. Use `joins` when additional information about nested associations isn't required.

```ruby
@authors = Author.joins(:books).where(books: { bestseller: true } )

# this will print results without additional queries
@authors.each { |author| puts author.name }

# this will print results with additional db queries
@authors.each do |author|
  author.books.each do |book|
    puts book.title
  end
end
```

Section 13.8: Joins

`joins()` allows you to join tables to your current model. For ex.

```ruby
User.joins(:posts)
```

will produce the following SQL query:

```sql
"SELECT "users".* FROM "users" INNER JOIN "posts" ON "posts"."user_id" = "users"."id""
```

Having table joined, you will have access to it:

```ruby
User.joins(:posts).where(posts: { title: "Hello world" })
```

Pay attention on plural form. If your relation is `:has_many`, then the `joins()` argument should be pluralized. Otherwise, use singular.

Nested `joins`:

```ruby
User.joins(posts: :images).where(images: { caption: 'First post' })
```

which will produce:

```sql
"SELECT "users".* FROM "users" INNER JOIN "posts" ON "posts"."user_id" = "users"."id" INNER JOIN "images" ON "images"."post_id" = "images"."id""
```

Section 13.9: Limit and Offset

You can use `limit` to tell the number of records to be fetched, and use `offset` to tell the number of records to skip before starting to return the records.

For Example

```
User.limit(3) #returns first three records
```

It will generate following sql query.

```
"SELECT  `users`.* FROM `users` LIMIT 3"
```

As offset is not mentioned in above query so it will return first three records.

```
User.limit(5).offset(30) #returns 5 records starting from 31th i.e from 31 to 35
```

It will generate following sql query.

```
"SELECT  `users`.* FROM `users` LIMIT 5 OFFSET 30"
```

Section 13.10: .find_by

You can find records by any field in your table using `find_by`.

So, if you have a `User` model with a `first_name` attribute you can do:

```
User.find_by(first_name: "John")
#=> #<User id: 2005, first_name: "John", last_name: "Smith">
```

Mind that `find_by` doesn't throw any exception by default. If the result is an empty set, it returns **nil** instead of `find`.

If the exception is needed may use `find_by!` that raises an **ActiveRecord::RecordNotFound** error like `find`.

Section 13.11: .delete_all

If you need to delete a lot of records quickly, **ActiveRecord** gives **.delete_all** method. to be called directly on a model, to delete all records in that table, or a collection. Beware though, as **.delete_all** does not instantiate any object hence does not provide any callback (`before_*` and `after_destroy` don't get triggered).

```
User.delete_all
#=> 39  <-- .delete_all return the number of rows deleted

User.where(name: "John").delete_all
```

Section 13.12: ActiveRecord case insensitive search

If you need to search an ActiveRecord model for similar values, you might be tempted to use `LIKE` or `ILIKE` but this isn't portable between database engines. Similarly, resorting to always downcasing or upcasing can create performance issues.

You can use ActiveRecord's underlying Arel `matches` method to do this in a safe way:

```ruby
addresses = Address.arel_table
Address.where(addresses[:address].matches("%street%"))
```

Arel will apply the appropriate LIKE or ILIKE construct for the database engine configured.

Section 13.13: .group and .count

We have a `Product` model and we want to group them by their `category`.

```ruby
Product.select(:category).group(:category)
```

This will query the database as follows:

```sql
SELECT "product"."category" FROM "product" GROUP BY "product"."category"
```

Make sure that the grouped field is also selected. Grouping is especially useful for counting the occurrence - in this case - of `categories`.

```ruby
Product.select(:category).group(:category).count
```

As the query shows, it will use the database for counting, which is much more efficient, than retrieving all record first and do the counting in the code:

```sql
SELECT COUNT("products"."category") AS count_categories, "products"."category" AS products_category
FROM "products" GROUP BY "products"."category"
```

Section 13.14: .distinct (or .uniq)

If you want to remove duplicates from a result, you can use `.distinct()`:

```ruby
Customers.select(:country).distinct
```

This queries the database as follows:

```sql
SELECT DISTINCT "customers"."country" FROM "customers"
```

`.uniq()` has the same effect. With Rails 5.0 it got deprecated and it will be removed from Rails with version 5.1. The reason is, that the word `unique` doesn't have the same meaning as distinct and it can be misleading. Furthermore `distinct` is closer to the SQL syntax.

Chapter 14: ActionMailer

Action Mailer allows you to send emails from your application using mailer classes and views. Mailers work very similarly to controllers. They inherit from ActionMailer::Base and live in app/mailers, and they have associated views that appear in app/views.

Section 14.1: Basic Mailer

This example uses four different files:

- The User model
- The User mailer
- The html template for the email
- The plain-text template for the email

In this case, the user model calls the approved method in the mailer and passes the post that has been approved (the approved method in the model may be called by a callback, from a controller method, etc). Then, the mailer generates the email from either the html or plain-text template using the information from the passed-in post (e.g. the title). By default, the mailer uses the template with the same name as the method in the mailer (which is why both the mailer method and the templates have the name 'approved').

user_mailer.rb

```ruby
class UserMailer < ActionMailer::Base
  default from: "donotreply@example.com"

  def approved(post)
    @title = post.title
    @user = post.user
    mail(to: @user.email, subject: "Your Post was Approved!")
  end
end
```

user.rb

```ruby
def approved(post)
    UserMailer.approved(post)
end
```

approved.html.erb

```erb
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Post Approved</title>
    </head>
    <body>
        <h2>Congrats <%= @user.name %>! Your post (#<%= @title %>) has been approved!</h2>
        <p>We look forward to your future posts!</p>
    </body>
</html>
```

approved.text.erb

```erb
Congrats <%= @user.name %>! Your post (#<%= @title %>) has been approved!
We look forward to your future posts!
```

Section 14.2: Generating a new mailer

To generate a new mailer, enter the following command

```
rails generate mailer PostMailer
```

This will generate a blank template file in app**/**mailers**/**post_mailer.**rb** named *PostMailer*

```ruby
class PostMailer < ApplicationMailer
end
```

Two layout files will also be generated for the email view, one for the html format and one for the text format.

If you prefer not to use the generator, you can create your own mailers. Make sure they inherit from `ActionMailer::Base`

Section 14.3: ActionMailer Interceptor

Action Mailer provides hooks into the interceptor methods. These allow you to register classes that are called during the mail delivery life cycle.

An interceptor class must implement the :delivering_email(message) method which will be called before the email is sent, allowing you to make modifications to the email before it hits the delivery agents. Your class should make any needed modifications directly to the passed in Mail::Message instance.

It can be useful for developers to send email to themselves not real users.

Example of registering an actionmailer interceptor:

```ruby
# config/initializers/override_mail_recipient.rb

if Rails.env.development? or Rails.env.test?
  class OverrideMailRecipient
    def self.delivering_email(mail)
      mail.subject = 'This is dummy subject'
      mail.bcc = 'test_bcc@noemail.com'
      mail.to = 'test@noemail.com'
    end
  end
  ActionMailer::Base.register_interceptor(OverrideMailRecipient)
end
```

Section 14.4: Adding Attachments

`ActionMailer` also allows attaching files.

```ruby
attachments['filename.jpg'] = File.read('/path/to/filename.jpg')
```

By default, attachments will be encoded with **Base64**. To change this, you can add a hash to the attachments method.

```ruby
attachments['filename.jpg'] = {
  mime_type: 'application/gzip',
  encoding: 'SpecialEncoding',
  content: encoded_content
```

```
  }
```

You can also add inline attachments

```
attachments.inline['image.jpg'] = File.read('/path/to/image.jpg')
```

Section 14.5: ActionMailer Callbacks

ActionMailer supports three callbacks

- before_action
- after_action
- around_action

Provide these in your Mailer class

```
class UserMailer < ApplicationMailer
  after_action :set_delivery_options, :prevent_delivery_to_guests, :set_business_headers
```

Then create these methods under the `private` keyword

```
private
  def set_delivery_options
  end

  def prevent_delivery_to_guests
  end

  def set_business_headers
  end
end
```

Section 14.6: Generate a Scheduled Newsletter

Create the **Newsletter** model:

```
rails g model Newsletter name:string email:string

subl app/models/newsletter.rb

validates :name, presence: true
validates :email, presence: true
```

Create the **Newsletter** controller:

```
rails g controller Newsletters create

class NewslettersController < ApplicationController
  skip_before_action :authenticate_user!
  before_action :set_newsletter, only: [:destroy]

  def create
    @newsletter = Newsletter.create(newsletter_params)
    if @newsletter.save
      redirect_to blog_index_path
    else
```

```
        redirect_to root_path
      end
    end

    private

      def set_newsletter
        @newsletter = Newsletter.find(params[:id])
      end

      def newsletter_params
        params.require(:newsletter).permit(:name, :email)
      end

  end
```

After that, change the **create.html.erb** view to the nex name. We will convert this file to and **partial view** which will be stored inside the **Footer**. The name will be **_form.html.erb**.

Change name file from: **To:**
app/views/newsletters/create.html.erb app/views/newsletters/_form.html.erb

After that set the routes:

```
subl app/config/routes.rb

resources :newsletters
```

Later on, we need to set the form we will use to save each mail:

```
subl app/views/newsletters/_form.html.erb

<%= form_for (Newsletter.new) do |f| %>
  <div class="col-md-12" style="margin: 0 auto; padding: 0;">
    <div class="col-md-6" style="padding: 0;">
      <%= f.text_field :name, class: 'form-control', placeholder:'Nombre' %>
    </div>
    <div class="col-md-6" style="padding: 0;">
      <%= f.text_field :email, class: 'form-control', placeholder:'Email' %>
    </div>
  </div>
  <div class="col-md-12" style="margin: 0 auto; padding:0;">
    <%= f.submit class:"col-md-12 tran3s s-color-bg hvr-shutter-out-horizontal", style:'border:
none; color: white; cursor: pointer; margin: 0.5em auto; padding: 0.75em; width: 100%;' %>
  </div>
<% end %>
```

And after that, insert on the footer:

```
subl app/views/layouts/_footer.html.erb

<%= render 'newsletters/form' %>
```

Now, install the -**letter_opener**- to can preview email in the default browser instead of sending it. This means you do not need to set up email delivery in your development environment, and you no longer need to worry about accidentally sending a test email to someone else's address.

First add the gem to your development environment and run the bundle command to install it.

```
subl your_project/Gemfile

gem "letter_opener", :group => :development
```

Then set the delivery method in the Development Environment:

```
subl your_project/app/config/environments/development.rb

config.action_mailer.delivery_method = :letter_opener
```

Now, create an **Mailer Structure** to manage the whole mailers which we will work. In terminal

```
rails generate mailer UserMailer newsletter_mailer
```

And inside the **UserMailer**, we have to create a method called **Newsletter Mailer** which will be created to contain inside on the lastest blog post and will be fired with a rake action. We will assume that you had a blog structure created before.

```
subl your_project/app/mailers/user_mailer.rb

class UserMailer 'your_gmail_account@gmail.com'

  def newsletter_mailer
    @newsletter = Newsletter.all
    @post = Post.last(3)
    emails = @newsletter.collect(&:email).join(", ")
    mail(to: emails, subject: "Hi, this is a test mail.")
  end

end
```

After that, create the **Mailer Template**:

```
subl your_project/app/views/user_mailer/newsletter_mailer.html.erb

<p> Dear Followers: </p>
<p> Those are the lastest entries to our blog. We invite you to read and share everything we did on
this week. </p>

<br/>
<table>
<% @post.each do |post| %>
  <%#= link_to blog_url(post) do %>
      <tr style="display:flex; float:left; clear:both;">
        <td style="display:flex; float:left; clear:both; height: 80px; width: 100px;">
          <% if post.cover_image.present? %>
            <%= image_tag post.cover_image.fullsize.url, class:"principal-home-image-slider" %>
          <%# else %>
            <%#= image_tag 'http://your_site_project.com' + post.cover_video, class:"principal-home-
image-slider" %>
            <%#= raw(video_embed(post.cover_video)) %>
          <% end %>
```

```
          </td>
          <td>
            <h3>
              <%= link_to post.title, :controller => "blog", :action => "show", :only_path => false,
:id => post.id %>
            </h3>
            <p><%= post.subtitle %></p>
          </td>
          <td style="display:flex; float:left; clear:both;">

          </td>
        </tr>
    <%# end %>
<% end %>
</table>
```

Since we want to send the email as a separate process, let's create a Rake task to fire off the email. Add a new file called email_tasks.rake to lib/tasks directory of your Rails application:

```
touch lib/taks/email_tasks.rake

desc 'weekly newsletter email'
task weekly_newsletter_email: :environment do
  UserMailer.newsletter_mailer.deliver!
end
```

The send_digest_email: :environment means to load the Rails environment before running the task, so you can access the application classes (like UserMailer) within the task.

Now, running the command rake -T will list the newly created Rake task. Test everything works by running the task and checking whether the email is sent or not.

To test if the mailer method works, run the rake command:

```
   rake weekly_newsletter_email
```

At this point, we have a working rake task which can be scheduled using **crontab**. So we will install the **Whenever Gem** which is used to provide a clear syntax for writing and deploying cron jobs.

```
subl your_project/Gemfile

  gem 'whenever', require: false
```

After that, run the next command to create an initial config/schedule.rb file for you (as long as the config folder is already present in your project).

```
   wheneverize .

   [add] writing `./config/schedule.rb'
   [done] wheneverized!
```

Now, inside the schedule file, we have to create our **CRON JOB** and call the mailer method inside determining the CRON JOB to operate some tasks without assistance and in a selected range of time. You can use different types of syntax as is explained on this link.

```
subl your_project/config/schedule.rb

every 1.day, :at => '4:30 am' do
  rake 'weekly_newsletter_email'
end
```



```
every 3.hours do # 1.minute 1.day 1.week 1.month 1.year is also supported
  runner "MyModel.some_process"
  rake "my:rake:task"
  command "/usr/bin/my_great_command"
end

every 1.day, :at => '4:30 am' do
  runner "MyModel.task_to_run_at_four_thirty_in_the_morning"
end

every :hour do # Many shortcuts available: :hour, :day, :month, :year, :reboot
  runner "SomeModel.ladeeda"
end

every :sunday, :at => '12pm' do # Use any day of the week or :weekend, :weekday
  runner "Task.do_something_great"
end

every '0 0 27-31 * *' do
  command "echo 'you can use raw cron syntax too'"
end

# run this task only on servers with the :app role in Capistrano
# see Capistrano roles section below
every :day, :at => '12:20am', :roles => [:app] do
  rake "app_server:task"
end
```

Now to test the **Cron Job** was succesfully created we can use the next command to read since terminal, our scheduled job in CRON SYNTAX:

```
your_project your_mac_user$ whenever
```

```
30 4 * * * /bin/bash -l -c 'cd /Users/your_mac_user/Desktop/your_project && RAILS_ENV=production
bundle exec rake weekly_newsletter_email --silent'
```

Now, to run the test in Development Environment, is wise to set the next line on the **application.rb** principal file to let the application knows where are the models it will use.

```
subl your_project/config/application.rb

config.action_mailer.default_url_options = { :host => "http://localhost:3000/" }
```

Now to let **Capistrano V3** save the new **Cron Job** inside the server and the trigger which will fired up the execution of this task, we have to add the next requirement:

```
subl your_project/Capfile

require 'whenever/capistrano'
```

And insert into the **deploy** file the identifier which **CRON JOB** will use about the **environment** and the name of the **application**.

```
subl your_project/config/deploy.rb

set :whenever_identifier, ->{ "#{fetch(:application)}_#{fetch(:rails_env)}" }
```

And ready, after save changes on each file, run the capistrano deploy command:

```
cap production deploy
```

And now your JOB was created and calendarize to run the Mailer Method which is what i want and in the range of time we set on this files.

Chapter 15: Rails generate commands

Parameter	Details
`-h`/`--help`	Get help on any generator command
`-p`/`--pretend`	Pretend Mode: Run generator but will not create or change any files
`field:type`	'field-name' is the name of the column to be created and 'type' is the data-type of column. The possible values for 'type' in `field:type` are given in the Remarks section.

Usage: `rails generate GENERATOR_NAME `**`[args]`**` `**`[options]`**`.`

Use `rails generate` to list available generators. Alias: `rails g`.

Section 15.1: Rails Generate Controller

we can create a new controller with `rails g controller` command.

```
$ bin/rails generate controller controller_name
```

The controller generator is expecting parameters in the form of `generate controller ControllerName action1 action2`.

The following creates a Greetings controller with an action of hello.

```
$ bin/rails generate controller Greetings hello
```

You will see the following output

```
create  app/controllers/greetings_controller.rb
 route  get "greetings/hello"
invoke  erb
create    app/views/greetings
create    app/views/greetings/hello.html.erb
invoke  test_unit
create    test/controllers/greetings_controller_test.rb
invoke  helper
create    app/helpers/greetings_helper.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/greetings.coffee
invoke    scss
create      app/assets/stylesheets/greetings.scss
```

This generates the following

File	Example
Controller File	`greetings_controller.rb`
View File	`hello.html.erb`
Functional Test File	`greetings_controller_test.rb`
View Helper	`greetings_helper.rb`
JavaScript File	`greetings.coffee`

It will also add routes for each action in `routes.rb`

Section 15.2: Rails Generate Migration

You can generate a rails migration file from the terminal using the following command:

```
rails generate migration NAME [field[:type][:index] field[:type][:index]] [options]
```

For a list of all the options supported by the command, you could run the command without any arguments as in `rails generate migration`.

For example, if you want to add `first_name` and `last_name` fields to `users` table, you can do:

```
rails generate migration AddNamesToUsers last_name:string first_name:string
```

Rails will create the following migration file:

```ruby
class AddNamesToUsers < ActiveRecord::Migration[5.0]
  def change
    add_column :users, :last_name, :string
    add_column :users, :first_name, :string
  end
end
```

Now, apply the pending migrations to the database by running the following in the terminal:

Version < 5.0
```
rake db:migrate
```
Version ≥ 5.0
```
rails db:migrate
```

> **Note:** For even less typing, you can replace `generate` with `g`.

Section 15.3: Rails Generate Scaffold

DISCLAIMER: Scaffolding is not recommended unless it's for very conventional CRUD apps/testing. This may generate a lot of files(views/models/controllers) that are not needed in your web application thus causing headaches(bad :().

To generate a fully working scaffold for a new object, including model, controller, views, assets, and tests, use the `rails g scaffold` command.

```
$ rails g scaffold Widget name:string price:decimal
    invoke  active_record
    create    db/migrate/20160722171221_create_widgets.rb
    create    app/models/widget.rb
    invoke    test_unit
    create      test/models/widget_test.rb
    create      test/fixtures/widgets.yml
    invoke  resource_route
     route    resources :widgets
    invoke  scaffold_controller
    create    app/controllers/widgets_controller.rb
    invoke    erb
    create      app/views/widgets
    create      app/views/widgets/index.html.erb
    create      app/views/widgets/edit.html.erb
```

```
create      app/views/widgets/show.html.erb
create      app/views/widgets/new.html.erb
create      app/views/widgets/_form.html.erb
invoke    test_unit
create      test/controllers/widgets_controller_test.rb
invoke    helper
create      app/helpers/widgets_helper.rb
invoke    jbuilder
create      app/views/widgets/index.json.jbuilder
create      app/views/widgets/show.json.jbuilder
invoke  assets
invoke    javascript
create      app/assets/javascripts/widgets.js
invoke    scss
create      app/assets/stylesheets/widgets.scss
```

Then you can run `rake db:migrate` to set up the database table.

Then you can visit http://localhost:3000/widgets and you'll see a fully functional CRUD scaffold.

Section 15.4: Rails Generate Model

To generate an `ActiveRecord` model that automagically creates the correct db migrations & boilerplate test files for your model, enter this command

```
rails generate model NAME column_name:column_type
```

'NAME' is the name of the model. 'field' is the name of the column in the DB table and 'type' is the column type (e.g. `name:string` or `body:text`). Check the Remarks section for a list of supported column types.

To setup foreign keys, add `belongs_to:model_name`.

So say you wanted to setup a `User` model that has a `username`, `email` and belongs to a `School`, you would type in the following

```
rails generate model User username:string email:string school:belongs_to
```

`rails g` is shorthand for `rails generate`. This would produce the same result

```
rails g model User username:string email:string school:belongs_to
```

Chapter 16: Configuration

Section 16.1: Custom configuration

Create a **YAML** file in the `config/` directory, for example: `config/neo4j.yml`

The content of `neo4j.yml` can be something like the below (for simplicity, `default` is used for all environments):

```yaml
default: &default
  host: localhost
  port: 7474
  username: neo4j
  password: root

development:
  <<: *default

test:
  <<: *default

production:
  <<: *default
```

in `config/application.rb`:

```ruby
module MyApp
  class Application < Rails::Application
    config.neo4j = config_for(:neo4j)
  end
end
```

Now, your custom config is accessible like below:

```ruby
Rails.configuration.neo4j['host']
 #=> localhost
Rails.configuration.neo4j['port']
 #=> 7474
```

More info

Rails official API document describes the `config_for` method as:

> Convenience for loading config/foo.yml for the current Rails env.

If you do not want to use a `yaml` file

You can configure your own code through the Rails configuration object with custom configuration under the `config.x` property.

Example

```ruby
config.x.payment_processing.schedule = :daily
config.x.payment_processing.retries  = 3
config.x.super_debugger = true
```

These configuration points are then available through the configuration object:

```
Rails.configuration.x.payment_processing.schedule # => :daily
Rails.configuration.x.payment_processing.retries  # => 3
Rails.configuration.x.super_debugger              # => true
Rails.configuration.x.super_debugger.not_set      # => nil
```

Chapter 17: I18n - Internationalization

Section 17.1: I18n with arguments

You can pass parameters to **I18n** t method:

```
# Example config/locales/en.yml
en:
  page:
    users: "%{users_count} users currently online"

# In models, controller, etc...
I18n.t('page.users', users_count: 12)

# In views

# ERB
<%= t('page.users', users_count: 12) %>

#SLIM
= t('page.users', users_count: 12)

# Shortcut in views - DRY!
# Use only the dot notation
# Important: Consider you have the following controller and view page#users

# ERB Example app/views/page/users.html.erb
<%= t('.users', users_count: 12) %>
```

And get the following output:

```
"12 users currently online"
```

Section 17.2: Translating ActiveRecord model attributes

globalize gem is a great solution to add translations to your `ActiveRecord` models. You can install it adding this to your `Gemfile`:

```
gem 'globalize', '~> 5.0.0'
```

If you're using `Rails 5` you will also need to add `activemodel-serializers-xml`

```
gem 'activemodel-serializers-xml'
```

Model translations allow you to translate your models' attribute values, for example:

```
class Post < ActiveRecord::Base
  translates :title, :text
end

I18n.locale = :en
post.title # => Globalize rocks!

I18n.locale = :he
post.title # => גלובאלייז 2שלט!
```

After you defined your model attributes that need to be translated you have to create a translation table, through a migration. **globalize** provides `create_translation_table!` and `drop_translation_table!`.

For this migration you need to use `up` and `down`, and **not** change. Also, in order to run this migration successfully, you have to define the translated attributes in your model first, like shown above. A proper migration for the previous `Post` model is this:

```ruby
class CreatePostsTranslationTable < ActiveRecord::Migration
  def up
    Post.create_translation_table! title: :string, text: :text
  end

  def down
    Post.drop_translation_table!
  end
end
```

You may also pass options for specific options, like:

```ruby
class CreatePostsTranslationTable < ActiveRecord::Migration
  def up
    Post.create_translation_table! title: :string,
      text: { type: :text, null: false, default: "Default text" }
  end

  def down
    Post.drop_translation_table!
  end
end
```

In case you already have any **existing data** in your needing translation columns, you can easily migrate it to the translations table, by adjusting your migration:

```ruby
class CreatePostsTranslationTable < ActiveRecord::Migration
  def up
    Post.create_translation_table!({
      title: :string,
      text: :text
    }, {
      migrate_data: true
    })
  end

  def down
    Post.drop_translation_table! migrate_data: true
  end
end
```

Make sure you drop the translated columns from the parent table after all your data is safely migrated. To automatically remove the translated columns from the parent table after the data migration, add the option `remove_source_columns` to the migration:

```ruby
class CreatePostsTranslationTable < ActiveRecord::Migration
  def up
    Post.create_translation_table!({
      title: :string,
      text: :text
```

```
    }, {
      migrate_data: true,
      remove_source_columns: true
    })

  end

  def down
    Post.drop_translation_table! migrate_data: true
  end
end
```

You may also add new fields to a previously created translations table:

```
class Post < ActiveRecord::Base
  # Remember to add your attribute here too.
  translates :title, :text, :author
end

class AddAuthorToPost < ActiveRecord::Migration
  def up
    Post.add_translation_fields! author: :text
  end

  def down
    remove_column :post_translations, :author
  end
end
```

Section 17.3: Get locale from HTTP request

Sometimes it can be useful to set your application locale based upon the request IP. You can easily achieve this using **Geocoder**. Among the many things **Geocoder** does, it can also tell the location of a request.

First, add **Geocoder** to your Gemfile

```
# Gemfile
gem 'geocoder'
```

Geocoder adds location and safe_location methods to the standard **Rack::Request** object so you can easily look up the location of any HTTP request by IP address. You can use this methods in a before_action in your ApplicationController:

```
class ApplicationController < ActionController::Base
  before_action :set_locale_from_request

  def set_locale_from_request
    country_code = request.location.data["country_code"] #=> "US"
    country_sym = country_code.underscore.to_sym #=> :us

    # If request locale is available use it, otherwise use I18n default locale
    if I18n.available_locales.include? country_sym
      I18n.locale = country_sym
    end
end
```

Beware that this will not work in development and test environments, as things like 0.0.0.0 and localhost are valid valid Internet IP addresses.

Limitations and alternatives

Geocoder is very powerful and flexible, but needs to be configured to work with a *geocoding service* (see more details); many of which place limits on usage. It's also worth bearing in mind that calling an external service on every request could impact performance.

To address these, it can also be worth considering:

1. An offline solution

Using a gem like **GeoIP** (see here) allows lookups to happen against a local datafile. There may be a trade-off in terms of accuracy, as these datafiles need to be kept up-to-date.

2. Use CloudFlare

Pages served through CloudFlare have the option of being geocoded transparently, with the country code being added to the header (`HTTP_CF_IPCOUNTRY`). More detail can be found here.

Section 17.4: Pluralization

You can let **I18n** handle pluralization for you, just use `count` argument.

You need to set up your locale file like this:

```
# config/locales/en.yml
en:
  online_users:
    one: "1 user is online"
    other: "%{count} users are online"
```

And then use the key you just created by passing the `count` argument to `I18n.t` helper:

```
I18n.t("online_users", count: 1)
#=> "1 user is online"

I18n.t("online_users", count: 4)
#=> "4 users are online"
```

Section 17.5: Set locale through requests

In most cases, you may want to set **I18n** locale. One might want to set the locale for the current session, the current user, or based on a URL parameter This is easily achievable by implementing a `before_action` in one of your controllers, or in `ApplicationController` to have it in all of your controllers.

```ruby
class ApplicationController < ActionController::Base
  before_action :set_locale

  protected

  def set_locale
    # Remove inappropriate/unnecessary ones
    I18n.locale = params[:locale] ||     # Request parameter
      session[:locale] ||                # Current session
      (current_user.preferred_locale if user_signed_in?) ||  # Model saved configuration
      extract_locale_from_accept_language_header ||       # Language header - browser config
      I18n.default_locale                # Set in your config files, english by super-default
  end
```

```ruby
# Extract language from request header
def extract_locale_from_accept_language_header
  if request.env['HTTP_ACCEPT_LANGUAGE']
    lg = request.env['HTTP_ACCEPT_LANGUAGE'].scan(/^[a-z]{2}/).first.to_sym
    lg.in?([:en, YOUR_AVAILABLE_LANGUAGES]) ? lg : nil
  end
end
```

URL-based

The `locale` param could come from an URL like this

```
http://yourapplication.com/products?locale=en
```

Or

```
http://yourapplication.com/en/products
```

To achieve the latter, you need to edit your `routes`, adding a `scope`:

```ruby
# config/routes.rb
scope "(:locale)", locale: /en|fr/ do
  resources :products
end
```

By doing this, visiting `http://yourapplication.com/en/products` will set your locale to `:en`. Instead, visiting `http://yourapplication.com/fr/products` will set it to `:fr`. Furthermore, you won't get a routing error when missing the `:locale` param, as visiting `http://yourapplication.com/products` will load the default **I18n** locale.

Session-based or persistence-based

This assumes the user can click on a button/language flag to change the language. The action can route to a controller that sets the session to the current language (and eventually persist the changes to a database if the user is connected)

```ruby
class SetLanguageController < ApplicationController
  skip_before_filter :authenticate_user!
  after_action :set_preferred_locale

  # Generic version to handle a large list of languages
  def change_locale
    I18n.locale = sanitize_language_param
    set_session_and_redirect
  end
```

> You have to define sanitize_language_param with your list of available languages, and eventually handle errors in case the language doesn't exist.

If you have very few languages, it may be worth defining them like this instead:

```ruby
def fr
  I18n.locale = :fr
  set_session_and_redirect
end

def en
```

```ruby
    I18n.locale = :en
    set_session_and_redirect
  end

private

  def set_session_and_redirect
    session[:locale] = I18n.locale
    redirect_to :back
  end

  def set_preferred_locale
    if user_signed_in?
      current_user.preferred_locale = I18n.locale.to_s
      current_user.save if current_user.changed?
    end
  end
end
```

Note: don't forget to add some routes to your `change_language` actions

Default Locale

Remember that you need to set your application default locale. You can do it by either setting it in `config/application.rb`:

```ruby
config.i18n.default_locale = :de
```

or by creating an initializer in the `config/initializers` folder:

```ruby
# config/initializers/locale.rb
I18n.default_locale = :it
```

Section 17.6: Use I18n with HTML Tags and Symbols

```yaml
# config/locales/en.yml
en:
  stackoverflow:
    header:
      title_html: "Use <strong>I18n</strong> with Tags &amp; Symbols"
```

Note the addition of extra `_html` after the name `title`.

And in Views,

```erb
# ERB
<h2><%= t(:title_html, scope: [:stackoverflow, :header]) %></h2>
```

Section 17.7: Use I18n in views

Assuming you have this YAML locale file:

```yaml
# config/locales/en.yml
en:
  header:
    title: "My header title"
```

and you want to display your title string, you can do this

```
# in ERB files
<%= t('header.title') %>

# in SLIM files
= t('header.title')
```

Chapter 18: Using GoogleMaps with Rails

Section 18.1: Add the google maps javascript tag to the layout header

In order to have google maps work properly with turbolinks, add the javascript tag directly to the layout header rather than including it in a view.

```
# app/views/layouts/my_layout.html.haml
!!!
%html{:lang => 'en'}
  %head
    - # ...
    = google_maps_api_script_tag
```

The `google_maps_api_script_tag` is best defined in a helper.

```
# app/helpers/google_maps_helper.rb
module GoogleMapsHelper
  def google_maps_api_script_tag
    javascript_include_tag google_maps_api_source
  end

  def google_maps_api_source
    "https://maps.googleapis.com/maps/api/js?key=#{google_maps_api_key}"
  end

  def google_maps_api_key
    Rails.application.secrets.google_maps_api_key
  end
end
```

You can register your application with google and get your api key in the google api console. Google has a short guide how to request an api key for the google maps javascript api.

The api key is stored in the `secrets.yml`file:

```
# config/secrets.yml
development:
  google_maps_api_key: '...'
  # ...
production:
  google_maps_api_key: '...'
  # ...
```

Don't forget to add `config/secrets.yml` to your `.gitignore` file and makre sure you don't commit the api key to the repository.

Section 18.2: Geocode the model

Suppose, your users and/or groups have profiles and you want to display address profile fields on a google map.

```
# app/models/profile_fields/address.rb
class ProfileFields::Address < ProfileFields::Base
  # Attributes:
  # label, e.g. "Work address"
```

```
  # value, e.g. "Willy-Brandt-Straße 1\n10557 Berlin"
end
```

A great way to geocode the addresses, i.e. provide `longitude` and `latitude` is the [geocoder gem](#).

Add geocoder to your `Gemfile` and run `bundle` to install it.

```
# Gemfile
gem 'geocoder', '~> 1.3'
```

Add database columns for `latitude` and `longitude` in order to save the location in the database. This is more efficient than querying the geocoding service every time you need the location. It's faster and you're not hitting the query limit so quickly.

```
➜ bin/rails generate migration add_latitude_and_longitude_to_profile_fields \
    latitude:float longitude:float
➜ bin/rails db:migrate   # Rails 5, or:
➜ rake db:migrate        # Rails 3, 4
```

Add the geocoding mechanism to your model. In this example, the address string is stored in the `value` attribute. Configure the geocoding to perform when the record has changed, and only whan a value is present:

```
# app/models/profile_fields/address.rb
class ProfileFields::Address < ProfileFields::Base
  geocoded_by :value
  after_validation :geocode, if: ->(address_field){
    address_field.value.present? and address_field.value_changed?
  }

end
```

By default, geocoder uses google as lookup service. It has lots of interesting features like distance calculations or proximity search. Fore more information, have a look at the [geocoder README](#).

Section 18.3: Show addresses on a google map in the profile view

On the profile view, show the profile fields of a user or group in a list as well as the address fields on a google map.

```
- # app/views/profiles/show.html.haml
%h1 Contact Information
.profile_fields
  = render @profile_fields
.google_map{data: address_fields: @address_fields.to_json }
```

The appropriate **@profile_fields** and **@address_fields** are set in the controller:

```
# app/controllers/profiles_controller.rb
class ProfilesController < ApplicationController
  def show
    # ...
    @profile_fields = @user_or_group.profile_fields
    @address_fields = @profile_fields.where(type: 'ProfileFields::Address')
  end
end
```

Initialize the map, place the markers, set the zoom and other map settings with javascript.
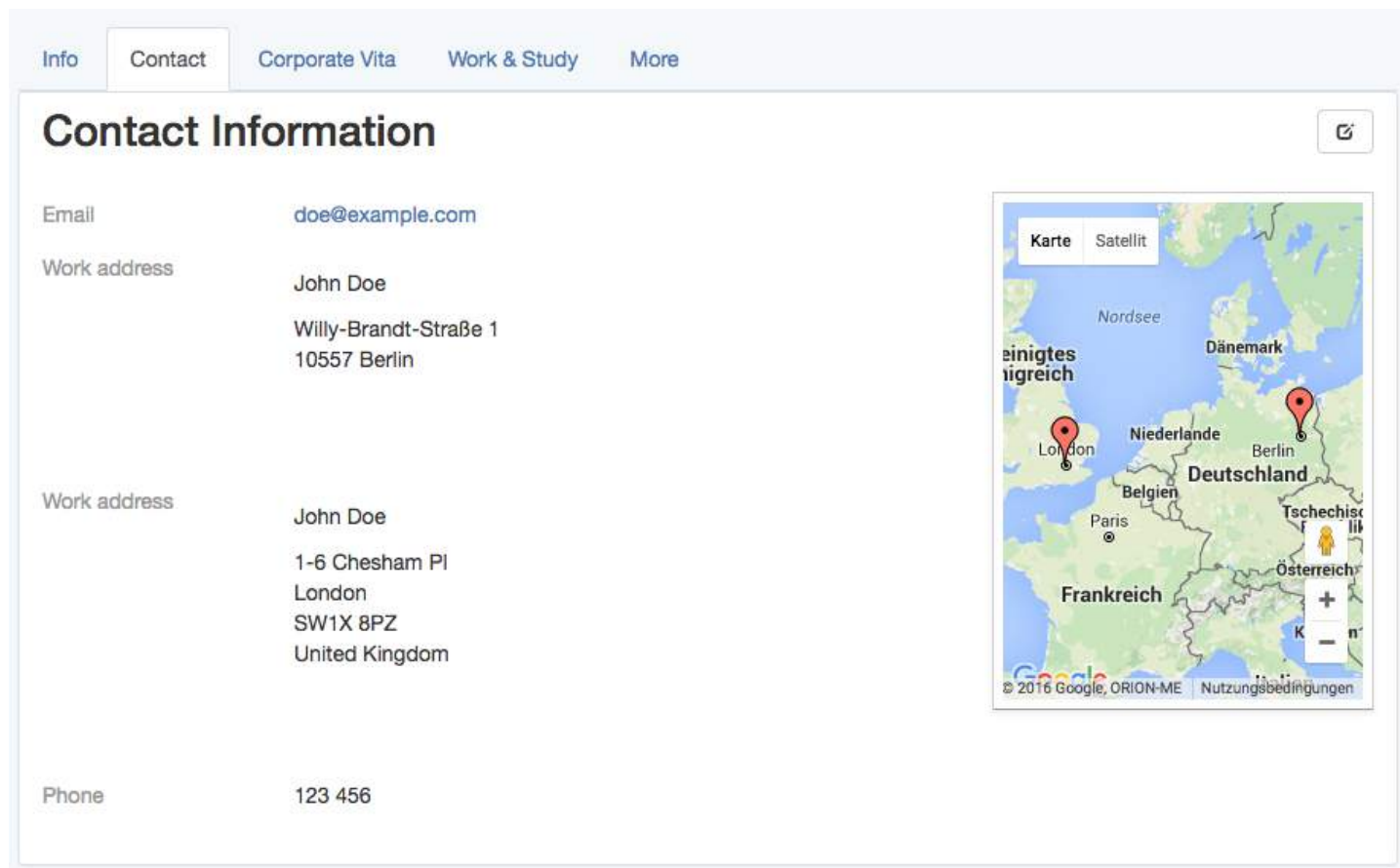
Section 18.4: Set the markers on the map with javascript

Suppose, there is a `.google_map` div, which will become the map, and which has the address fields to show as markers as `data` attribute.

For example:

```
<!-- http://localhost:3000/profiles/123 -->
<div class="google_map" data-address-fields="[
  {label: 'Work address', value: 'Willy-Brandt-Straße 1\n10557 Berlin',
   position: {lng: ..., lat: ...}},
  ...
]"></div>
```

To make use of the `$(document).ready` event with turbolinks without managing the turbolinks events by hand, use the jquery.turbolinks gem.

If you want to perform some other operations with the map, later, for example filtering or info windows, it's convenient to have the map managed by a coffee script class.

```
# app/assets/javascripts/google_maps.js.coffee
window.App = {} unless App?
class App.GoogleMap
  constructor: (map_div)->
    # TODO: initialize the map
    # TODO: set the markers
```

When using several coffee script files, which are namespaced by default, it's convenient to define a global `App` namespace, which is shared by all coffee script files.

Then, loop through (possibly several) `.google_map` divs and create one instance of the `App.GoogleMap` class for each of them.

```coffee
# app/assets/javascripts/google_maps.js.coffee
# ...
$(document).ready ->
  App.google_maps = []
  $('.google_map').each ->
    map_div = $(this)
    map = new App.GoogleMap map_div
    App.google_maps.push map
```

Section 18.5: Initialize the map using a coffee script class

Provided an `App.GoogleMap` coffee script class, the google map can be initialized like this:

```coffee
# app/assets/javascripts/google_maps.js.coffee
# ...
class App.GoogleMap
  map_div: {}
  map: {}

  constructor: (map_div)->
    @map_div = map_div
    @init_map()
    @reference_the_map_as_data_attribute

  # To access the GoogleMap object or the map object itself
  # later via the DOM, for example
  #
  #     $('.google_map').data('GoogleMap')
  #
  # store references as data attribute of the map_div.
  #
  reference_the_map_as_data_attribute: ->
    @map_div.data 'GoogleMap', this
    @map_div.data 'map', @map

  init_map: ->
    @map = new google.maps.Map(@dom_element, @map_configuration) if google?

  # `@map_div` is the jquery object. But google maps needs
  # the real DOM element.
  #
  dom_element: ->
    @map_div.get(0)

  map_configuration: -> {
    scrollWheel: true
  }
```

To learn more about the possible `map_configuration` options, have a look at google's MapOptions documentation and their guide to adding control elements.

For reference, the class `google.maps.Map`is extensively documented here.

Section 18.6: Initialize the map markers using a coffee script

class

Provided an `App.GoogleMap` [coffee script class](#) and the marker information being stored in the `data-address-fields` attribute of the `.google_map` div, the map markers can be initialized on the map like this:

```coffee
# app/assets/javascripts/google_maps.js.coffee
# ...
class App.GoogleMap
  # ...
  markers: []

  constructor: (map_div)->
    # ...
    @init_markers()

  address_fields: ->
    @map_div.data('address-fields')

  init_markers: ->
    self = this  # to reference the instance as `self` when `this` is redefined.
    self.markers = []
    for address_field in self.address_fields()
      marker = new google.maps.Marker {
        map: self.map,
        position: {
          lng: address_field.longitude,
          lat: address_field.latitude
        },
        # # or, if `position` is defined in `ProfileFields::Address#as_json`:
        # position: address_field.position,
        title: address_field.value
      }
      self.markers.push marker
```

To learn more about marker options, have a look at google's [MarkerOptions documentation](#) and their [guide to markers](#).

Section 18.7: Auto-zoom a map using a coffee script class

Provided an `App.GoogleMap` [coffee script class](#) with the `google.maps.Map` stored as `@map` and the `google.maps.Marker`s stored as `@markers`, the map can be auto-zoomed, i.e. adjusted that all markers are visible, like this: on the map like this:

```coffee
# app/assets/javascripts/google_maps.js.coffee
# ...
class App.GoogleMap
  # ...
  bounds: {}

  constructor: (map_div)->
    # ...
    @auto_zoom()

  auto_zoom: ->
    @init_bounds()
    # TODO: Maybe, adjust the zoom to have a maximum or
    # minimum zoom level, here.

  init_bounds: ->
```

```
@bounds = new google.maps.LatLngBounds()
for marker in @markers
  @bounds.extend marker.position
@map.fitBounds @bounds
```

To learn more about bounds, have a look at google's LatLngBounds documentation.

Section 18.8: Exposing the model properties as json

To display address profile fields as markers on a google map, the address field objects need to be passed as json objects to javascript.

Regular database attributes

When calling to_json on an ApplicationRecord object, the database attributes are automatically exposed.

Given a **ProfileFields::Address** model with label, value, longitude and latitude attributes, address_field.as_json results in a **Hash**, e.g. representation,

```
address_field.as_json  # =>
  {label: "Work address", value: "Willy-Brandt-Straße 1\n10557 Berlin",
    longitude: ..., latitude: ...}
```

which is converted to a json string by to_json:

```
address_field.to_json  # =>
  "{\"label\":\"Work address\",\"value\":\"Willy-Brandt-Straße 1\\n
    10557 Berlin\",\"longitude\":...,\"latitude\":...}"
```

This is useful because it allows to use label and value later in javascript, for example to show tool tips for the map markers.

Other attributes

Other virtual attributes can be exposed by overriding the as_json method.

For example, to expose a title attribute, include it in the merged as_json hash:

```
# app/models/profile_fields/address.rb
class ProfileFields::Address < ProfileFields::Base
  # ...

  # For example: "John Doe, Work address"
  def title
    "#{self.parent.name}, #{self.label}"
  end

  def as_json
    super.merge {
      title: self.title
    }
  end
end
```

The above example uses super to call the original as_json method, which returns the original attribute hash of the object, and merges it with the required position hash.

To understand the difference between `as_json` and `to_json`, have a look at [this blog post by jjulian](#).

Position

To render markers, the google maps api, by default, requires a `position` hash which has longitude and latitude stored as `lng` and `lat` respectively.

This position hash can be created in javascript, later, or here when defining the json representation of the address field:

To provide this `position` as json attribute of the address field, just override the `as_json` method on the model.

```ruby
# app/models/profile_fields/address.rb
class ProfileFields::Address < ProfileFields::Base
  # ...

  def as_json
    super.merge {
      # ...
      position: {
        lng: self.longitude,
        lat: self.latitude
      }
    }
  end
end
```

Chapter 19: File Uploads

Section 19.1: Single file upload using Carrierwave

Start using File Uploads in Rails is quite simple, first thing you have to do is to choice plugin for managing uploads. The most common onces are **Carrierwave** and **Paperclip**. Both are similar in functionality and rich in documentation on

Let's have an look on example with simple avatar upload image with Carrierwave

After `bundle install` Carrierwave, type in console

```
$ rails generate uploader ProfileUploader
```

This will create an config file located at */app/uploaders/profile_uploader.rb*

Here you can set up storage (i.e local or cloud), apply extensions for image manipulations (i.e. generting thumbs via MiniMagick) and set server-side extension white list

Next, create new migration with string tipe for user_pic and mount uploader for it in *user.rb* model.

```
mount_uploader :user_pic, ProfileUploader
```

Next, display an form to upload avatar (may be an edit view for the user)

```
<% form_for @user, html: { multipart: true } do |f| %>
    <%= f.file_field :user_pic, accept: 'image/png, image/jpg' %>
    <%= f.submit "update profile pic", class: "btn" %>
<% end %>
```

Make sure to include { multipart: true } in order form can process uploads. Accept is an optional to set client-side extension white-list.

To display an avatar, simply do

```
<%= image_tag @user.user_pic.url %>
```

Section 19.2: Nested model - multiple uploads

If you want to create multiple uploads, first thing you might want to do is create new model and set up relations

Let's say you want an multiple images for the Product model. Create an new model and make it `belongs_to` your parent model

```
rails g model ProductPhoto

#product.rb
has_many :product_photos, dependent: :destroy
accepts_nested_attributes_for :product_photos

#product_photo.rb
belongs_to :product
mount_uploader :image_url, ProductPhotoUploader # make sure to include uploader (Carrierwave example)
```

accepts_nested_attributes_for is must, because it allow us to create nested form, so we can upload new file, change product name and set price from an single form

Next, create form in a view (edit/create)

```erb
<%= form_for @product, html: { multipart: true } do |product|%>

    <%= product.text_field :price # just normal type of field %>

    <%= product.fields_for :product_photos do |photo| # nested fields %>
        <%= photo.file_field :image, :multiple => true, name: "product_photos[image_url][]" %>
    <% end %>
    <%= p.submit "Update", class: "btn" %>
<% end %>
```

Controller is nothing special, if you don't want to create an new one, just make an new one inside your product controller

```ruby
# create an action
def upload_file
  printer = Product.find_by_id(params[:id])
  @product_photo = printer.prodcut_photos.create(photo_params)
end

# strong params
private
  def photo_params
    params.require(:product_photos).permit(:image)
  end
```

Display all images in a view

```erb
<% @product.product_photos.each do |i| %>
    <%= image_tag i.image.url, class: 'img-rounded' %>
<% end %>
```

Chapter 20: Caching

Section 20.1: Russian Doll Caching

You may want to nest cached fragments inside other cached fragments. This is called `Russian doll caching`.

The advantage of `Russian doll caching` is that if a single product is updated, all the other inner fragments can be reused when regenerating the outer fragment.

As explained in the previous section, a cached file will expire if the value of `updated_at` changes for a record on which the cached file directly depends. However, this will not expire any cache the fragment is nested within.

For example, take the following view:

```erb
<% cache product do %>
  <%= render product.games %>
<% end %>
```

Which in turn renders this view:

```erb
<% cache game do %>
  <%= render game %>
<% end %>
```

If any attribute of game is changed, the `updated_at` value will be set to the current time, thereby expiring the cache.

However, because `updated_at` will not be changed for the product object, that cache will not be expired and your app will serve stale data. To fix this, we tie the models together with the touch method:

```ruby
class Product < ApplicationRecord
  has_many :games
end

class Game < ApplicationRecord
  belongs_to :product, touch: true
end
```

Section 20.2: SQL Caching

Query caching is a `Rails` feature that caches the result set returned by each query. If `Rails` encounters the same query again for that request, it will use the cached result set as opposed to running the query against the database again.

For example:

```ruby
class ProductsController < ApplicationController

  def index
    # Run a find query
    @products = Product.all

    ...

    # Run the same query again
    @products = Product.all
  end
```

```
end
```

The second time the same query is run against the database, it's not actually going to hit the database. The first time the result is returned from the query it is stored in the query cache (in memory) and the second time it's pulled from memory.

However, it's important to note that query caches are created at the start of an action and destroyed at the end of that action and thus persist only for the duration of the action. If you'd like to store query results in a more persistent fashion, you can with low level caching.

Section 20.3: Action caching

Like page caching, action caching caches the whole page. The difference is that the request hits the Rails stack so before filters are run before the cache is served. It's extracted from Rails to actionpack-action_caching gem.

A common example is caching of an action that requires authentication:

```ruby
class SecretIngredientsController < ApplicationController
  before_action :authenticate_user!, only: :index, :show
  caches_action :index

  def index
    @secret_ingredients = Recipe.find(params[:recipe_id]).secret_ingredients
  end
end
```

Options include :expires_in, a custom :cache_path (for actions with multiple routes that should be cached differently) and :if/:unless to control when the action should be cached.

```ruby
class RecipesController < ApplicationController
  before_action :authenticate_user!, except: :show
  caches_page :show
  caches_action :archive, expires_in: 1.day
  caches_action :index, unless: { request.format.json? }
end
```

When the layout has dynamic content, cache only the action content by passing `layout: false`.

Section 20.4: Fragment caching

`Rails.cache`, provided by ActiveSupport, can be used to cache any serializable Ruby object across requests.

To fetch a value from the cache for a given key, use `cache.read`:

```ruby
Rails.cache.read('city')
# => nil
```

Use `cache.write` to write a value to the cache:

```ruby
Rails.cache.write('city', 'Duckburgh')
Rails.cache.read('city')
# => 'Duckburgh'
```

Alternatively, use `cache.fetch` to read a value from the cache and optionally write a default if there is no value:

```ruby
Rails.cache.fetch('user') do
```

```ruby
  User.where(:is_awesome => true)
end
```

The return value of the passed block will be assigned to the cache under the given key, and then returned.

You can also specify a cache expiry:

```ruby
Rails.cache.fetch('user', :expires_in => 30.minutes) do
  User.where(:is_awesome => true)
end
```

Section 20.5: Page caching

You can use the ActionPack page_caching gem to cache individual pages. This stores the result of one dynamic request as a static HTML file, which is served in place of the dynamic request on subsequent requests. The README contains full setup instructions. Once set up, use the `caches_page` class method in a controller to cache the result of an action:

```ruby
class UsersController < ActionController::Base
  caches_page :index
end
```

Use `expire_page` to force expiration of the cache by deleting the stored HTML file:

```ruby
class UsersController < ActionController::Base
  caches_page :index

  def index
    @users = User.all
  end

  def create
    expire_page :action => :index
  end
end
```

The syntax of `expire_page` mimics that of `url_for` and friends.

Section 20.6: HTTP caching

Rails >= 3 comes with HTTP caching abilities out of the box. This uses the `Cache-Control` and `ETag` headers to control how long a client or intermediary (such as a CDN) can cache a page.

In a controller action, use `expires_in` to set the length of caching for that action:

```ruby
def show
  @user = User.find params[:id]
  expires_in 30.minutes, :public => true
end
```

Use `expires_now` to force immediate expiration of a cached resource on any visiting client or intermediary:

```ruby
def show
  @users = User.find params[:id]
  expires_now if params[:id] == 1
end
```

Chapter 21: ActionController

Action Controller is the C in MVC. After the router has determined which controller to use for a request, the controller is responsible for making sense of the request and producing the output.

The controller will receive the request, fetch or save data from a model and use a view to create output. A controller can be thought of as a middleman between models and views. It makes the model data available to the view so it can display to the user, and it saves or updates user data to the model.

Section 21.1: Basic REST Controller

```ruby
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]

  def index
    @posts = Post.all
  end

  def show

  end

  def new
    @post = Post.new
  end

  def edit

  end

  def create
    @post = Post.new(post_params)

    respond_to do |format|
      if @post.save
        format.html { redirect_to @post, notice: 'Post was successfully created.' }
        format.json { render :show, status: :created, location: @post }
      else
        format.html { render :new }
        format.json { render json: @post.errors, status: :unprocessable_entity }
      end
    end
  end

  def update
    respond_to do |format|
      if @post.update(post_params)
        format.html { redirect_to @post.company, notice: 'Post was successfully updated.' }
        format.json { render :show, status: :ok, location: @post }
      else
        format.html { render :edit }
        format.json { render json: @post.errors, status: :unprocessable_entity }
      end
    end
  end

  def destroy
    @post.destroy
    respond_to do |format|
```

```
      format.html { redirect_to posts_url, notice: 'Post was successfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private
    def set_post
      @post = Post.find(params[:id])
    end

    def post_params
      params.require(:post).permit(:title, :body, :author)
    end
end
```

Section 21.2: Filters

Filters are methods that are run "before", "after" or "around" a controller action. They are inherited, so if you set any in your `ApplicationController` they will be run for every request your application receives.

Before Filter

Before filters are executed before the controller action and can halt the request (and/or redirect). A common use is to verify if a user is logged in:

```
class ApplicationController < ActionController::Base
  before_action :authenticate_user!

  def authenticate_user!
    redirect_to some_path unless user_signed_in?
  end
end
```

Before filters are run on requests before the request gets to the controller's action. It can return a response itself and completely bypass the action.

Other common uses of before filters is validating a user's authentication before granting them access to the action designated to handle their request. I've also seen them used to load a resource from the database, check permissions on a resource, or manage redirects under other circumstances.

After Filter

After filters are similar to "before" ones, but as they get executed after the action run they have access the response object that's about to be sent. So in short after filters are run after the action completes. It can modify the response. Most of the time if something is done in an after filter, it can be done in the action itself, but if there is some logic to be run after running any of a set of actions, then an after filter is a good place to do it.

Generally, I've seen after and around filters used for logging.

Around Filter

Around filters may have logic before and after the action being run. It simply yields to the action in whatever place is necessary. Note that it doesn't need to yield to the action and may run without doing so like a before filter.

Around filters are responsible for running their associated actions by yielding, similar to how Rack middlewares work.

Around callbacks wrap the execution of actions. You can write an around callback in two different styles. In the first, the callback is a single chunk of code. That code is called before the action is executed. If the callback code invokes yield, the action is executed. When the action completes, the callback code continues executing. Thus, the code before the yield is like a before action callback and the code after the yield is the after action callback. If the callback code never invokes yield. the action is not run-this is the same as having a before action callback return false.

Here's an example of the around filter:

```ruby
around_filter :catch_exceptions

private
  def catch_exceptions
    begin
      yield
    rescue Exception => e
      logger.debug "Caught exception! #{e.message}"
    end
  end
```

This will catch exception of any action and put the message in your log. You can use around filters for exception handling, setup and teardown, and a myriad of other cases.

Only and Except

All filters can be applied to specific actions, using **:only** and **:except**:

```ruby
class ProductsController < ApplicationController
  before_action :set_product, only: [:show, :edit, :update]

  # ... controller actions

  # Define your filters as controller private methods
  private

  def set_product
    @product = Product.find(params[:id])
  end
end
```

Skipping Filter

All filters (inherited ones too) can also be skipped for some specific actions:

```ruby
class ApplicationController < ActionController::Base
  before_action :authenticate_user!

  def authenticate_user!
    redirect_to some_path unless user_signed_in?
  end
end

class HomeController < ApplicationController
  skip_before_action :authenticate_user!, only: [:index]

  def index
  end
end
```

As they're inherited, filters can also be defined in a `namespace` "parent" controller. Say for example that you have an `admin` namespace, and you of course want only admin users to be able to access it. You could do something like this:

```ruby
# config/routes.rb
namespace :admin do
  resources :products
end

# app/controllers/admin_controller.rb
class AdminController < ApplicationController
  before_action :authenticate_admin_user!

  private

  def authenticate_admin_user!
    redirect_to root_path unless current_user.admin?
  end
end

# app/controllers/admin/products_controller.rb
class Admin::ProductsController < AdminController
  # This controller will inherit :authenticate_admin_user! filter
end
```

Beware that in **Rails 4.x** you could use `before_filter` along with `before_action`, but `before_filter` is currently deprecated in **Rails 5.0.0** and will be removed in **5.1**.

Section 21.3: Generating a controller

Rails provides a lot of generators, for controllers too of course.

You can generate a new controller by running this command in your app folder

```
rails generate controller NAME [action action] [options]
```

Note: You can also use `rails g` alias to invoke `rails generate`

For example, to generate a controller for a `Product` model, with *#index* and *#show* actions you would run

```
rails generate controller products index show
```

This will create the controller in app**/**controllers**/**products_controller.`rb`, with both the actions you specified

```ruby
class ProductsController < ApplicationController
  def index
  end

  def show
  end
end
```

It will also create a `products` folder inside app**/**views**/**, containing the two templates for your controller's actions (i.e. index.`html`.`erb` and show.`html`.`erb`, *note that the extension may vary according to your template engine, so if you're using `slim`, for example, generator will create index.`html`.`slim` and show.`html`.`slim`*)

Furthermore, if you specified any actions they will also be added to your `routes` file

```ruby
# config/routes.rb
get 'products/show'
get 'products/index'
```

Rails creates a helper file for you, in app/helpers/products_helper.rb, and also the assets files in app/assets/javascripts/products.js and app/assets/stylesheets/products.css. As for views, the generator changes this behaviour according to what's specified in your Gemfile: i.e., if you're using Coffeescript and Sass in your application, the controller generator will instead generator products.coffee and products.sass.

At last, but not least, Rails also generates test files for your controller, your helper and your views.

If you don't want any of these to be created for you can tell Rails to skip them, just prepend any option with

--no-

or --skip, like this:

```
rails generate controller products index show --no-assets --no-helper
```

And the generator will skip both assets and helper

If you need to create a controller for a specific **namespace** add it in front of NAME:

```
rails generate controller admin/products
```

This will create your controller inside app/controllers/admin/products_controller.rb

Rails can also generate a complete RESTful controller for you:

```
rails generate scaffold_controller MODEL_NAME # available from Rails 4
rails generate scaffold_controller Product
```

Section 21.4: Rescuing ActiveRecord::RecordNotFound with redirect_to

You can rescue a RecordNotFound exception with a redirect instead of showing an error page:

```ruby
class ApplicationController < ActionController::Base

  # your other stuff

  rescue_from ActiveRecord::RecordNotFound do |exception|
    redirect_to root_path, 404, alert: I18n.t("errors.record_not_found")
  end
end
```

Section 21.5: Display error pages for exceptions

If you want to display to your users meaningful errors instead of simple "sorry, something went wrong", Rails has a nice utility for the purpose.

Open the file app/controllers/application_controller.rb and you should find something like this:

```ruby
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
```

```
  end
```

We can now add a `rescue_from` to recover from specific errors:

```ruby
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  rescue_from ActiveRecord::RecordNotFound, with: :record_not_found

  private

  def record_not_found
    render html: "Record <strong>not found</strong>", status: 404
  end
end
```

It's recommended not to rescue from **Exception** or **StandardError** otherwise Rails won't be able to display helpful pages in case of errors.

Section 21.6: Output JSON instead of HTML

```ruby
class UsersController < ApplicationController
  def index
    hashmap_or_array = [{ name: "foo", email: "foo@example.org" }]

    respond_to do |format|
      format.html { render html: "Hello World" }
      format.json { render json: hashmap_or_array }
    end
  end
end
```

In addition you will need the route:

```ruby
resources :users, only: [:index]
```

This will respond in two different ways to requests on `/users`:

- If you visit `/users` or `/users.html`, it will show an html page with the content `Hello World`
- If you visit `/users.json`, it will display a JSON object containing:

```json
[
  {
    "name": "foo",
    "email": "foo@example.org"
  }
]
```

You can **omit** `format.html { render inline: "Hello World" }` if you want to make sure that your route will answer only to JSON requests.

Section 21.7: Controllers (Basic)

```ruby
class UsersController < ApplicationController
  def index
    respond_to do |format|
      format.html { render html: "Hello World" }
    end
```

```
    end
  end
```

This is a basic controller, with the addition of the following route (in routes.rb):

```
resources :users, only: [:index]
```

Will display the `Hello World` message in a webpage when you access the URL `/users`

Section 21.8: Parameters

Controllers have access to HTTP parameters (you might know them as `?name=foo` in URLs, but Ruby on Rails handle different formats too!) and output different responses based on them. There isn't a way to distinguish between GET and POST parameters, but you shouldn't do that in any case.

```
class UsersController < ApplicationController
  def index
    respond_to do |format|
      format.html do
        if params[:name] == "john"
          render html: "Hello John"
        else
          render html: "Hello someone"
        end
      end
    end
  end
end
```

As usual our route:

```
resources :users, only: [:index]
```

Access the URL `/users?name=john` and the output will be `Hello John`, access `/users?name=whatever` and the output will be `Hello someone`

Section 21.9: Filtering parameters (Basic)

```
class UsersController < ApplicationController
  def index
    respond_to do |format|
      format.html do
        render html: "Hello #{ user_params[:name] } user_params[:sentence]"
      end
    end
  end

  private

  def user_params
    if params[:name] == "john"
      params.permit(:name, :sentence)
    else
      params.permit(:name)
    end
  end
end
```

You can allow (or reject) some params so that only what you want will *pass through* and you won't have bad surprises like user setting options not meant to be changed.

Visiting `/users?name=john&sentence=developer` will display `Hello john developer`, however visiting `/users?name=smith&sentence=spy` will display `Hello smith` only, because `:sentence` is only allowed when you access as `john`

Section 21.10: Redirecting

Assuming the route:

```
resources :users, only: [:index]
```

You can redirect to a different URL using:

```
class UsersController
  def index
    redirect_to "http://stackoverflow.com/"
  end
end
```

You can go back to the previous page the user visited using:

```
redirect_to :back
```

Note that in *Rails 5* the syntax for redirecting back is different:

```
redirect_back fallback_location: "http://stackoverflow.com/"
```

Which will try to redirect to the previous page and in case not possible (the browser is blocking the HTTP_REFERRER header), it will redirect to `:fallback_location`

Section 21.11: Using Views

Assuming the route:

```
resources :users, only: [:index]
```

And the controller:

```
class UsersController < ApplicationController
  def index
    respond_to do |format|
      format.html { render }
    end
  end
end
```

The view app/users/index.html.erb will be rendered. If the view is:

```
Hello <strong>World</strong>
```

The output will be a webpage with the text: "Hello **World**"

If you want to render a different view, you can use:

```
render "pages/home"
```

And the file app**/**views**/**pages**/**home.html.erb will be used instead.

You can pass variables to views using controller instance variables:

```ruby
class UsersController < ApplicationController
  def index
    @name = "john"

    respond_to do |format|
      format.html { render }
    end
  end
end
```

And in the file app**/**views**/**users**/**index.html.erb you can use @name:

```erb
Hello <strong><%= @name %></strong>
```

And the output will be: "Hello **john**"

An important note around the render syntax, you can omit the render syntax entirely, Rails assumes that if you omit it. So:

```ruby
class UsersController < ApplicationController
  def index
    respond_to do |format|
      format.html { render }
    end
  end
end
```

Can be written instead as:

```ruby
class UsersController < ApplicationController
  def index
    respond_to do |format|
      format.html
    end
  end
end
```

Rails is smart enough to figure out that it must render the file app**/**views**/**users**/**index.html.erb.

Chapter 22: Configuration

Section 22.1: Rails General Configuration

The following configuration options should be called on a `Rails::Railtie` object

- **config.after_initialize**: Takes a block which will be run after rails has initialized the application.
- **config.asset_host**: This sets the host for the assets. This is useful when using a *Content Delivery Network*. This is shorthand for `config.action_controller.asset_host`
- **config.autoload_once_paths**: This option accepts an array of paths where Rails autoloads constants. The default value is an empty array
- **config.autoload_paths**: This accepts an array of paths where Rails autoloads constants. It defaults to all directories under app
- **config.cache_classes**: Determines if classes and modules should be reloaded on each request. In development mode, this defaults to `false` and in the production and test modes it defaults to `true`
- **config.action_view.cache_template_loading**: This determines if templates should be reloaded on each request. It defaults to the `config.cache_classes` setting
- **config.beginning_of_week**: This sets the default beginning of week. It requires a valid week day symbol (`:monday`)
- **config.cache_store**: Choose which cache store to use. Options include `:file_store`, `:memory_store`, `mem_cache_store` or `null_store`.
- **config.colorize_logging**: This controls whether logging information is colorized
- **config.eager_load**: Eager-loads all registered
- **config.encoding**: Specifies the application encoding. The default value is UTF`-8`
- **config.log_level**: Sets the verbosity of the Rails Logger. It defaults to `:debug` in all environments.
- **config.middleware**: Use this to configure the application's middleware
- **config.time_zone**: This sets the application's default time zone.

Section 22.2: Configuring assets

The following configuration options can be used for configuring assets

- **config.assets.enabled**: Determines whether the asset pipeline is enabled. This defaults to true
- **config.assets.raise_runtime_errors**: This enables runtime error checking. It's useful for `development mode`
- **config.assets.compress**: Lets assets be compressed. In production mode, this defaults to true
- **config.assets.js_compressor**: Specifies which JS compressor to use. Options include `:closure`, `:uglifier` and `:yui`
- **config.assets.paths**: Specifies which paths to search for assets.
- **config.assets.precompile**: Lets you choose additional assets to be precompiled when `rake assets:precompile` is run
- **config.assets.digest**: This option allows the use of MD`-5` fingerprints in the asset names. It defaults to true in development mode
- **config.assets.compile**: Toggles live `Sprockets` compilation in production mode

Section 22.3: Configuring generators

Rails allows you to configure what generators are used when running `rails generate` commands. This method, `config.generators` takes a block

```
config.generators do |g|
  g.orm :active_record
```

```
    g.test_framework :test_unit
end
```

Here are some of the options

Option	Description	Default
assets	Creates assets when generating scaffold	true
force_plural	Allows pluralized model names	false
helper	Determines whether to generate helpers	true
integration_tool	Specify integration tool	`test_unit`
javascript_engine	Configures JS engine	`:js`
resource_route	Generates resource route	`true`
stylesheet_engine	Configures stylesheet engine	`:cs`
scaffold_stylesheet	Creates CSS upon scaffolding	`true`
test_framework	Specify Test Framework	`Minitest`
template_engine	Configures template engine	`:erb`

Section 22.4: Environments in Rails

Configuration files for rails can be found in `config/environments/`. By default rails has 3 environments, `development`, `production` and `test`. By editing each file you are editing the configuration for that environment only.

Rails also has a configuration file in `config/application.rb`. This is a common configuration file as any settings defined here are overwritten by the config specified in each environment.

You add or modify configuration options within the `Rails.application.configure do` block and configuration options start with `config`.

Section 22.5: Database Configuration

Database configuration of a rails project lies in a file `config/database.yml`. If you create a project using `rails new` command and don't specify a database engine to be used then rails uses `sqlite` as the default database. A typical `database.yml` file with default configuration will look similar to following.

```
# SQLite version 3.x
#   gem install sqlite3
#
#   Ensure the SQLite 3 gem is defined in your Gemfile
#   gem 'sqlite3'
#
default: &default
  adapter: sqlite3
  pool: 5
  timeout: 5000

development:
  <<: *default
  database: db/development.sqlite3


# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  <<: *default
  database: db/test.sqlite3


production:
```

```
  <<: *default
  database: db/production.sqlite3
```

If you want to change the default database while creating a new project you can specify `database: rails new hello_world --database=mysql`

Chapter 23: Safe Constantize

Section 23.1: Successful safe_constantize

User is an `ActiveRecord` or `Mongoid` class. Replace `User` with any `Rails` class in your project (even something like **Integer** or **Array**)

```
my_string = "User" # Capitalized string
  # => 'User'
my_constant = my_string.safe_constantize
  # => User
my_constant.all.count
  # => 18

my_string = "Array"
  # => 'Array'
my_constant = my_string.safe_constantize
  # => Array
my_constant.new(4)
  # => [nil, nil, nil, nil]
```

Section 23.2: Unsuccessful safe_constantize

This example will not work because the string passed in isn't recognized as a constant in the project. Even if you pass in `"array"`, it won't work as it isn't capitalized.

```
my_string = "not_a_constant"
  # => 'not_a_constant'
my_string.safe_constantize
  # => nil

my_string = "array" #Not capitalized!
  # => 'array'
my_string.safe_constantize
  # => nil
```

Chapter 24: Rails 5

Section 24.1: How to install Ruby on Rails 5 on RVM

RVM is a great tool to manage your ruby versions and set up your working environment.

Assuming you already have RVM installed, to get the latest version of ruby, which is needed for these examples, open a terminal and run:

```
$ rvm get stable
$ rvm install ruby --latest
```

Check your ruby version by running:

```
$ ruby -v
> ruby 2.3.0p0
```

To install Rails 5, first create a new gemset using the latest ruby version and then install rails:

```
$ rvm use ruby-2.3.0@my_app --create
$ gem install rails
```

To check your rails version, run:

```
$ rails -v
> Rails 5.0.0
```

Section 24.2: Creating a Ruby on Rails 5 API

To create a new Rails 5 API, open a terminal and run the following command:

```
rails new app_name --api
```

The following file structure will be created:

```
create
create  README.rdoc
create  Rakefile
create  config.ru
create  .gitignore
create  Gemfile
create  app
create  app/assets/javascripts/application.js
create  app/assets/stylesheets/application.css
create  app/controllers/application_controller.rb
create  app/helpers/application_helper.rb
create  app/views/layouts/application.html.erb
create  app/assets/images/.keep
create  app/mailers/.keep
create  app/models/.keep
create  app/controllers/concerns/.keep
create  app/models/concerns/.keep
create  bin
create  bin/bundle
create  bin/rails
create  bin/rake
```

```
create    bin/setup
create    config
create    config/routes.rb
create    config/application.rb
create    config/environment.rb
create    config/secrets.yml
create    config/environments
create    config/environments/development.rb
create    config/environments/production.rb
create    config/environments/test.rb
create    config/initializers
create    config/initializers/assets.rb
create    config/initializers/backtrace_silencers.rb
create    config/initializers/cookies_serializer.rb
create    config/initializers/filter_parameter_logging.rb
create    config/initializers/inflections.rb
create    config/initializers/mime_types.rb
create    config/initializers/session_store.rb
create    config/initializers/wrap_parameters.rb
create    config/locales
create    config/locales/en.yml
create    config/boot.rb
create    config/database.yml
create    db
create    db/seeds.rb
create    lib
create    lib/tasks
create    lib/tasks/.keep
create    lib/assets
create    lib/assets/.keep
create    log
create    log/.keep
create    public
create    public/404.html
create    public/422.html
create    public/500.html
create    public/favicon.ico
create    public/robots.txt
create    test/fixtures
create    test/fixtures/.keep
create    test/controllers
create    test/controllers/.keep
create    test/mailers
create    test/mailers/.keep
create    test/models
create    test/models/.keep
create    test/helpers
create    test/helpers/.keep
create    test/integration
create    test/integration/.keep
create    test/test_helper.rb
create    tmp/cache
create    tmp/cache/assets
create    vendor/assets/javascripts
create    vendor/assets/javascripts/.keep
create    vendor/assets/stylesheets
create    vendor/assets/stylesheets/.keep
```

This file structure will be created inside a new folder called app_name. It contains all the assets and code needed to start your project.

Enter the folder and install the dependencies:

```
cd app_name
bundle install
```

You should also start your database. Rails uses SQLite as a default database. To create it, run:

```
rake db:setup
```

Now run your appplication:

```
$ rails server
```

When you open your browser at http://localhost:3000, your shiny new (empty) API should be running!

Chapter 25: Authorization with CanCan

CanCan is a simple authorization strategy for Rails which is decoupled from user roles. All permissions are stored in a single location.

Section 25.1: Getting started with CanCan

CanCan is a a popular authorization library for Ruby on Rails which restricts user access to specific resources. The latest gem (CanCanCan) is a continuation of the dead project CanCan.

Permissions are defined in the `Ability` class and can be used from controllers, views, helpers, or any other place in the code.

To add authorization support to an app, add the CanCanCan gem to the `Gemfile`:

```
gem 'cancancan'
```

Then define the ability class:

```ruby
# app/models/ability.rb
class Ability
  include CanCan::Ability

  def initialize(user)
  end
end
```

Then check authorization using `load_and_authorize_resource` to load authorized models into the controller:

```ruby
class ArticlesController < ApplicationController
  load_and_authorize_resource

  def show
    # @article is already loaded and authorized
  end
end
```

`authorize!` to check authorization or raise an exception

```ruby
def show
  @article = Article.find(params[:id])
  authorize! :read, @article
end
```

`can?` to check if an object is authorized against a particular action anywhere in the controllers, views, or helpers

```erb
<% if can? :update, @article %>
  <%= link_to "Edit", edit_article_path(@article) %>
<% end %>
```

Note: This assumes the signed user is provided by the `current_user` method.

Section 25.2: Handling large number of abilities

Once the number of abilities definitions start to grow in number, it becomes more and more difficult to handle the

Ability file.

The first strategy to handle these issue is to move abilities into meaningful methods, as per this example:

```ruby
class Ability
  include CanCan::Ability

  def initialize(user)
    anyone_abilities

    if user
      if user.admin?
        admin_abilities
      else
        authenticated_abilities
      end
    else
      guest_abilities
    end
  end

  private

  def anyone_abilities
    # define abilities for everyone, both logged users and visitors
  end

  def guest_abilities
    # define abilities for visitors only
  end

  def authenticated_abilities
    # define abilities for logged users only
  end

  def admin_abilities
    # define abilities for admins only
  end
end
```

Once this class grow large enough, you can try breaking it into different classes to handle the different responsibilities like this:

```ruby
# app/models/ability.rb
class Ability
  include CanCan::Ability

  def initialize(user)
    self.merge Abilities::Everyone.new(user)

    if user
      if user.admin?
        self.merge Abilities::Admin.new(user)
      else
        self.merge Abilities::Authenticated.new(user)
      end
    else
      self.merge Abilities::Guest.new(user)
    end
  end
```

```
end
```

and then define those classes as:

```ruby
# app/models/abilities/guest.rb
module Abilities
  class Guest
    include CanCan::Ability

    def initialize(user)
      # Abilities for anonymous visitors only
    end
  end
end
```

and so on with **Abilities::Authenticated**, **Abilities::Admin** or any other else.

Section 25.3: Defining abilities

Abilities are defined in the `Ability` class using `can` and `cannot` methods. Consider the following commented example for basic reference:

```ruby
class Ability
  include CanCan::Ability

  def initialize(user)
    # for any visitor or user
    can :read, Article

    if user
      if user.admin?
        # admins can do any action on any model or action
        can :manage, :all
      else
        # regular users can read all content
        can :read, :all
        # and edit, update and destroy their own user only
        can [:edit, :destroy], User, id: user_id
        # but cannot read hidden articles
        cannot :read, Article, hidden: true
      end
    else
      # only unlogged visitors can visit a sign_up page:
      can :read, :sign_up
    end
  end
end
```

Section 25.4: Quickly test an ability

If you'd like to quickly test if an ability class is giving the correct permissions, you can initialize an ability in the console or on another context with the rails environment loaded, just pass an user instance to test against:

```ruby
test_ability = Ability.new(User.first)
test_ability.can?(:show, Post) #=> true
other_ability = Ability.new(RestrictedUser.first)
other_ability.cannot?(:show, Post) #=> true
```

More information: https://github.com/ryanb/cancan/wiki/Testing-Abilities

Chapter 26: Mongoid

Section 26.1: Fields

As per the [Mongoid Documentation](), there are 16 valid field types:

- Array
- BigDecimal
- Boolean
- Date
- DateTime
- Float
- Hash
- Integer
- BSON::ObjectId
- BSON::Binary
- Range
- Regexp
- String
- Symbol
- Time
- TimeWithZone

To add a field (let's call it `name` and have it be a `String`), add this to your model file:

```
field :name, type: String
```

To set a default value, just pass in the `default` option:

```
field :name, type: String, default: ""
```

Section 26.2: Installation

First, add `Mongoid` to your `Gemfile`:

```
gem "mongoid", "~> 4.0.0"
```

and then run `bundle install`. Or just run:

```
$ gem install mongoid
```

After installation, run the generator to create the config file:

```
$ rails g mongoid:config
```

which will create the file `(myapp)/config/mongoid.yml`.

Section 26.3: Creating a Model

Create a model (lets call it `User`) by running:

```
$ rails g model User
```

which will generate the file app**/**models**/**user.rb:

```
class User
  include Mongoid::Document

end
```

This is all you need to have a model (albeit nothing but an `id` field). Unlike `ActiveRecord`, there is no migration files. All the database information for the model is contained in the model file.

Timestamps are not automatically included in your model when you generate it. To add `created_at` and `updated_at` to your model, add

```
include Mongoid::Timestamps
```

to your model underneath `include Mongoid::Document` like so:

```
class User
  include Mongoid::Document
  include Mongoid::Timestamps

end
```

Section 26.4: Classic Associations

Mongoid allows the classic `ActiveRecord` associations:

- One-to-one: `has_one` / `belongs_to`
- One-to-many: `has_many` / `belongs_to`
- Many-to-many: `has_and_belongs_to_many`

To add an association (lets say the User `has_many` posts), you can add this to your `User` model file:

```
has_many :posts
```

and this to your `Post` model file:

```
belongs_to :user
```

This will add a `user_id` field in your `Post` model, add a `user` method to your `Post` class, and add a `posts` method to your `User` class.

Section 26.5: Embedded Associations

Mongoid allows Embedded Associations:

- One-to-one: `embeds_one` / `embedded_in`
- One-to-many: `embeds_many` / `embedded_in`

To add an association (lets say the User `embeds_many` addresses), add this to your `User` file:

```
embeds_many :addresses
```

and this to your `Address` model file:

```
embedded_in :user
```

This will embed `Address` in your `User` model, adding a `addresses` method to your `User` class.

Section 26.6: Database Calls

Mongoid tries to have similar syntax to `ActiveRecord` when it can. It supports these calls (and many more)

```ruby
User.first #Gets first user from the database

User.count #Gets the count of all users from the database

User.find(params[:id]) #Returns the user with the id found in params[:id]

User.where(name: "Bob") #Returns a Mongoid::Criteria object that can be chained
                        #with other queries (like another 'where' or an 'any_in')
                        #Does NOT return any objects from database

User.where(name: "Bob").entries #Returns all objects with name "Bob" from database

User.where(:name.in => ['Bob', 'Alice']).entries #Returns all objects with name "Bob" or "Alice"
from database

User.any_in(name: ["Bob", "Joe"]).first #Returns the first object with name "Bob" or "Joe"
User.where(:name => 'Bob').exists? # will return true if there is one or more users with name bob
```

Chapter 27: Gems

Section 27.1: Gemfiles

To start, gemfiles require at least one source, in the form of the URL for a RubyGems server.

Generate a Gemfile with the default rubygems.org source by running `bundle init`. Use https so your connection to the server will be verified with SSL.

```
source 'https://rubygems.org'
```

Next, declare the gems that you need, including version numbers.

```
gem 'rails', '4.2.6'
gem 'rack',  '>=1.1'
gem 'puma',  '~>3.0'
```

Most of the version specifiers, like >= 1.0, are self-explanatory. The specifier ~> has a special meaning. ~> 2.0.3 is identical to >= 2.0.3 and < 2.1. ~> 2.1 is identical to >= 2.1 and < 3.0. ~> 2.2.beta will match prerelease versions like 2.2.beta.12.

Git repositories are also valid gem sources, as long as the repo contains one or more valid gems. Specify what to check out with `:tag`, `:branch`, or `:ref`. The default is the `master` branch.

```
gem 'nokogiri', :git => 'https://github.com/sparklemotion/nokogiri', :branch => 'master'
```

If you would like to use an unpacked gem directly from the filesystem, simply set the :path option to the path containing the gem's files.

```
gem 'extracted_library', :path => './vendor/extracted_library'
```

Dependencies can be placed into groups. Groups can be ignored at install-time (using `--without`) or required all at once (using `Bundler.require`).

```
gem 'rails_12factor', group: :production

group :development, :test do
  gem 'byebug'
  gem 'web-console', '~> 2.0'
  gem 'spring'
  gem 'dotenv-rails'
end
```

You can specify the required version of Ruby in the Gemfile with `ruby`. If the Gemfile is loaded on a different Ruby version, Bundler will raise an exception with an explanation.

```
ruby '2.3.1'
```

Section 27.2: What is a gem?

A gem is the equivalent to a plugin or an extension for the programming language ruby.

To be exact even rails is nothing more than a gem. A lot of gems are built on rails or other gems (they are dependent of said gem) or are standalone.

In your Rails project
Gemfile

For your Rails project you have a file called `Gemfile`. In here you can add gems you want to include and use in your project. Once added you need to install the gem by using `bundler` (See Bundler section).

Gemfile.lock

Once you have done this, your `Gemfile.lock` will be updated with your newly added gems and their dependencies. This file locks your used gems so they use that specific version declared in that file.

```
GEM
remote: https://rubygems.org/
specs:
devise (4.0.3)
bcrypt (~> 3.0)
orm_adapter (~> 0.1)
railties (>= 4.1.0, < 5.1)
responders
warden (~> 1.2.3)
```

This example is for the gem `devise`. In the `Gemfile.lock` the version `4.0.3` is declared, to tell when installing your project on an other machine or on your production server which specified version to use.

Development

Either a single person, a group or a whole community works on and maintains a gem. Work done is usually released after certain `issues` have been fixed or `features` have been added.

Usually the releases follow the [Semantic Versioning 2.0.0](#) principle.

Section 27.3: Bundler

The easiest way to handle and manage gems is by using `bundler`. [Bundler](#) is a package manager comparable to bower.

To use bundler you first need to install it.

```
gem install bundler
```

After you have bundler up and running all you need to do is add gems to your `Gemfile` and run

```
bundle
```

in your terminal. This installs your newly added gems to your project. Should an issue arise, you would get a prompt in your terminal.

If you are interested in more details, I suggest you have a look at the [docs](#).

Section 27.4: Gemsets

If you are using RVM`(Ruby Version Manager)` then using a `gemset` for each project is a good idea. A `gemset` is just a container you can use to keep gems separate from each other. Creating a `gemset` per project allows you to change gems (and gem versions) for one project without breaking all your other projects. Each project need only worry about its own gems.

RVM provides (>= 0.1.8) a **@global** gemset per ruby interpreter. Gems you install to the **@global** gemset for a given ruby are available to all other gemsets you create in association with that ruby. This is a good way to allow all of your projects to share the same installed gem for a specific ruby interpreter installation.

Creating gemsets

Suppose you already have `ruby-2.3.1` installed and you have selected it using this command:

```
rvm use ruby-2.3.1
```

Now to create gemset for this ruby version:

```
rvm gemset create new_gemset
```

where the `new_gemset` is the name of gemset. To see the list of available gemsets for a ruby version:

```
rvm gemset list
```

to list the gems of all ruby versions:

```
rvm gemset list_all
```

to use a gemset from the list (suppose `new_gemset` is the gemset I want to use):

```
rvm gemset use new_gemset
```

you can also specify the ruby version with the gemset if you want to shift to some other ruby version:

```
rvm use ruby-2.1.1@new_gemset
```

to specify a default gemset for a particular ruby version:

```
rvm use 2.1.1@new_gemset --default
```

to remove all the installed gems from a gemset you can empty it by:

```
rvm gemset empty new_gemset
```

to copy a gemset from one ruby to another you can do it by:

```
rvm gemset copy 2.1.1@rails4 2.1.2@rails4
```

to delete a gemset:

```
rvm gemset delete new_gemset
```

to see the current gemset name:

```
rvm gemset name
```

to install a gem in the global gemset:

```
rvm @global do gem install ...
```

Initializing Gemsets during Ruby Installs

When you install a new ruby, RVM not only creates two gemsets (the default, empty gemset and the global gemset), it also uses a set of user-editable files to determine which gems to install.

Working in ~**/**.`rvm`**/**gemsets, rvm searchs for `global.`gems and `default.`gems using a tree-hierachy based on the ruby string being installed. Using the example of `ree-1.8.7-p2010.02`, rvm will check (and import from) the following files:

```
~/.rvm/gemsets/ree/1.8.7/p2010.02/global.gems
~/.rvm/gemsets/ree/1.8.7/p2010.02/default.gems
~/.rvm/gemsets/ree/1.8.7/global.gems
~/.rvm/gemsets/ree/1.8.7/default.gems
~/.rvm/gemsets/ree/global.gems
~/.rvm/gemsets/ree/default.gems
~/.rvm/gemsets/global.gems
~/.rvm/gemsets/default.gems
```

For example, if you edited ~**/**.`rvm`**/**gemsets**/**global.`gems` by adding these two lines:

```
bundler
awesome_print
```

every time you install a new ruby, these two gems are installed into your global gemset. `default.`gems and `global.`gems files are usually overwritten during update of rvm.

Chapter 28: Change default timezone

Section 28.1: Change Rails timezone AND have Active Record store times in this timezone

```
# application.rb
config.time_zone = 'Eastern Time (US & Canada)'
config.active_record.default_timezone = :local
```

Section 28.2: Change Rails timezone, but continue to have Active Record save in the database in UTC

```
# application.rb
config.time_zone = 'Eastern Time (US & Canada)'
```

Chapter 29: Asset Pipeline

The asset pipeline provides a framework to concatenate and minify or compress JavaScript and CSS assets. It also adds the ability to write these assets in other languages and pre-processors such as CoffeeScript, Sass and ERB. It allows assets in your application to be automatically combined with assets from other gems. For example, jquery-rails includes a copy of jquery.js and enables AJAX features in Rails.

Section 29.1: Manifest Files and Directives

In the assets initalizer (config**/**initializers**/**assets.rb) are a few files explicitly defined to be precompiled.

```
# Precompile additional assets.
# application.coffee, application.scss, and all non-JS/CSS in app/assets folder are already added.
# Rails.application.config.assets.precompile += %w( search.js )
```

In this example the application.coffee and application.scss are so called 'Manifest Files'. This files should be used to include other JavaScript or CSS assets. The following command are available:

- **require <path>**: The **require** directive functions similar to Ruby's own **require**. It provides a way to declare a dependency on a file in your path and ensures it's only loaded once before the source file.
- require_directory **<path>**: requires all the files inside a single directory. It's similar to path**/*** since it does not follow nested directories.
- require_tree **<path>**: requires all the nested files in a directory. Its glob equivalent is path**/**/***.
- require_self: causes the body of the current file to be inserted before any subsequent **require** directives. Useful in CSS files, where it's common for the index file to contain global styles that need to be defined before other dependencies are loaded.
- stub **<path>**: remove a file from being included
- depend_on **<path>**: Allows you to state a dependency on a file without including it. This is used for caching purposes. Any changes made to the dependency file will invalidate the cache of the source file.

An application.scss file could look like:

```
/*
 *= require bootstrap
 *= require_directory .
 *= require_self
 */
```

Another example is the application.coffee file. Here with including jquery and Turbolinks:

```
#= require jquery2
#= require jquery_ujs
#= require turbolinks
#= require_tree .
```

If you don't use CoffeeScript, but plain JavaScript, the syntax would be:

```
//= require jquery2
//= require jquery_ujs
//= require turbolinks
//= require_tree .
```

Section 29.2: Rake tasks

By default `sprockets-rails` is shipped with the following rake tasks:

- `assets:clean[keep]`: Remove old compiled assets
- `assets:clobber`: Remove compiled assets
- `assets:environment`: Load asset compile environment
- `assets:precompile`: Compile all the assets named in `config.assets.precompile`

Section 29.3: Basic Usage

There are two basic ways that the asset pipeline is used:

1. When running a server in development mode, it automatically pre-processes and prepares your assets on-the-fly.

2. In production mode, you'll probably use it to pre-process, versionize, and compress and compile your assets. You can do so by running the following command:

```
bundle exec rake assets:precompile
```

Chapter 30: Upgrading Rails

Section 30.1: Upgrading from Rails 4.2 to Rails 5.0

Note: Before upgrading your Rails app, always make sure to save your code on a version control system, such as Git.

To upgrade from Rails 4.2 to Rails 5.0, you must be using Ruby 2.2.2 or newer. After upgrading your Ruby version if required, go to your Gemfile and change the line:

```
gem 'rails', '4.2.X'
```

to:

```
gem 'rails', '~> 5.0.0'
```

and on the command line run:

```
$ bundle update
```

Now run the update task using the command:

```
$ rake rails:update
```

This will help you to update configuration files. You will be prompted to overwrite files and you have several options to input:

- Y – yes, overwrite
- n – no, do not overwrite
- a – all, overwrite this and all others
- q – quit, abort
- d – diff, show the differences between the old and the new
- h – help

Typically, you should check the differences between the old and new files to make sure you aren't getting any unwanted changes.

Rails 5.0 `ActiveRecord` models inherit from `ApplicationRecord`, rather than `ActiveRecord::Base`. `ApplicationRecord` is the superclass for all models, similar to how `ApplicationController` is the superclass for controllers. To account for this new way in which models are handled, you must create a file in your `app/models/` folder called `application_record.rb` and then edit that file's contents to be:

```ruby
class ApplicationRecord < ActiveRecord::Base
  self.abstract_class = true
end
```

Rails 5.0 also handles callbacks slightly different. Callbacks that return `false` won't halt the callback chain, which means subsequent callbacks will still run, unlike Rails 4.2. When you upgrade, the Rails 4.2 behavior will remain, though you can switch to the Rails 5.0 behavior by adding:

```ruby
ActiveSupport.halt_callback_chains_on_return_false = false
```

to the `config/application.rb` file. You can explicitly halt the callback chain by calling `throw(:abort)`.

In Rails 5.0, `ActiveJob` will inherit from `ApplicationJob`, rather than **ActiveJob::Base** like in Rails 4.2. To upgrade to Rails 5.0, create a file called `application_job.rb` in the app**/**jobs**/** folder. Edit that file's contents to be:

```
class ApplicationJob < ActiveJob::Base
end
```

Then, you must change all of your jobs to inherit from `ApplicationJob` rather than **ActiveJob::Base**.

One of the other biggest changes of Rails 5.0 doesn't require any code changes, but will change the way you use the command line with your Rails apps. You will be able to use `bin/rails`, or just `rails`, to run tasks and tests. For example, instead of using $ `rake db:migrate`, you can now do $ `rails db:migrate`. If you run $ `bin/rails`, you can view all the available commands. Note that many of the tasks that can now be run with `bin/rails` still work using `rake`.

Chapter 31: ActiveRecord Locking

Section 31.1: Optimistic Locking

```ruby
user_one = User.find(1)
user_two = User.find(1)

user_one.name = "John"
user_one.save
# Run at the same instance
user_two.name = "Doe"
user_two.save # Raises a ActiveRecord::StaleObjectError
```

Section 31.2: Pessimistic Locking

```ruby
appointment = Appointment.find(5)
appointment.lock!
#no other users can read this appointment,
#they have to wait until the lock is released
appointment.save!
#lock is released, other users can read this account
```

Chapter 32: Debugging

Section 32.1: Debugging Rails Application

To be able to debug an application is very important to understand the flow of an application's logic and data. It helps solving logical bugs and adds value to the programming experience and code quality. Two popular gems for debugging are debugger (for ruby 1.9.2 and 1.9.3) and byebug (for ruby >= 2.x).

For debugging `.rb` files, follow these steps:

1. Add `debugger` or byebug to the `development` group of `Gemfile`
2. Run `bundle install`
3. Add `debugger` or byebug as the breakpoint
4. Run the code or make request
5. See the rails server log stopped at the specified breakpoint
6. At this point you can use your server terminal just like `rails console` and check the values of variable and params
7. For moving to next instruction, type **next** and press `enter`
8. For stepping out type `c` and press `enter`

If you want to debug `.html.erb` files, break point will be added as `<% debugger %>`

Section 32.2: Debugging Ruby on Rails Quickly + Beginner advice

Debugging by raising exceptions is *far easier* than squinting through **print** log statements, and for most bugs, its generally *much faster* than opening up an irb debugger like `pry` or `byebug`. Those tools should not be your first step.

Debugging Ruby/Rails Quickly:

1. Fast Method: Raise an Exception then and `.inspect` its result

The *fastest* way to debug Ruby (especially Rails) code is to **raise** an exception along the execution path of your code while calling `.inspect` on the method or object (e.g. `foo`):

```
raise foo.inspect
```

In the above code, ***raise*** triggers an ***Exception*** that *halts execution of your code*, and returns an error message that conveniently contains `.inspect` information about the object/method (i.e. `foo`) on the line that you're trying to debug.

This technique is useful for *quickly* examining an object or method (*e.g. is it **nil**?*) and for immediately confirming whether a line of code is even getting executed at all within a given context.

2. Fallback: Use a ruby *IRB* debugger like byebug or pry

Only after you have information about the state of your codes execution flow should you consider moving to a ruby gem irb debugger like `pry` or `byebug` where you can delve more deeply into the state of objects within your execution path.

To use the byebug gem for debugging in Rails:

1. Add `gem 'byebug'` inside the *development* group in your *Gemfile*
2. Run `bundle install`
3. Then to use, insert the phrase `byebug` inside the execution path of the code you want examined.

This `byebug` variable when executed will open up an ruby IRB session of your code, giving you direct access to the state of objects as they are at that point in the code's execution.

IRB debuggers like Byebug are useful for deeply analyzing the state of your code as it executes. However, they are more time consuming procedure compared to raising errors, so in most situations they should not be your first step.

General Beginner Advice

When you are trying to debug a problem, good advice is to always: **Read The !@#$ing Error Message (RTFM)**

That means reading error messages *carefully* and *completely* before acting so that you *understand what it's trying to tell you.* When you debug, ask the following mental questions, *in this order*, when reading an error message:

1. What **class** does the error reference? (i.e. *do I have the correct object class or is my object `nil`?*)
2. What **method** does the error reference? (i.e. *is their a type in the method; can I call this method on this type/class of object?*)
3. Finally, using what I can infer from my last two questions, what **lines of code** should I investigate? (remember: the last line of code in the stack trace is not necessarily where the problem lies.)

In the stack trace pay particular attention to lines of code that come from your project (e.g. lines starting with `app/...` if you are using Rails). 99% of the time the problem is with your own code.

To illustrate why interpreting *in this order* is important...

E.g. a Ruby error message that confuses many beginners:

You execute code that at some point executes as such:

```
@foo = Foo.new

...

@foo.bar
```

and you get an error that states:

`undefined method "bar" for Nil:nilClass`

Beginners see this error and think the problem is that the method `bar` is *undefined*. **It's not.** In this error the real part that matters is:

`for Nil:nilClass`

`for Nil:nilClass` **means that** `@foo` **is Nil!** `@foo` is not a Foo instance variable! You have an object that is `Nil`. When you see this error, it's simply ruby trying to tell you that the method `bar` doesn't exist for objects of the class `Nil`. (well duh! since we are trying to use a method for an object of the class Foo not `Nil`).

Unfortunately, due to how this error is written (`undefined method "bar" for Nil:nilClass`) its easy to get tricked into thinking this error has to do with `bar` being `undefined`. When not read carefully this error causes beginners to mistakenly go digging into the details of the `bar` method on Foo, entirely missing the part of the error that hints that

the object is of the wrong class (in this case: nil). It's a mistake that's easily avoided by reading error messages in their entirety.

Summary:

Always carefully **read the *entire* error message** before beginning any debugging. That means: Always check the ***class*** type of an object in an error message *first*, then its **methods**, *before* you begin sleuthing into any stacktrace or line of code where you think the error may be occurring. Those 5 seconds can save you 5 hours of frustration.

tl;dr: Don't squint at print logs: raise exceptions instead. Avoid rabbit holes by reading errors carefully before debugging.

Section 32.3: Debugging ruby-on-rails application with pry

pry is a powerful tool that can be used to debug any ruby application. Setting up a ruby-on-rails application with this gem is very easy and straightforward.

Setup

To start debugging your application with pry

- Add `gem 'pry'` to the application's `Gemfile` and bundle it

```
group :development, :test do
  gem 'pry'
end
```

- Navigate to the application's root directory on terminal console and run `bundle install`. You're all set to start using it anywhere on your application.

Use

Using pry in your application is just including **binding**.pry on the breakpoints you want to inspect while debugging. You can add **binding**.pry breakpoints anywhere in your application that is interpreted by ruby interpreter (any app/controllers, app/models, app/views files)

i) Debugging a Controller

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  def show
    use_id = params[:id]
    // breakpoint to inspect if the action is receiving param as expected
    binding.pry
    @user = User.find(user_id)
    respond_to do |format|
      format.html
    end
  end
end
```

In this example, the rails server pauses with a pry console at the break-point when you try to visit a page routing to `show` action on `UsersController`. You can inspect `params` object and make ActiveRecord query on `User` model from that breakpoint

ii) Debugging a View

app/views/users/show.html.haml

```haml
%table
  %tbody
    %tr
      %td ID
      %td= @user.id
    %tr
      %td email
      %td= @user.email
    %tr
      %td logged in ?
      %td
        - binding.pry
        - if @user.logged_in?
          %p= "Logged in"
        - else
          %p= "Logged out"
```

In this example, the break-point pauses with pry console when the `users/show` page is pre-compiled in the rails server before sending it back to the client's browser. This break-point allows to debug correctness of `@user.logged_in?` when it is misbehaving.

ii) Debugging a Model

```ruby
app/models/user.rb

class User < ActiveRecord::Base
  def full_name
    binding.pry
    "#{self.first_name} #{self.last_name}"
  end
end
```

In this example, the break-point can be used to debug `User` model's instance method `full_name` when this method is called from anywhere in the application.

In conclusion, pry is a powerful debugging tool for rails application with easy setup and straightforward debugging guideline. Give this a try.

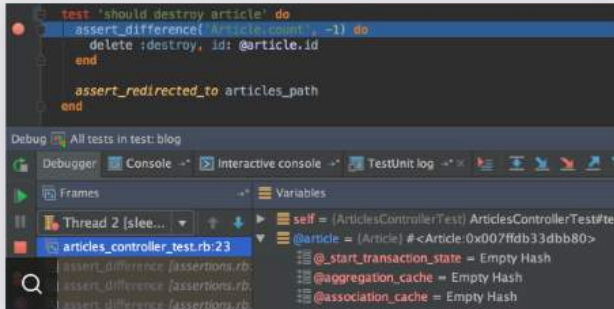# Section 32.4: Debugging in your IDE

Every good IDE provides a GUI for interactively debugging Ruby (and thus Rails) applications where you can add breakpoints, watches, auto pausing on exception and allows you to follow the code execution even step by step, line by line.

For example, take a look at one of the best Ruby IDE's, RubyMine's debugging features on the picture

Powerful Debugger

RubyMine brings a clever debugger with a graphical UI for Ruby, JS, and CoffeeScript. Set breakpoints and run your code step by step with all the information at your fingertips.

Convenient user interface

- Look under the hood of any code and see what's going on—thanks to Frames, Variables and Watches views.
- The UI is fully customizable letting you arrange, resize and float the views, select toolbar commands, etc. You can also choose whether to ignore non-project code while stepping or not.
- The debugger UI is also tightly integrated with the code editor: you can navigate between the debugger and the code, view breakpoint information, etc.
- You also get the complete coding assistance and highlighting in all debugger views.

Smart, flexible breakpoints

- Place a breakpoint on a line of code and define the hit conditions—a set of Boolean expressions that are evaluated to determine whether to stop the code execution or not.
- If you have multiple breakpoints in your code, you can set dependencies between them to define the order in which they can be hit.
- Setting a breakpoint is just a matter of a single mouse click on the gutter or invoking a shortcut.
- Breakpoints are also available in Rails views, so you can use them for debugging Rails code as well.

Debugging JavaScript and Node.js

- RubyMine provides an advanced built-in debugger for your JavaScript code, which works with Google Chrome.
- You can easily debug ECMAScript 6, TypeScript or CoffeeScript code, relying on RubyMine debugger's support for source maps.
- A full-featured debugger for Node.js works right out of the box as well. Use it to debug apps running locally or on a remote machine.

Built-in expression evaluator

Evaluate any expression while your debugging session is paused. Type an expression or a code fragment, with coding assistance available in the dialog. All expressions are evaluated against the current context.

Dedicated Watches view

Track any number of expressions, variables, fields and object instances in the current stack frame context. The data are updated dynamically as you progress through your debugging session.

Frames and call stack

When a breakpoint is hit or code execution is suspended, you can use the Frames panel to examine the current threads, their state, call stack, methods, and variables along with their values.

Remote debugging

As you connect to a remote host and run a debugging session, RubyMine keeps mapping between the local sources and the sources on the remote side. Several debug processes can be launched simultaneously.

Chapter 33: Configure Angular with Rails

Section 33.1: Angular with Rails 101

Step 1: Create a new Rails app

```
gem install rails -v 4.1
rails new angular_example
```

Step 2: Remove Turbolinks

Removing turbolinks requires removing it from the Gemfile.

```
gem 'turbolinks'
```

Remove the **require** from app**/**assets**/**javascripts**/**application.`js`:

```
//= require turbolinks
```

Step 3: Add AngularJS to the asset pipeline

In order to get Angular to work with the Rails asset pipeline we need to add to the Gemfile:

```
gem 'angular-rails-templates'
gem 'bower-rails'
```

Now run the command

```
bundle install
```

Add bower so that we can install the AngularJS dependency:

```
rails g bower_rails:initialize json
```

Add Angular to bower.`json`:

```json
{
  "name": "bower-rails generated dependencies",

  "dependencies": {

    "angular": "latest",
    "angular-resource": "latest",
    "bourbon": "latest",
    "angular-bootstrap": "latest",
    "angular-ui-router": "latest"
  }
}
```

Now that bower.`json` is setup with the right dependencies, let's install them:

```
bundle exec rake bower:install
```

Step 4: Organize the Angular app

Create the following folder structure in app**/**assets**/**javascript**/**angular**-**app**/**:

```
templates/
```

```
modules/
filters/
directives/
models/
services/
controllers/
```

In app/assets/javascripts/application.js, add **require** for Angular, the template helper, and the Angular app file structure. Like this:

```
//= require jquery
//= require jquery_ujs

//= require angular
//= require angular-rails-templates
//= require angular-app/app

//= require_tree ./angular-app/templates
//= require_tree ./angular-app/modules
//= require_tree ./angular-app/filters
//= require_tree ./angular-app/directives
//= require_tree ./angular-app/models
//= require_tree ./angular-app/services
//= require_tree ./angular-app/controllers
```

Step 5: Bootstrap the Angular app

Create app/assets/javascripts/angular-app/app.js.coffee:

```
@app = angular.module('app', [ 'templates' ])

@app.config([ '$httpProvider', ($httpProvider)->
$httpProvider.defaults.headers.common['X-CSRF-Token'] = $('meta[name=csrftoken]').attr('content')
])  @app.run(->   console.log 'angular app running' )
```

Create an Angular module at app/assets/javascripts/angular-app/modules/example.js.coffee.erb:

```
@exampleApp = angular.module('app.exampleApp', [     # additional dependencies here   ])   .run(->
  console.log 'exampleApp running'   )
```

Create an Angular controller for this app at app/assets/javascripts/angular-app/controllers/exampleCtrl.js.coffee:

```
angular.module('app.exampleApp').controller("ExampleCtrl", [   '$scope',   ($scope)->
console.log 'ExampleCtrl running'       $scope.exampleValue = "Hello angular and rails"  ])
```

Now add a route to Rails to pass control over to Angular. Inconfig/routes.rb:

```
Rails.application.routes.draw do   get 'example' => 'example#index' end
```

Generate the Rails controller to respond to that route:

```
rails g controller Example
```

In app/controllers/example_controller.rb:

```
class ExampleController < ApplicationController
    def index
```

```
        end
end
```

In the view, we need to specify which Angular app and which Angular controller will drive this page. So in
app**/**views**/**example**/**index.html.erb:

```
<div ng-app='app.exampleApp' ng-controller='ExampleCtrl'>

  <p>Value from ExampleCtrl:</p>
  <p>{{ exampleValue }}</p>

</div>
```

To view the app, start your Rails server and visit http://localhost:3000/example.

Chapter 34: Rails logger

Section 34.1: Rails.logger

Always use `Rails.logger.{debug|info|warn|error|fatal}` rather than **puts**. This allows your logs to fit into the standard log format, have a timestamp and have a level so you choose whether they are important enough to be shown in a specific environment. You can see the separate log files for your application under `log/` directory with your rails app environment name. like: `development.log` or `production.log` or `staging.log`

You can easily rotating rails production logs with LogRotate.You just have to do small configuration as below

Open `/etc/logrotate.conf` with your favourite linux editor `vim` or `nano` and add the below code in this file at bottom.

```
/YOUR/RAILSAPP/PATH/log/*.log {
  daily
  missingok
  rotate 7
  compress
  delaycompress
  notifempty
  copytruncate
}
```

So, **How It Works** This is fantastically easy. Each bit of the configuration does the following:

- **daily** – Rotate the log files each day. You can also use weekly or monthly here instead.
- **missingok** – If the log file doesn't exist,ignore it
- **rotate 7** – Only keep 7 days of logs around
- **compress** – GZip the log file on rotation
- **delaycompress** – Rotate the file one day, then compress it the next day so we can be sure that it won't interfere with the Rails server
- **notifempty** – Don't rotate the file if the logs are empty
- **copytruncate** – Copy the log file and then empties it. This makes sure that the log file Rails is writing to always exists so you won't get problems because the file does not actually change. If you don't use this, you would need to restart your Rails application each time.

Running Logrotate Since we just wrote this configuration, you want to test it.

To run logrotate manually, just do: **sudo** `/usr/sbin/logrotate -f /etc/logrotate.conf`

That's it.

Chapter 35: Prawn PDF

Section 35.1: Advanced Example

This is the advanced approach with example

```ruby
class FundsController < ApplicationController

  def index
    @funds = Fund.all_funds(current_user)
  end

  def show
    @fund = Fund.find(params[:id])
    respond_to do |format|
      format.html
      format.pdf do
        pdf = FundsPdf.new(@fund, view_context)
        send_data pdf.render, filename:
        "fund_#{@fund.created_at.strftime("%d/%m/%Y")}.pdf",
        type: "application/pdf"
      end
    end
  end
end
```

I above code we have this line `FundsPdf.new(@fund, view_context)`. Here we are initializing FundsPdf class with @fund instance and view_context to use helper methods in FundsPdf. FundsPdf wuld look like this

```ruby
class FundPdf < Prawn::Document

  def initialize(fund, view)
    super()
    @fund = fund
    @view = view
    upper_half
    lower_half
  end

  def upper_half
    logopath =  "#{Rails.root}/app/assets/images/logo.png"
    image logopath, :width => 197, :height => 91
    move_down 10
    draw_text "Receipt", :at => [220, 575], size: 22
    move_down 80
    text "Hello #{@invoice.customer.profile.first_name.capitalize},"
  end

  def thanks_message
    move_down 15
    text "Thank you for your order.Print this receipt as
    confirmation of your order.",
    :indent_paragraphs => 40, :size => 13
  end
end
```

This is one of the best approach to generate PDF with classes using Prawn gem.

Section 35.2: Basic Example

You need to add Gem and PDF MIME:Type inside mime_types.rb as we need to notify rails about PDF mime type.

After that we can generate Pdf with Prawn in following basic ways

This is the basic assignment

```ruby
pdf = Prawn::Document.new
pdf.text "Hello World"
pdf.render_file "assignment.pdf"
```

We can do it with Implicit Block

```ruby
Prawn::Document.generate("implicit.pdf") do
 text "Hello World"
end
```

With Explicit Block

```ruby
Prawn::Document.generate("explicit.pdf") do |pdf|
 pdf.text "Hello World"
end
```

Chapter 36: Rails API

Section 36.1: Creating an API-only application

To build a Rails application that will be an API server, you can start with a more limited subset of Rails in Rails 5.

To generate a new Rails API app:

```
rails new my_api --api
```

What `--api` does is to remove functionality that is not needed when building an API. This includes sessions, cookies, assets, and anything that makes Rails work on a browser.

It will also configure the generators so that they don't generate views, helpers, and assets when generating a new resource.

When you compare the `ApplicationController` on a web app versus an API app, you will see that the web version extends from **`ActionController::Base`**, whereas the API version extends from **`ActionController::API`**, which includes a much smaller subset of functionality.

Chapter 37: Deploying a Rails app on Heroku

Section 37.1: Deploying your application

Make sure you are in the directory that contains your Rails app, then create an app on Heroku.

```
$ heroku create example
Creating □ example... done
https://example.herokuapp.com/ | https://git.heroku.com/example.git
```

The first URL of the ouput, http://example.herokuapp.com, is the location the app is available at. The second URL, git@heroku.com:example.git, is the remote git repository URL.

This command should only be used on an initialized git repository. The heroku create command automatically adds a git remote named "heroku" pointing at this URL.

The app name argument ("example") is optional. If no app name is specified, a random name will be generated. Since Heroku app names are in a global namespace, you can expect that common names, like "blog" or "wiki", will already be taken. It's often easier to start with a default name and rename the app later.

Next, deploy your code:

```
$ git push heroku master
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Ruby app detected
remote: -----> Compiling Ruby/Rails
remote: -----> Using Ruby version: ruby-2.3.1
remote: -----> Installing dependencies using bundler 1.11.2
remote:        Running: bundle install --without development:test --path vendor/bundle --binstubs
vendor/bundle/bin -j4 --deployment
remote:        Warning: the running version of Bundler is older than the version that created the
lockfile. We suggest you upgrade to the latest version of Bundler by running `gem install bundler`.
remote:        Fetching gem metadata from https://rubygems.org/..........
remote:        Fetching version metadata from https://rubygems.org/...
remote:        Fetching dependency metadata from https://rubygems.org/..
remote:        Installing concurrent-ruby 1.0.2
remote:        Installing i18n 0.7.0
remote:        Installing rake 11.2.2
remote:        Installing minitest 5.9.0
remote:        Installing thread_safe 0.3.5
remote:        Installing builder 3.2.2
remote:        Installing mini_portile2 2.1.0
remote:        Installing erubis 2.7.0
remote:        Installing pkg-config 1.1.7
remote:        Installing rack 2.0.1
remote:        Installing nio4r 1.2.1 with native extensions
remote:        Installing websocket-extensions 0.1.2
remote:        Installing mime-types-data 3.2016.0521
remote:        Installing arel 7.0.0
remote:        Installing coffee-script-source 1.10.0
remote:        Installing execjs 2.7.0
remote:        Installing method_source 0.8.2
remote:        Installing thor 0.19.1
remote:        Installing multi_json 1.12.1
```

```
remote:         Installing puma 3.4.0 with native extensions
remote:         Installing pg 0.18.4 with native extensions
remote:         Using bundler 1.11.2
remote:         Installing sass 3.4.22
remote:         Installing tilt 2.0.5
remote:         Installing turbolinks-source 5.0.0
remote:         Installing tzinfo 1.2.2
remote:         Installing nokogiri 1.6.8 with native extensions
remote:         Installing rack-test 0.6.3
remote:         Installing sprockets 3.6.3
remote:         Installing websocket-driver 0.6.4 with native extensions
remote:         Installing mime-types 3.1
remote:         Installing coffee-script 2.4.1
remote:         Installing uglifier 3.0.0
remote:         Installing turbolinks 5.0.0
remote:         Installing activesupport 5.0.0
remote:         Installing mail 2.6.4
remote:         Installing globalid 0.3.6
remote:         Installing activemodel 5.0.0
remote:         Installing jbuilder 2.5.0
remote:         Installing activejob 5.0.0
remote:         Installing activerecord 5.0.0
remote:         Installing loofah 2.0.3
remote:         Installing rails-dom-testing 2.0.1
remote:         Installing rails-html-sanitizer 1.0.3
remote:         Installing actionview 5.0.0
remote:         Installing actionpack 5.0.0
remote:         Installing actionmailer 5.0.0
remote:         Installing railties 5.0.0
remote:         Installing actioncable 5.0.0
remote:         Installing sprockets-rails 3.1.1
remote:         Installing coffee-rails 4.2.1
remote:         Installing jquery-rails 4.1.1
remote:         Installing rails 5.0.0
remote:         Installing sass-rails 5.0.5
remote:         Bundle complete! 15 Gemfile dependencies, 54 gems now installed.
remote:         Gems in the groups development and test were not installed.
remote:         Bundled gems are installed into ./vendor/bundle.
remote:         Bundle completed (31.86s)
remote:         Cleaning up the bundler cache.
remote:         Warning: the running version of Bundler is older than the version that created the
lockfile. We suggest you upgrade to the latest version of Bundler by running `gem install bundler`.
remote: -----> Preparing app for Rails asset pipeline
remote:         Running: rake assets:precompile
remote:         I, [2016-07-08T17:08:57.046245 #1222]  INFO -- : Writing
/tmp/build_49ba6c877f5502cd4029406e981f90b4/public/assets/application-1bf5315c71171ad5f9cbef00193d56b
7e45263ddc64caf676ce988cfbb6570bd.js
remote:         I, [2016-07-08T17:08:57.046951 #1222]  INFO -- : Writing
/tmp/build_49ba6c877f5502cd4029406e981f90b4/public/assets/application-1bf5315c71171ad5f9cbef00193d56b
7e45263ddc64caf676ce988cfbb6570bd.js.gz
remote:         I, [2016-07-08T17:08:57.060208 #1222]  INFO -- : Writing
/tmp/build_49ba6c877f5502cd4029406e981f90b4/public/assets/application-
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css
remote:         I, [2016-07-08T17:08:57.060656 #1222]  INFO -- : Writing
/tmp/build_49ba6c877f5502cd4029406e981f90b4/public/assets/application-
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855.css.gz
remote:         Asset precompilation completed (4.06s)
remote:         Cleaning assets
remote:         Running: rake assets:clean
remote:
remote: ###### WARNING:
remote:         No Procfile detected, using the default web server.
```

```
remote:           We recommend explicitly declaring how to boot your server process via a Procfile.
remote:           https://devcenter.heroku.com/articles/ruby-default-web-server
remote:
remote: -----> Discovering process types
remote:           Procfile declares types      -> (none)
remote:           Default types for buildpack -> console, rake, web, worker
remote:
remote: -----> Compressing...
remote:           Done: 29.2M
remote: -----> Launching...
remote:           Released v5
remote:           https://example.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/example.git
 * [new branch]       master -> master
```

If you are using the database in your application you need to manually migrate the database by running:

```
$ heroku run rake db:migrate
```

Any commands after `heroku run` will be executed on a Heroku dyno. You can obtain an interactive shell session by running:

```
$ heroku run bash
```

Ensure you have one dyno running the web process type:

```
$ heroku ps:scale web=1
```

The heroku ps command lists the running dynos of your application:

```
$ heroku ps
=== web (Standard-1X): bin/rails server -p $PORT -e $RAILS_ENV (1)
web.1: starting 2016/07/08 12:09:06 -0500 (~ 2s ago)
```

You can now visit the app in our browser with `heroku open`.

```
$ heroku open
```

Heroku gives you a default web URL in the `herokuapp.com` domain. When you are ready to scale up for production, you can add your own custom domain.

Section 37.2: Managing Production and staging environments for a Heroku

Every Heroku app runs in at least two environments: on Heroku (we'll call that production) and on your local machine (development). If more than one person is working on the app, then you've got multiple development environments - one per machine, usually. Usually, each developer will also have a test environment for running tests. Unfortunately, this approach breaks down as the environments become less similar. Windows and Macs, for instance, both provide different environments than the Linux stack on Heroku, so you can't always be sure that code that works in your local development environment will work the same way when you deploy it to production.

The solution is to have a staging environment that is as similar to production as is possible. This can be achieved by creating a second Heroku application that hosts your staging application. With staging, you can check your code in a

production-like setting before having it affect your actual users.

Starting from scratch

Assume you have an application running on your local machine, and you're ready to push it to Heroku. We'll need to create both remote environments, staging and production. To get in the habit of pushing to staging first, we'll start with this:

```
$ heroku create --remote staging
Creating strong-river-216.... done
http://strong-river-216.heroku.com/ | https://git.heroku.com/strong-river-216.git
Git remote staging added
```

By default, the heroku CLI creates projects with a heroku git remote. Here, we're specifying a different name with the --remote flag, so pushing code to Heroku and running commands against the app look a little different than the normal git push heroku master:

```
 $ git push staging master
...
 $ heroku ps --remote staging
=== web: `bundle exec puma -C config/puma.rb``
web.1: up for 21s
```

Once your staging app is up and running properly, you can create your production app:

```
$ heroku create --remote production
Creating fierce-ice-327.... done
http://fierce-ice-327.heroku.com/ | https://git.heroku.com/fierce-ice-327.git
Git remote production added
$ git push production master
...
$ heroku ps --remote production
=== web: `bundle exec puma -C config/puma.rb
web.1: up for 16s
```

And with that, you've got the same codebase running as two separate Heroku apps – one staging and one production, set up identically. Just remember you will have to specify which app you are going to operate on your daily work. You can either use flag '--remote' or use your git config to specify a default app.

Chapter 38: ActiveSupport

Section 38.1: Core Extensions: String Access

String#at

Returns a substring of a string object. Same interface as `String#[]`.

```
str = "hello"
str.at(0)      # => "h"
str.at(1..3)   # => "ell"
str.at(-2)     # => "l"
str.at(-2..-1) # => "lo"
str.at(5)      # => nil
str.at(5..-1)  # => ""
```

String#from

Returns a substring from the given position to the end of the string.

```
str = "hello"
str.from(0)  # => "hello"
str.from(3)  # => "lo"
str.from(-2) # => "lo"
```

String#to

Returns a substring from the beginning of the string to the given position.
If the position is negative, it is counted from the end of the string.

```
str = "hello"
str.to(0)  # => "h"
str.to(3)  # => "hell"
str.to(-2) # => "hell"
```

`from` and `to` can be used in tandem.

```
str = "hello"
str.from(0).to(-1) # => "hello"
str.from(1).to(-2) # => "ell"
```

String#first

Returns the first character, or a given number of characters up to the length of the string.

```
str = "hello"
str.first     # => "h"
str.first(1)  # => "h"
str.first(2)  # => "he"
str.first(0)  # => ""
str.first(6)  # => "hello"
```

String#last

Returns the last character, or a given number of characters from the end of the string counting backwards.

```
str = "hello"
str.last     # => "o"
str.last(1)  # => "o"
```

```
str.last(2)  # => "lo"
str.last(0)  # => ""
str.last(6)  # => "hello"
```

Section 38.2: Core Extensions: String to Date/Time Conversion

String#to_time

Converts a string to a Time value. The `form` parameter can be either `:utc` or `:local`, defaults to `:local`.

```
"13-12-2012".to_time             # => 2012-12-13 00:00:00 +0100
"06:12".to_time                  # => 2012-12-13 06:12:00 +0100
"2012-12-13 06:12".to_time       # => 2012-12-13 06:12:00 +0100
"2012-12-13T06:12".to_time       # => 2012-12-13 06:12:00 +0100
"2012-12-13T06:12".to_time(:utc) # => 2012-12-13 06:12:00 UTC
"12/13/2012".to_time             # => ArgumentError: argument out of range
```

String#to_date

Converts a string to a Date value.

```
"1-1-2012".to_date   # => Sun, 01 Jan 2012
"01/01/2012".to_date # => Sun, 01 Jan 2012
"2012-12-13".to_date # => Thu, 13 Dec 2012
"12/13/2012".to_date # => ArgumentError: invalid date
```

String#to_datetime

Converts a string to a DateTime value.

```
"1-1-2012".to_datetime            # => Sun, 01 Jan 2012 00:00:00 +0000
"01/01/2012 23:59:59".to_datetime # => Sun, 01 Jan 2012 23:59:59 +0000
"2012-12-13 12:50".to_datetime    # => Thu, 13 Dec 2012 12:50:00 +0000
"12/13/2012".to_datetime          # => ArgumentError: invalid date
```

Section 38.3: Core Extensions: String Exclusion

String#exclude?

The inverse of **String**#include?

```
"hello".exclude? "lo" # => false
"hello".exclude? "ol" # => true
"hello".exclude? ?h   # => false
```

Section 38.4: Core Extensions: String Filters

String#squish

Returns a version of the given string without leading or trailing whitespace, and combines all consecutive whitespace in the interior to single spaces. Destructive version `squish!` operates directly on the string instance.

Handles both ASCII and Unicode whitespace.

```
%{ Multi-line
   string }.squish              # => "Multi-line string"
" foo   bar    \n   \t   boo".squish # => "foo bar boo"
```

String#remove

Returns a new string with all occurrences of the patterns removed. Destructive version `remove!` operates directly on the given string.

```ruby
str = "foo bar test"
str.remove(" test")                # => "foo bar"
str.remove(" test", /bar/)         # => "foo "
```

String#truncate

Returns a copy of a given string truncated at a given length if the string is longer than the length.

```ruby
'Once upon a time in a world far far away'.truncate(27)
# => "Once upon a time in a wo..."
```

Pass a string or regexp `:separator` to truncate at a natural break

```ruby
'Once upon a time in a world far far away'.truncate(27, separator: ' ')
# => "Once upon a time in a..."

'Once upon a time in a world far far away'.truncate(27, separator: /\s/)
# => "Once upon a time in a..."
```

String#truncate_words

Returns a string truncated after a given number of words.

```ruby
'Once upon a time in a world far far away'.truncate_words(4)
# => "Once upon a time..."
```

Pass a string or regexp to specify a different separator of words

```ruby
'Once<br>upon<br>a<br>time<br>in<br>a<br>world'.truncate_words(5, separator: '<br>')
# => "Once<br>upon<br>a<br>time<br>in..."
```

The last characters will be replaced with the `:omission` string (defaults to "…")

```ruby
'And they found that many people were sleeping better.'.truncate_words(5, omission: '...
(continued)')
# => "And they found that many... (continued)"
```

String#strip_heredoc

Strips indentation in heredocs. Looks for the least-indented non-empty line and removes that amount of leading whitespace.

```ruby
if options[:usage]
  puts <<-USAGE.strip_heredoc
    This command does such and such.

    Supported options are:
      -h         This message
    ...
  USAGE
end
```

the user would see

NO_FORMAT]This command does such and such. Supported options are: -h This message ...

Section 38.5: Core Extensions: String Inflection

String#pluralize

Returns of plural form of the string. Optionally takes a `count` parameter and returns singular form if `count == 1`. Also accepts a `locale` parameter for language-specific pluralization.

```
'post'.pluralize             # => "posts"
'octopus'.pluralize          # => "octopi"
'sheep'.pluralize            # => "sheep"
'words'.pluralize            # => "words"
'the blue mailman'.pluralize # => "the blue mailmen"
'CamelOctopus'.pluralize     # => "CamelOctopi"
'apple'.pluralize(1)         # => "apple"
'apple'.pluralize(2)         # => "apples"
'ley'.pluralize(:es)         # => "leyes"
'ley'.pluralize(1, :es)      # => "ley"
```

String#singularize

Returns the singular form of the string. Accepts an optional `locale` parameter.

```
'posts'.singularize             # => "post"
'octopi'.singularize            # => "octopus"
'sheep'.singularize             # => "sheep"
'word'.singularize              # => "word"
'the blue mailmen'.singularize  # => "the blue mailman"
'CamelOctopi'.singularize       # => "CamelOctopus"
'leyes'.singularize(:es)        # => "ley"
```

String#constantize

Tries to find a declared constant with the name specified in the string. It raises a `NameError` when the name is not in CamelCase or is not initialized.

```
'Module'.constantize  # => Module
'Class'.constantize   # => Class
'blargle'.constantize # => NameError: wrong constant name blargle
```

String#safe_constantize

Performs a `constantize` but returns `nil` instead of raising `NameError`.

```
'Module'.safe_constantize  # => Module
'Class'.safe_constantize   # => Class
'blargle'.safe_constantize # => nil
```

String#camelize

Converts strings to UpperCamelCase by default, if `:lower` is given as param converts to lowerCamelCase instead.

alias: `camelcase`

Note: will also convert / to :: which is useful for converting paths to namespaces.

```
'active_record'.camelize            # => "ActiveRecord"
'active_record'.camelize(:lower)    # => "activeRecord"
'active_record/errors'.camelize     # => "ActiveRecord::Errors"
```

```ruby
'active_record/errors'.camelize(:lower) # => "activeRecord::Errors"
```

String#titleize

Capitalizes all the words and replaces some characters in the string to create a nicer looking title.

alias: `titlecase`

```ruby
'man from the boondocks'.titleize # => "Man From The Boondocks"
'x-men: the last stand'.titleize  # => "X Men: The Last Stand"
```

String#underscore

Makes an underscored, lowercase form from the expression in the string. The reverse of `camelize`.

Note: `underscore` will also change `::` to `/` to convert namespaces to paths.

```ruby
'ActiveModel'.underscore          # => "active_model"
'ActiveModel::Errors'.underscore  # => "active_model/errors"
```

String#dasherize

Replaces underscores with dashes in the string.

```ruby
'puni_puni'.dasherize # => "puni-puni"
```

String#demodulize

Removes the module part from the constant expression in the string.

```ruby
'ActiveRecord::CoreExtensions::String::Inflections'.demodulize # => "Inflections"
'Inflections'.demodulize                                       # => "Inflections"
'::Inflections'.demodulize                                     # => "Inflections"
''.demodulize                                                  # => ''
```

String#deconstantize

Removes the rightmost segment from the constant expression in the string.

```ruby
'Net::HTTP'.deconstantize    # => "Net"
'::Net::HTTP'.deconstantize  # => "::Net"
'String'.deconstantize       # => ""
'::String'.deconstantize     # => ""
''.deconstantize             # => ""
```

String#parameterize

Replaces special characters in a string so that it may be used as part of a 'pretty' URL.

```ruby
"Donald E. Knuth".parameterize # => "donald-e-knuth"
```

Preserve the case of the characters in a string with the **:preserve_case** argument.

```ruby
"Donald E. Knuth".parameterize(preserve_case: true) # => "Donald-E-Knuth"
```

A very common use-case for `parameterize` is to override the `to_param` method of an ActiveRecord model to support more descriptive url slugs.

```ruby
class Person < ActiveRecord::Base
  def to_param
    "#{id}-#{name.parameterize}"
```

```
    end
end
```

```
Person.find(1).to_param # => "1-donald-e-knuth"
```

String#tableize

Creates the name of a table like Rails does for models to table names. Pluralizes the last word in the string.

```
'RawScaledScorer'.tableize # => "raw_scaled_scorers"
'ham_and_egg'.tableize     # => "ham_and_eggs"
'fancyCategory'.tableize   # => "fancy_categories"
```

String#classify

Returns a class name string from a plural table name like Rails does for table names to models.

```
'ham_and_eggs'.classify # => "HamAndEgg"
'posts'.classify        # => "Post"
```

String#humanize

Capitalizes the first word, turns underscores into spaces, and strips a trailing _id if present.

```
'employee_salary'.humanize              # => "Employee salary"
'author_id'.humanize                    # => "Author"
'author_id'.humanize(capitalize: false) # => "author"
'_id'.humanize                          # => "Id"
```

String#upcase_first

Converts just the first character to uppercase.

```
'what a Lovely Day'.upcase_first # => "What a Lovely Day"
'w'.upcase_first                 # => "W"
''.upcase_first                  # => ""
```

String#foreign_key

Creates a foreign key name from a class name. Pass **false** param to disable adding _ between name and id.

```
'Message'.foreign_key        # => "message_id"
'Message'.foreign_key(false) # => "messageid"
'Admin::Post'.foreign_key    # => "post_id"
```

Chapter 39: Form Helpers

Rails provides view helpers for generating form markup.

Section 39.1: Creating a search form

To create a search form, enter the following code

```
<%= form_tag("/search", method: "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

- `form_tag`: This is the default helper for creating a form. It's first parameter, `/search` is the action and the second parameter specifies the HTTP method. For search forms, it is important to always use the method `get`
- `label_tag`: This helper creates an html `<label>` tag.
- `text_field_tag`: This will create an input element with type `text`
- `submit_tag`: This creates an input element with type `submit`

Section 39.2: Dropdown

Standard example:

```
@models = Model.all
select_tag "models", options_from_collection_for_select(@models, "id", "name"), {}
```

This will generate the following HTML: David

The last argument are options, which accepts the following:

```
{
multiple: false,
disabled: false,
include_blank: false,
prompt: false
}
```

More examples can be found: http://apidock.com/rails/ActionView/Helpers/FormTagHelper/select_tag

Section 39.3: Helpers for form elements

Checkboxes

```
<%= check_box_tag(:pet_dog) %>
<%= label_tag(:pet_dog, "I own a dog") %>
<%= check_box_tag(:pet_cat) %>
<%= label_tag(:pet_cat, "I own a cat") %>
```

This will generate the following html

```
<input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
```

```
<label for="pet_cat">I own a cat</label>
```

Radio Buttons

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 18") %>
<%= radio_button_tag(:age, "adult") %>
<%= label_tag(:age_adult, "I'm over 18") %>
```

This generates the following HTML

```
<input id="age_child" name="age" type="radio" value="child" />
<label for="age_child">I am younger than 18</label>
<input id="age_adult" name="age" type="radio" value="adult" />
<label for="age_adult">I'm over 18</label>
```

Text Area

To create a larger text box, it is recommended to use the `text_area_tag`

```
<%= text_area_tag(:message, "This is a longer text field", size: "25x6") %>
```

This will create the following HTML

```
<textarea id="message" name="message" cols="25" rows="6">This is a longer text field</textarea>
```

Number Field

This will create an `input<type="number">` element

```
<%= number_field :product, :rating %>
```

To specify a range of values, we can use the `in:` option

```
<%= number_field :product, :rating, in: 1..10 %>
```

Password Field

Sometimes you want the characters typed by the user to be masked. This will generate an `<input type="password">`

```
<%= password_field_tag(:password) %>
```

Email Field

This will create an `<input type="email">`

```
<%= email_field(:user, :email) %>
```

Telephone Field

This will create an `<input type="tel">`.

```
<%= telephone_field :user, :phone %>
```

Date Helpers

- input**[**type=**"date"]**

  ```erb
  <%= date_field(:user, :reservation) %>
  ```

- input**[**type=**"week"]**

  ```erb
  <%= week_field(:user, :reservation) %>
  ```

- input**[**type=**"year"]**

  ```erb
  <%= year_field(:user, :reservation) %>
  ```

- input**[**type=**"time"]**

  ```erb
  <%= time_field(:user, :check_in) %>
  ```

Chapter 40: ActiveRecord Transactions

Section 40.1: Basic example

For example:

```
ActiveRecord::Base.transaction do
  david.withdrawal(100)
  mary.deposit(100)
end
```

This example will only take money from David and give it to Mary if neither withdrawal nor deposit raise an exception. Exceptions will force a ROLLBACK that returns the database to the state before the transaction began. Be aware, though, that the objects will not have their instance data returned to their pre-transactional state.

Section 40.2: Different ActiveRecord classes in a single transaction

Though the transaction class method is called on some ActiveRecord class, the objects within the transaction block need not all be instances of that class. This is because transactions are per-database connection, not per-model.

In this example a balance record is transactionally saved even though transaction is called on the Account class:

```
Account.transaction do
  balance.save!
  account.save!
end
```

The transaction method is also available as a model instance method. For example, you can also do this:

```
balance.transaction do
  balance.save!
  account.save!
end
```

Section 40.3: Multiple database connections

A transaction acts on a single database connection. If you have multiple class-specific databases, the transaction will not protect interaction among them. One workaround is to begin a transaction on each class whose models you alter:

```
Student.transaction do
  Course.transaction do
    course.enroll(student)
    student.units += course.units
  end
end
```

This is a poor solution, but fully distributed transactions are beyond the scope of ActiveRecord.

Section 40.4: save and destroy are automatically wrapped in a transaction

Both #save and #destroy come wrapped in a transaction that ensures that whatever you do in validations or

callbacks will happen under its protected cover. So you can use validations to check for values that the transaction depends on or you can raise exceptions in the callbacks to rollback, including `after_*` callbacks.

As a consequence changes to the database are not seen outside your connection until the operation is complete. For example, if you try to update the index of a search engine in `after_save` the indexer won't see the updated record. The `after_commit` callback is the only one that is triggered once the update is committed.

Section 40.5: Callbacks

There are two types of callbacks associated with committing and rolling back transactions: `after_commit` and `after_rollback`.

`after_commit` callbacks are called on every record saved or destroyed within a transaction immediately after the transaction is committed. `after_rollback` callbacks are called on every record saved or destroyed within a transaction immediately after the transaction or savepoint is rolled back.

These callbacks are useful for interacting with other systems since you will be guaranteed that the callback is only executed when the database is in a permanent state. For example, `after_commit` is a good spot to put in a hook to clearing a cache since clearing it from within a transaction could trigger the cache to be regenerated before the database is updated.

Section 40.6: Rolling back a transaction

`ActiveRecord::Base.transaction` uses the `ActiveRecord::Rollback` exception to distinguish a deliberate rollback from other exceptional situations. Normally, raising an exception will cause the `.transaction` method to rollback the database transaction and pass on the exception. But if you raise an `ActiveRecord::Rollback` exception, then the database transaction will be rolled back, without passing on the exception.

For example, you could do this in your controller to rollback a transaction:

```ruby
class BooksController < ActionController::Base
  def create
    Book.transaction do
      book = Book.new(params[:book])
      book.save!
      if today_is_friday?
        # The system must fail on Friday so that our support department
        # won't be out of job. We silently rollback this transaction
        # without telling the user.
        raise ActiveRecord::Rollback, "Call tech support!"
      end
    end
    # ActiveRecord::Rollback is the only exception that won't be passed on
    # by ActiveRecord::Base.transaction, so this line will still be reached
    # even on Friday.
    redirect_to root_url
  end
end
```

Chapter 41: RSpec and Ruby on Rails

Section 41.1: Installing RSpec

If you want to use RSpec for a Rails project, you should use the [rspec-rails](#) gem, which can generate helpers and spec files for you automatically (for example, when you create models, resources or scaffolds using `rails generate`).

Add `rspec-rails` to both the **:development** and **:test** groups in the `Gemfile`:

```ruby
group :development, :test do
  gem 'rspec-rails', '~> 3.5'
end
```

Run `bundle` to install the dependencies.

Initialize it with:

```
rails generate rspec:install
```

This will create a `spec/` folder for your tests, along with the following configuration files:

- `.rspec` contains default options for the command-line `rspec` tool
- `spec/spec_helper.rb` includes basic RSpec configuration options
- `spec/rails_helper.rb` adds further configuration options that are more specific to use RSpec and Rails together.

All these files are written with sensible defaults to get you started, but you can add features and change configurations to suit your needs as your test suite grows.

Chapter 42: Decorator pattern

Section 42.1: Decorating a Model using Draper

Draper automatically matches up models with their decorators by convention.

```ruby
# app/decorators/user_decorator.rb
class UserDecorator < Draper::Decorator
  def full_name
    "#{object.first_name} #{object.last_name}"
  end

  def created_at
    Time.use_zone(h.current_user.timezone) do
      object.created_at.strftime("%A, %d %b %Y %l:%M %p")
    end
  end
end
```

Given a `@user` variable containing an ActiveRecord object, you can access your decorator by calling *#decorate* on the `@user`, or by specifying the Draper class if you want to be specific.

```erb
<% user = @user.decorate %><!-- OR -->
<% user = UserDecorator.decorate(@user) %>
<h1><%= user.full_name %></h1>
<h3>joined: <%= user.created_at %></h3>
```

Section 42.2: Decorating a Model using SimpleDelegator

Most Rails developers start by modifying their model information within the template itself:

```erb
<h1><%= "#{ @user.first_name } #{ @user.last_name }" %></h1>
<h3>joined: <%= @user.created_at.in_time_zone(current_user.timezone).strftime("%A, %d %b %Y %l:%M
%p") %></h3>
```

For models with a lot of data, this can quickly become cumbersome and lead to copy-pasting logic from one template to another.

This example uses `SimpleDelegator` from the stdlib.

All requests to a `SimpleDelegator` object are passed to the parent object by default. You can override any method with presentation logic, or you can add new methods that are specific to this view.

`SimpleDelegator` provides two methods: `__setobj__` to set what object is being delegated to, and `__getobj__` to get that object.

```ruby
class UserDecorator < SimpleDelegator
  attr_reader :view
  def initialize(user, view)
    __setobj__ @user
    @view = view
  end

  # new methods can call methods on the parent implicitly
  def full_name
    "#{ first_name } #{ last_name }"
```

```ruby
  end

  # however, if you're overriding an existing method you need
  # to use __getobj__
  def created_at
    Time.use_zone(view.current_user.timezone) do
      __getobj__.created_at.strftime("%A, %d %b %Y %l:%M %p")
    end
  end
end
```

Some decorators rely on magic to wire-up this behavior, but you can make it more obvious where the presentation logic is coming from by initializing the object on the page.

```erb
<% user = UserDecorator.new(@user, self) %>
<h1><%= user.full_name %></h1>
<h3>joined: <%= user.created_at %></h3>
```

By passing a reference to the view object into the decorator, we can still access all of the rest of the view helpers while building the presentation logic without having to include it.

Now the view template is only concerned with inserting data into the page, and it is much more clear.

Chapter 43: Elasticsearch

Section 43.1: Searchkick

If you want to setup quickly elasticsearch you can use the searchkick gem :

```
gem 'searchkick'
```

Add searchkick to models you want to search.

```
class Product < ActiveRecord::Base
  searchkick
end
```

Add data to the search index.

```
Product.reindex
```

And to query, use:

```
products = Product.search "apples"
products.each do |product|
  puts product.name
end
```

Pretty quick, elasticsearch knowledge not required ;-)

More information here : https://github.com/ankane/searchkick

Section 43.2: Installation and testing

The first thing you want to do for local development is install ElasticSearch in your machine and test it to see if it is running. It requires Java to be installed. The installation is pretty straightforward:

- Mac OS X: `brew install elasticsearch`
- Ubuntu: `sudo apt-get install elasticsearch`

Then start it:

- Mac OS X: `brew services start elasticsearch`
- Ubuntu: `sudo service elasticsearch start`

For testing it, the easiest way is with `curl`. It might take a few seconds for it to start, so don't panic if you don't get any response at first.

```
curl localhost:9200
```

Example response:

```
{
  "name" : "Hydro-Man",
  "cluster_name" : "elasticsearch_gkbonetti",
  "version" : {
    "number" : "2.3.5",
    "build_hash" : "90f439ff60a3c0f497f91663701e64ccd01edbb4",
```

```
    "build_timestamp" : "2016-07-27T10:36:52Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.0"
  },
  "tagline" : "You Know, for Search"
}
```

Section 43.3: Setting up tools for development

When you are getting started with ElasticSearch (ES) it might be good to have a graphical tool that helps you explore your data. A plugin called <u>elasticsearch-head</u> does just that. To install it, do the following:

- Find out in which folder ES is installed: `ls -l $(which elasticsearch)`
- `cd` into this folder and run the plugin installation binary: `elasticsearch/bin/plugin -install mobz/elasticsearch-head`
- Open `http://localhost:9200/_plugin/head/` in your browser

If everything worked as expected you should be seeing a nice GUI where you can explore your data.

Section 43.4: Introduction

ElasticSearch has a well-documented JSON API, but you'll probably want to use some libraries that handle that for you:

- <u>Elasticsearch</u> - the official low level wrapper for the HTTP API

- <u>Elasticsearch-rails</u> - the official high level Rails integration that helps you to connect your Rails models with ElasticSearch using either ActiveRecord or Repository pattern

- <u>Chewy</u> - An alternative, non-official high level Rails integration that is very popular and arguably has better documentation

Let's use the first option for testing the connection:

`gem install elasticsearch`

Then fire up the ruby terminal and try it out:

```ruby
require 'elasticsearch'

client = Elasticsearch::Client.new log: true
# by default it connects to http://localhost:9200

client.transport.reload_connections!
client.cluster.health

client.search q: 'test'
```

Chapter 44: React with Rails using react-rails gem

Section 44.1: React installation for Rails using rails_react gem

Add react-rails to your Gemfile:

```
gem 'react-rails'
```

And install:

```
bundle install
```

Next, run the installation script:

```
rails g react:install
```

This will:

create a components.js manifest file and a app/assets/javascripts/components/ directory, where you will put your components place the following in your application.js:

```
//= require react
//= require react_ujs
//= require components
```

Section 44.2: Using react_rails within your application

React.js builds

You can pick which React.js build (development, production, with or without add-ons) to serve in each environment by adding a config. Here are the defaults:

```ruby
# config/environments/development.rb
MyApp::Application.configure do
  config.react.variant = :development
end


# config/environments/production.rb
MyApp::Application.configure do
  config.react.variant = :production
end
```

To include add-ons, use this config:

```ruby
MyApp::Application.configure do
  config.react.addons = true # defaults to false
end
```

After restarting your Rails server, //= require react will provide the build of React.js which was specified by the configurations.

react-rails offers a few other options for versions & builds of React.js. See VERSIONS.md for more info about using the react-source gem or dropping in your own copies of React.js.

JSX

After installing react-rails, restart your server. Now, .js.jsx files will be transformed in the asset pipeline.

BabelTransformer options

You can use babel's transformers and custom plugins, and pass options to the babel transpiler adding following configurations:

```
config.react.jsx_transform_options = {
  blacklist: ['spec.functionName', 'validation.react', 'strict'], # default options
  optional: ["transformerName"],   # pass extra babel options
  whitelist: ["useStrict"] # even more options[enter link description here][1]
}
```

Under the hood, react-rails uses ruby-babel-transpiler, for transformation.

Section 44.3: Rendering & mounting

react-rails includes a view helper (react_component) and an unobtrusive JavaScript driver (react_ujs) which work together to put React components on the page. You should require the UJS driver in your manifest after react (and after turbolinks if you use Turbolinks).

The view helper puts a div on the page with the requested component class & props. For example:

```
<%= react_component('HelloMessage', name: 'John') %>
<!-- becomes: -->
<div data-react-class="HelloMessage" data-react-props="{&quot;name&quot;:&quot;John&quot;}"></div>
```

On page load, the react_ujs driver will scan the page and mount components using data-react-class and data-react-props.

If Turbolinks is present components are mounted on the page:change event, and unmounted on page:before-unload. Turbolinks >= 2.4.0 is recommended because it exposes better events.

In case of Ajax calls, the UJS mounting can be triggered manually by calling from javascript:

ReactRailsUJS.mountComponents() The view helper's signature is:

```
react_component(component_class_name, props={}, html_options={})
```

component_class_name is a string which names a globally-accessible component class. It may have dots (eg, "MyApp.Header.MenuItem").

```
    `props` is either an object that responds to `#to_json` or an    already-stringified JSON object
(eg, made with Jbuilder, see note    below).
```

html_options may include: tag: to use an element other than a div to embed data-react-class and data-react-props. prerender: **true** to render the component on the server. **other Any other arguments (eg class:, id:) are passed through to content_tag.

Chapter 45: Rails Cookbook - Advanced rails recipes/learnings and coding techniques

Section 45.1: Playing with Tables using rails console

View tables

```
ActiveRecord::Base.connection.tables
```

Delete any table.

```
    ActiveRecord::Base.connection.drop_table("users")
------------OR----------------------
ActiveRecord::Migration.drop_table(:users)
------------OR--------------------
ActiveRecord::Base.connection.execute("drop table users")
```

Remove index from existing column

```
ActiveRecord::Migration.remove_index(:users, :name => 'index_users_on_country')
```

where country is a column name in the migration file with **already** added index in users table as shown below:

```
t.string :country,add_index: true
```

Remove foreign key constraint

```
ActiveRecord::Base.connection.remove_foreign_key('food_items', 'menus')
```

where menus has_many food_items and their respective migrations too.

Add column

```
ActiveRecord::Migration.remove_column :table_name, :column_name
```

for example:

```
ActiveRecord::Migration.add_column :profiles, :profile_likes, :integer, :default => 0
```

Section 45.2: Rails methods - returning boolean values

Any method in Rails model can return boolean value.

simple method-

```
##this method return ActiveRecord::Relation
def check_if_user_profile_is_complete
  User.includes( :profile_pictures,:address,:contact_detail).where("user.id = ?",self)
end
```

Again simple method returning boolean value-

```ruby
##this method return Boolean(NOTE THE !! signs before result)
def check_if_user_profile_is_complete
  !!User.includes( :profile_pictures,:address,:contact_detail).where("user.id = ?",self)
end
```

So,the same method will now return boolean instead of anything else :).

Section 45.3: Handling the error - undefined method `where' for #<Array:0x000000071923f8>

Sometimes we want to use a `where` query on a a collection of records returned which is not `ActiveRecord::Relation`.Hence we get the above error as `Where` clause is know to `ActiveRecord` and not to **Array**.

There is a precise solution for this by using `Joins`.

EXAMPLE:

> Suppose i need to find all user profiles(UserProfile) which are active which is not a user(User) with an id=10.

```ruby
UserProfiles.includes(:user=>:profile_pictures]).where(:active=>true).map(&:user).where.not(:id=>10
)
```

So above query will fail after `map` as `map` will return an **array** which will **not** work with `where` clause.

But using joins,will make it work,

```ruby
UserProfiles.includes(:user=>:profile_pictures]).where(:active=>true).joins(:user).where.not(:id=>1
0)
```

As `joins` will output similar records like `map` but they will be `ActiveRecord` and **not** an **Array**.

Chapter 46: Multipurpose ActiveRecord columns

Section 46.1: Saving an object

If you have an attribute that needs to be saved and retrieved to database as an object, then specify the name of that attribute using the `serialize` method and it will be handled automatically.

The attribute must be declared as a `text` field.

In the model you must declare the type of the field (**Hash** or **Array**)

More info at: serialize >> apidock.com

Section 46.2: How To

In your migration

```ruby
class Users < ActiveRecord::Migration[5.0]
  def change
    create_table :users do |t|
      ...
      t.text :preference
      t.text :tag
      ...
      t.timestamps
    end
  end
end
```

In your model

```ruby
class User < ActiveRecord::Base
    serialize :preferences, Hash
    serialize :tags, Array
end
```

Chapter 47: Class Organization

Section 47.1: Service Class

Controller is an entry point to our application. However, it's not the only possible entry point. I would like to have my logic accessible from:

- Rake tasks
- background jobs
- console
- tests

If I throw my logic into a controller it won't be accessible from all these places. So let's try "skinny controller, fat model" approach and move the logic to a model. But which one? If a given piece of logic involves `User`, `Cart` and `Product` models – where should it live?

A class which inherits from **`ActiveRecord::Base`** already has a lot of responsibilities. It handles query interface, associations and validations. If you add even more code to your model it will quickly become an unmaintainable mess with hundreds of public methods.

A service is just a regular Ruby object. Its class does not have to inherit from any specific class. Its name is a verb phrase, for example `CreateUserAccount` rather than `UserCreation` or `UserCreationService`. It lives in app/services directory. You have to create this directory by yourself, but Rails will autoload classes inside for you.

A service object does one thing

A service object (aka method object) performs one action. It holds the business logic to perform that action. Here is an example:

```ruby
# app/services/accept_invite.rb
class AcceptInvite
  def self.call(invite, user)
    invite.accept!(user)
    UserMailer.invite_accepted(invite).deliver
  end
end
```

The three conventions I follow are:

Services go under the app**/**`services directory`. I encourage you to use subdirectories for business logic-heavy domains. For instance:

- The file app**/**services**/**invite**/**accept.`rb` will define **`Invite::Accept`** while app**/**services**/**invite**/**create.`rb` will define **`Invite::Create`**
- Services start with a verb (and do not end with Service): `ApproveTransaction`, `SendTestNewsletter`, `ImportUsersFromCsv`
- Services respond to the `call` method. I found using another verb makes it a bit redundant: `ApproveTransaction.approve()` does not read well. Also, the `call` method is the de facto method for **`lambda`**, procs, and method objects.

Benefits

Service objects show what my application does

I can just glance over the services directory to see what my application does: `ApproveTransaction`, `CancelTransaction`, `BlockAccount`, `SendTransactionApprovalReminder`…

A quick look into a service object and I know what business logic is involved. I don't have to go through the controllers, `ActiveRecord` model callbacks and observers to understand what "approving a transaction" involves.

Clean-up models and controllers

Controllers turn the request (params, session, cookies) into arguments, pass them down to the service and redirect or render according to the service response.

```ruby
class InviteController < ApplicationController
  def accept
    invite = Invite.find_by_token!(params[:token])
    if AcceptInvite.call(invite, current_user)
      redirect_to invite.item, notice: "Welcome!"
    else
      redirect_to '/', alert: "Oopsy!"
    end
  end
end
```

Models only deal with associations, scopes, validations and persistence.

```ruby
class Invite < ActiveRecord::Base
  def accept!(user, time=Time.now)
    update_attributes!(
      accepted_by_user_id: user.id,
      accepted_at: time
    )
  end
end
```

This makes models and controllers much easier to test and maintain!

When to use Service Class

Reach for Service Objects when an action meets one or more of these criteria:

- The action is complex (e.g. closing the books at the end of an accounting period)
- The action reaches across multiple models (e.g. an e-commerce purchase using Order, CreditCard and Customer objects)
- The action interacts with an external service (e.g. posting to social networks)
- The action is not a core concern of the underlying model (e.g. sweeping up outdated data after a certain time period).
- There are multiple ways of performing the action (e.g. authenticating with an access token or password).

Sources

[Adam Niedzielski Blog](#)

[Brew House Blog](#)

[Code Climate Blog](#)

Section 47.2: Model Class

```ruby
class Post < ActiveRecord::Base
  belongs_to :user
  has_many :comments

  validates :user, presence: true
  validates :title, presence: true, length: { in: 6..40 }

  scope :topic, -> (topic) { joins(:topics).where(topic: topic) }

  before_save :update_slug
  after_create :send_welcome_email

  def publish!
    update(published_at: Time.now, published: true)
  end

  def self.find_by_slug(slug)
    find_by(slug: slug)
  end

  private

  def update_slug
    self.slug = title.join('-')
  end

  def send_welcome_email
    WelcomeMailer.welcome(self).deliver_now
  end
end
```

Models are typically responsible for:

- setting up relationships
- validating data
- providing access to data via scopes and methods
- Performing actions around persistence of data.

At the highest level, models describe domain concepts and manages their persistence.

Chapter 48: Shallow Routing

Section 48.1: Use of shallow

One way to avoid deep nesting (as recommended above) is to generate the collection actions scoped under the parent, so as to get a sense of the hierarchy, but to not nest the member actions. In other words, to only build routes with the minimal amount of information to uniquely identify the resource, like this:

```ruby
resources :articles, shallow: true do
  resources :comments
  resources :quotes
  resources :drafts
end
```

The shallow method of the DSL creates a scope inside of which every nesting is shallow. This generates the same routes as the previous example:

```ruby
shallow do
  resources :articles do
    resources :comments
    resources :quotes
    resources :drafts
  end
end
```

There exist two options for scope to customize shallow routes. :shallow_path prefixes member paths with the specified parameter:

```ruby
scope shallow_path: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```

Use Rake Command for get generated routes as define below:

```
rake routes
```

Chapter 49: Model states: AASM

Section 49.1: Basic state with AASM

Usually you'll end up creating models which will contain a state, and that state will be changing during the lifespan of the object.

AASM is a finite state machine enabler library that can help you out with dealing with having an easy passing through the process design of your objects.

Having something like this in your model goes pretty aligned with the Fat Model, Skinny Controller idea, one of Rails best practices. The model is the sole responsible of managing its state, its changes and of generating the events triggered by those changes.

To install, in Gemfile

```ruby
gem 'aasm'
```

Consider an App where the user Quotes a product for a price.

```ruby
class Quote

  include AASM

  aasm do
    state :requested, initial: true  # User sees a product and requests a quote
    state :priced                    # Seller sets the price
    state :payed                     # Buyer pays the price
    state :canceled                  # The buyer is not willing to pay the price
    state :completed                 # The product has been delivered.

    event :price do
        transitions from: requested, to: :priced
    end

    event :pay do
        transitions from: :priced, to: :payed, success: :set_payment_date
    end

    event :complete do
        transitions from: :payed, to: :completed, guard: product_delivered?
    end

    event :cancel do
        transitions from: [:requested, :priced], to: :canceled
        transitions from: :payed, to: canceled, success: :reverse_charges
    end

  end

  private

  def set_payment_date
    update payed_at: Time.zone.now
  end
end
```

The Quote class' states can go however it's best for your process.

You can think of the states as being past, like in the previous example or algo in other tense, for example: pricing, paying, delivering, etc. The naming of the states depends on you. From a personal point a view, past states work better because your end state will surely be a past action and links up better with the event names, which will be explained later.

NOTE: Be careful what names you use, you have to worry about not using Ruby or Ruby on Rails reserved keywords, like `valid`, `end`, `being`, etc.

Having defined the states and transitions we can now access some methods created by AASM.

For example:

```
Quote.priced  # Shows all Quotes with priced events
quote.priced? # Indicates if that specific quote has been priced
quote.price!  # Triggers the event the would transition from requested to priced.
```

As you can see the event has transitions, this transitions determine the way the state will change upon the event call. If the event is invalid due to the current state an Error will be raised.

The events and transitions also have some other callbacks, for example

```
guard: product_delivered?
```

Will call the `product_delivered?` method which will return a boolean. If it turns out false, the transition will not be applied and if the no other transitions are available, the state won't change.

```
success: :reverse_charges
```

If that translation successfully happens the `:reverse_charges` method will be invoked.

There are several other methods in AASM with more callbacks in the process but this will help you creating your first models with finite states.

Chapter 50: Rails 5 API Authetication

Section 50.1: Authentication with Rails authenticate_with_http_token

```ruby
authenticate_with_http_token do |token, options|
  @user = User.find_by(auth_token: token)
end
```

You can test this endpoint with `curl` by making a request like

```
curl -IH "Authorization: Token token=my-token" http://localhost:3000
```

Chapter 51: Testing Rails Applications

Section 51.1: Unit Test

Unit tests test parts of the application in isolation. usually a unit under test is a class or module.

```
let(:gift) { create :gift }

describe '#find' do
  subject { described_class.find(user, Time.zone.now.to_date) }
  it { is_expected.to eq gift }
end
```

source

This kind if test is as direct and specific as possible.

Section 51.2: Request Test

Request tests are end to end tests that imitate the behavior of a user.

```
it 'allows the user to set their preferences' do
  check 'Ruby'
  click_on 'Save and Continue'
  expect(user.languages).to eq ['Ruby']
end
```

source

This kind of test focuses on user flows and runs through all layers of the system sometimes even rendering javascript.

Chapter 52: Active Jobs

Section 52.1: Introduction

Available since Rails 4.2, Active Job is a framework for declaring jobs and making them run on a variety of queuing backends. Recurring or punctual tasks that are not blocking and can be run in parallel are good use cases for Active Jobs.

Section 52.2: Sample Job

```ruby
class UserUnsubscribeJob < ApplicationJob
  queue_as :default

  def perform(user)
    # this will happen later
    user.unsubscribe
  end
end
```

Section 52.3: Creating an Active Job via the generator

```
$ rails g job user_unsubscribe
```

Chapter 53: Rails frameworks over the years

When you're new to Rails and working on legacy Rails applications, it can be confusing to understand which framework was introduced when. This topic is designed to be the *definitive* list of all frameworks across Rails versions.

Section 53.1: How to find what frameworks are available in the current version of Rails?

Use the

> config.frameworks

option to get an array of **Symbol**s that represent each framework.

Section 53.2: Rails versions in Rails 1.x

- ActionMailer
- ActionPack
- ActionWebService
- ActiveRecord
- ActiveSupport
- Railties

Section 53.3: Rails frameworks in Rails 2.x

- ActionMailer
- ActionPack
- ActiveRecord
- ActiveResource (*ActiveWebService was replaced by ActiveResource, and with that, Rails moved from SOAP to REST by default*)
- ActiveSupport
- Railties

Section 53.4: Rails frameworks in Rails 3.x

- ActionMailer
- ActionPack
- ActiveModel
- ActiveRecord
- ActiveResource
- ActiveSupport
- Railties

Chapter 54: Nested form in Ruby on Rails

Section 54.1: How to setup a nested form in Ruby on Rails

The first to thing to have: a model that contains a `has_many` relation with another model.

```ruby
class Project < ApplicationRecord
  has_many :todos
end

class Todo < ApplicationRecord
  belongs_to :project
end
```

In `ProjectsController`:

```ruby
class ProjectsController < ApplicationController
  def new
    @project = Project.new
  end
end
```

In a nested form, you can create child objects with a parent object at the same time.

```erb
<%= nested_form_for @project do |f| %>
  <%= f.label :name %>
  <%= f.text_field :name %>

  <% # Now comes the part for `Todo` object %>
  <%= f.fields_for :todo do |todo_field| %>
    <%= todo_field.label :name %>
    <%= todo_field.text_field :name %>
  <% end %>
<% end %>
```

As we initialized `@project` with `Project.new` to have something for creating a new `Project` object, same way for creating a `Todo` object, we have to have something like this, and there are multiple ways to do so:

1. In `Projectscontroller`, in new method, you can write: `@todo = @project.todos.build` or `@todo = @project.todos.new` to instantiate a new `Todo` object.

2. You can also do this in view: `<%= f.fields_for :todos, @project.todos.build %>`

For strong params, you can include them in the following way:

```ruby
def project_params
  params.require(:project).permit(:name, todo_attributes: [:name])
end
```

Since, the `Todo` objects will be created through the creation of a `Project` object, so you have to specify this thing in `Project` model by adding the following line:

```ruby
accepts_nested_attributes_for :todos
```

Chapter 55: Factory Girl

Section 55.1: Defining Factories

If you have a ActiveRecord User class with name and email attributes, you could create a factory for it by making the FactoryGirl guess it:

```
FactoryGirl.define do
  factory :user do # it will guess the User class
    name     "John"
    email    "john@example.com"
  end
end
```

Or you can make it explicit and even change its name:

```
FactoryGirl.define do
  factory :user_jack, class: User do
    name     "Jack"
    email    "jack@example.com"
  end
end
```

Then in your spec you can use the FactoryGirl's methods with these, like this:

```
# To create a non saved instance of the User class filled with John's data
build(:user)
# and to create a non saved instance of the User class filled with Jack's data
build(:user_jack)
```

The most common methods are:

```
# Build returns a non saved instance
user = build(:user)

# Create returns a saved instance
user = create(:user)

# Attributes_for returns a hash of the attributes used to build an instance
attrs = attributes_for(:user)
```

Chapter 56: Import whole CSV files from specific folder

In this example, lets say we have many product CSV files in a folder. Each CSV file need to upload our database from our console write a command. Run the following command in a new or existing project to create this model.

Section 56.1: Uploads CSV from console command

Terminal Commands:

```
rails g model Product name:string quantity:integer price:decimal{12,2}
rake db:migrate
```

Lates create controller.

Terminal Commands:

```
rails g controller Products
```

Controller Code:

```ruby
class HistoriesController < ApplicationController
    def create
        file = Dir.glob("#{Rails.root}/public/products/**/*.csv") #=> This folder directory where
read the CSV files
        file.each do |file|
            Product.import(file)
        end
    end
end
```

Model:

```ruby
class Product< ApplicationRecord
  def self.import(file)
      CSV.foreach(file.path, headers: true) do |row|
          Product.create! row.to_hash
      end
  end
end
```

routes.rb

```ruby
resources :products
```

app/config/application.rb

```ruby
require 'csv'
```

Now open your development `console` & run

```ruby
=> ProductsController.new.create #=> Uploads your whole CSV files from your folder directory
```

Chapter 57: Tools for Ruby on Rails code optimization and cleanup

Keeping your code clean and organized while developing a large Rails application can be quite a challenge, even for an experienced developer. Fortunately, there is a whole category of gems that make this job much easier.

Section 57.1: If you want to keep your code maintainable, secure and optimized, look at some gems for code optimization and cleanup :

Bullet

This one particularly blew my mind. The bullet gem helps you kill all the N+1 queries, as well as unnecessarily eager loaded relations. Once you install it and start visiting various routes in development, alert boxes with warnings indicating database queries that need to be optimized will pop out. It works right out of the box and is extremely helpful for optimizing your application.

Rails Best Practices

Static code analyzer for finding Rails specific code smells. It offers a variety of suggestions; use scope access, restrict auto-generated routes, add database indexes, etc. Nevertheless, it contains lots of nice suggestions that will give you a better perspective on how to re-factor your code and learn some best practices.

Rubocop

A Ruby static code analyzer which you can use to check if your code complies with the Ruby community code guidelines. The gem reports style violations through the command line, with lots of useful code refactoring goodies such as useless variable assignment, redundant use of Object#to_s in interpolation or even unused method argument.

A good thing is that it's highly configurable, since the analyzer can be quite irritating if you're not following the Ruby style guide 100% (i.e. you have lots of trailing whitespaces or you double quote your strings even when not interpolating, etc.).

It's divided into 4 sub-analyzers (called cops): Style, Lint, Metrics and Rails.

Chapter 58: ActiveJob

Active Job is a framework for declaring jobs and making them run on a variety of queuing backends. These jobs can be everything from regularly scheduled clean-ups, to billing charges, to mailings. Anything that can be chopped up into small units of work and run in parallel, really.

Section 58.1: Create the Job

```ruby
class GuestsCleanupJob < ApplicationJob
  queue_as :default

  def perform(*guests)
    # Do something later
  end
end
```

Section 58.2: Enqueue the Job

```ruby
# Enqueue a job to be performed as soon as the queuing system is free.
GuestsCleanupJob.perform_later guest
```

Chapter 59: Active Model Serializers

ActiveModelSerializers, or AMS for short, bring 'convention over configuration' to your JSON generation. ActiveModelSerializers work through two components: serializers and adapters. Serializers describe which attributes and relationships should be serialized. Adapters describe how attributes and relationships should be serialized.

Section 59.1: Using a serializer

```
class SomeSerializer < ActiveModel::Serializer
  attribute :title, key: :name
  attributes :body
end
```

Chapter 60: Rails Engine - Modular Rails

Quick overview of Rails engines

Engines are small Rails applications that can be used to add functionalities to the application hosting them. The class defining a Ruby on Rails application is `Rails::Application` which actually inherits a lot of its behavior from `Rails::Engine`, the class defining an engine. We can say that a regular Rails application is simply an engine with more features.

Section 60.1: Create a modular app

Getting started

First, let's generate a new Ruby on Rails application:

```
rails new ModularTodo
```

The next step is to generate an engine!

```
cd ModularTodo && rails plugin new todo --mountable
```

We will also create an 'engines' folder to store the engines (even if we just have one!).

```
mkdir engines && mv todo ./engines
```

Engines, just like gems, come with a gemspec file. Let's put some real values to avoid warnings.

```
#ModularTodo/engines/todo/todo.gemspec
$:.push File.expand_path("../lib", __FILE__)

#Maintain your gem's version:
require "todo/version"

#Describe your gem and declare its dependencies:
Gem::Specification.new do |s|
  s.name        = "todo"
  s.version     = Todo::VERSION
  s.authors     = ["Thibault Denizet"]
  s.email       = ["bo@samurails.com"]
  s.homepage    = "//samurails.com"
  s.summary     = "Todo Module"
  s.description = "Todo Module for Modular Rails article"
  s.license     = "MIT"

  #Moar stuff
  #...
end
```

Now we need to add the Todo engine to the parent application Gemfile.

```
#ModularTodo/Gemfile
#Other gems
gem 'todo', path: 'engines/todo'
```

Let's run bundle `install`. You should see the following in the list of gems:

```
Using todo 0.0.1 from source at engines/todo
```

Great, our Todo engine is loaded correctly! Before we start coding, we have one last thing to do: mount the Todo engine. We can do that in the routes.rb file in the parent app.

```ruby
Rails.application.routes.draw do
  mount Todo::Engine => "/", as: 'todo'
end
```

We are mounting it at `/` but we could also make it accessible at `/todo`. Since we have only one module, `/` is fine.

Now you can fire up your server and check it in your browser. You should see the default Rails view because we didn't define any controllers/views yet. Let's do that now!

Building the Todo list

We are going to scaffold a model named `Task` inside the Todo module but to correctly migrate the database from the parent application, we need to add a small initializer to the `engine.rb` file.

```ruby
#ModularTodo/engines/todo/lib/todo/engine.rb
module Todo
  class Engine < ::Rails::Engine
    isolate_namespace Todo

    initializer :append_migrations do |app|
      unless app.root.to_s.match(root.to_s)
        config.paths["db/migrate"].expanded.each do |p|
          app.config.paths["db/migrate"] << p
        end
      end
    end

  end
end
```

That's it, now when we run migrations from the parent application, the migrations in the Todo engine will be loaded too.

Let's create the `Task` model. The `scaffold` command needs to be run from the engine folder.

```
cd engines/todo && rails g scaffold Task title:string content:text
```

Run the migrations from the parent folder:

```
rake db:migrate
```

Now, we just need to define the root route inside the Todo engine:

```ruby
#ModularTodo/engines/todo/config/routes.rb
Todo::Engine.routes.draw do
  resources :tasks
  root 'tasks#index'
end
```

You can play with it, create tasks, delete them... Oh wait, the delete is not working! Why?! Well, it seems JQuery is not loaded, so let's add it to the `application.js` file inside the engine!

```
// ModularTodo/engines/todo/app/assets/javascripts/todo/application.js
//= require jquery
//= require jquery_ujs
//= require_tree .
```

Yay, now we can destroy tasks!

Chapter 61: Single Table Inheritance

Single Table Inheritance (STI) is a design pattern which is based on the idea of saving the data of multiple models which are all inheriting from the same Base model, into a single table in the database.

Section 61.1: Basic example

First we need a table to hold our data

```ruby
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :password
      t.string :type # <- This makes it an STI

      t.timestamps
    end
  end
end
```

Then lets create some models

```ruby
class User < ActiveRecord::Base
  validates_presence_of :password
  # This is a parent class. All shared logic goes here
end

class Admin < User
  # Admins must have more secure passwords than regular users
  # We can add it here
  validates :custom_password_validation
end

class Guest < User
  # Lets say that we have a guest type login.
  # It has a static password that cannot be changed
  validates_inclusion_of :password, in: ['guest_password']
end
```

When you do a `Guest.create(name: 'Bob')` ActiveRecord will translate this to create an entry in the Users table with `type: 'Guest'`.

When you retrieve the record bob = `User.where(name: 'Bob').first` the object returned will be an instance of `Guest`, which can be forcibly treated as a User with bob`.becomes(User)`

becomes is most useful when dealing with shared partials or routes/controllers of the superclass instead of the subclass.

Section 61.2: Custom inheritance column

By default STI model class name is stored in a column named `type`. But its name can be changed by overriding `inheritance_column` value in a base class. E.g.:

```ruby
class User < ActiveRecord::Base
  self.inheritance_column = :entity_type # can be string as well
```

```
  end

class Admin < User; end
```

Migration in this case will look as follows:

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :password
      t.string :entity_type

      t.timestamps
    end
  end
end
```

When you do `Admin.create`, this record will be saved in the users table with `entity_type` = `"Admin"`

Section 61.3: Rails model with type column and without STI

Having `type` column in a Rails model without invoking STI can be achieved by assigning `:_type_disabled` to `inheritance_column`:

```
class User < ActiveRecord::Base
  self.inheritance_column = :_type_disabled
end
```

Chapter 62: ActiveRecord Transactions

ActiveRecord Transactions are protective blocks where sequence of active record queries are only permanent if they can all succeed as one atomic action.

Section 62.1: Getting Started with Active Record Transactions

Active Record Transactions can be applied to Model classes as well as Model instances, the objects within the transaction block need not all be instances of same class. This is because transactions are per-database connection, not per-model. For example:

```ruby
User.transaction do
  account.save!
  profile.save!
  print "All saves success, returning 1"
  return 1
end
rescue_from ActiveRecord::RecordInvalid do |exception|
  print "Exception thrown, transaction rolledback"
  render_error "failure", exception.record.errors.full_messages.to_sentence
end
```

Using save with a bang ensures that transaction will be automatically rolled back when the exception is thrown and after the rollback, control goes to the rescue block for the exception. **Make sure you rescue the exceptions thrown from the save! in Transaction Block.**

If you don't want to use save!, you can manually raise **raise ActiveRecord::Rollback** when the save fails. You need not handle this exception. It will then rollback the transaction and take the control to the next statement after transaction block.

```ruby
  User.transaction do
    if account.save && profile.save
      print "All saves success, returning 1"
      return 1
    else
      raise ActiveRecord::Rollback
    end
  end
  print "Transaction Rolled Back"
```

Chapter 63: Turbolinks

Turbolinks is a javascript library that makes navigating your web application faster. When you follow a link, Turbolinks automatically fetches the page, swaps in its <body>, and merges its <head>, all without incurring the cost of a full page load.

Section 63.1: Binding to turbolink's concept of a page load

With turbolinks, the traditional approach to using:

```
$(document).ready(function() {
  // awesome code
});
```

won't work. While using turbolinks, the $(document).ready() event will only fire once: on the initial page load. From that point on, whenever a user clicks a link on your website, turbolinks will intercept the link click event and make an ajax request to replace the <body> tag and to merge the <head> tags. The whole process triggers the notion of a "visit" in turbolinks land. Therefore, instead of using the traditional document.ready() syntax above, you'll have to bind to turbolink's visit event like so:

```
// pure js
document.addEventListener("turbolinks:load", function() {
  // awesome code
});

// jQuery
$(document).on('turbolinks:load', function() {
  // your code
});
```

Section 63.2: Disable turbolinks on specific links

It is very easy to disable turbolinks on specific links. According to the official turbolinks documentation:

> Turbolinks can be disabled on a per-link basis by annotating a link or any of its ancestors with data-turbolinks="false".

Examples:
```
// disables turbolinks for this one link
<a href="/" data-turbolinks="false">Disabled</a>

// disables turbolinks for all links nested within the div tag
<div data-turbolinks="false">
  <a href="/">I'm disabled</a>
  <a href="/">I'm also disabled</a>
</div>

// re-enable specific link when ancestor has disabled turbolinks
<div data-turbolinks="false">
  <a href="/">I'm disabled</a>
  <a href="/" data-turbolinks="true">I'm re-enabled</a>
</div>
```

Section 63.3: Understanding Application Visits

Application visits are initiated by clicking a Turbolinks-enabled link, or programmatically by calling

```
Turbolinks.visit(location)
```

By default, the visit function uses the 'advance' action. More understandably, the default behavior for the visit function is to advance to the page indicated by the "location" parameter. Whenever a page is visited, turbolinks pushes a new entry onto the browser's history using `history.pushState`. The history is important because turbolinks will try to use the history to load pages from cache whenever possible. This allows for extremely fast page rendering for frequently visited pages.

However, if you want to visit a location without pushing any history onto the stack, you can use the 'replace' action on the visit function like so:

```
// using links
<a href="/edit" data-turbolinks-action="replace">Edit</a>

// programmatically
Turbolinks.visit("/edit", { action: "replace" })
```

This will replace the top of the history stack with the new page so that the total number of items on the stack remains unchanged.

There is also a "restore" action that aids in restoration vists, the visits that occur as a result of the user clicking the forward button or back button on their browser. Turbolinks handles these types of events internally and recommends that users don't manually tamper with the default behavior.

Section 63.4: Cancelling visits before they begin

Turbolinks provides an event listener that can be used to stop visits from occurring. Listen to the `turbolinks:before-visit` event to be notified when a visit is about to commence.

In the event handler, you can use:

```
// pure javascript
event.data.url
```

or

```
// jQuery
$event.originalEvent.data.url
```

to retrieve the location of the visit. The visit can then be cancelled by calling:

```
event.preventDefault()
```

NOTE:

According to the official turbolinks docs:

> Restoration visits cannot be canceled and do not fire turbolinks:before-visit.

Section 63.5: Persisting elements across page loads

Consider the following situation: Imagine that you are the developer of a social media website that allows users to be friends with other users and that employs turbolinks to make page loading faster. In the top right of every page on the site, there is a number indicating the total number of friends that a user currently has. Imagine you are using your site and that you have 3 friends. Whenever a new friend is added, you have some javascript that runs which updates the friend counter. Imagine that you just added a new friend and that your javascript ran properly and updated the friend count in the top right of the page to now render 4. Now, imagine that you click the browser's back button. When the page loads, you notice that the friend counter says 3 even though you have four friends.

This is a relatively common problem and one that turbolinks has provided a solution for. The reason the problem occurs is because turbolinks automatically loads pages from the cache when a user clicks the back button. The cached page won't always be updated with the database.

To solve this issue, imagine that you render the friend count inside a <div> tag with an id of "friend-count":

```
<div id="friend-count" data-turbolinks-permanent>3 friends</div>
```

By adding the `data-turbolinks-permanent` attribute, you're telling turbolinks to persist certain elements across page loads. The [official docs say]:

> Designate permanent elements by giving them an HTML id and annotating them with data-turbolinks-permanent. Before each render, Turbolinks matches all permanent elements by id and transfers them from the original page to the new page, preserving their data and event listeners.

Chapter 64: Friendly ID

FriendlyId is the "Swiss Army bulldozer" of slugging and permalink plugins for Active Record. It lets you create pretty URLs and work with human-friendly strings as if they were numeric ids. With FriendlyId, it's easy to make your application use URLs like:

http://example.com/states/washington

Section 64.1: Rails Quickstart

```
rails new my_app
cd my_app
```

Gemfile

```
gem 'friendly_id', '~> 5.1.0' # Note: You MUST use 5.0.0 or greater for Rails 4.0+
rails generate friendly_id
rails generate scaffold user name:string slug:string:uniq
rake db:migrate
```

edit app/models/user.rb

```
class User < ApplicationRecord
  extend FriendlyId
  friendly_id :name, use: :slugged
end

User.create! name: "Joe Schmoe"

# Change User.find to User.friendly.find in your controller
User.friendly.find(params[:id])
```

```
rails server
GET http://localhost:3000/users/joe-schmoe
```

```
# If you're adding FriendlyId to an existing app and need
# to generate slugs for existing users, do this from the
# console, runner, or add a Rake task:
User.find_each(&:save)
```

Finders are no longer overridden by default. **If** you want to **do** friendly finds, you must **do** Model.friendly.find rather than Model.find. You can however restore FriendlyId 4-style finders by using the :finders addon

```
friendly_id :foo, use: :slugged # you must do MyClass.friendly.find('bar')
#or...
friendly_id :foo, use: [:slugged, :finders] # you can now do MyClass.find('bar')
```

A new "candidates" functionality which makes it easy to set up a list of alternate slugs that can be used to uniquely distinguish records, rather than appending a sequence. For example:

```
class Restaurant < ActiveRecord::Base
  extend FriendlyId
  friendly_id :slug_candidates, use: :slugged

  # Try building a slug based on the following fields in
  # increasing order of specificity.
  def slug_candidates
```

```ruby
    [
      :name,
      [:name, :city],
      [:name, :street, :city],
      [:name, :street_number, :street, :city]
    ]
  end
end
```

Set slug limit length using friendly_id gem?

```ruby
def normalize_friendly_id(string)
    super[0..40]
end
```

Chapter 65: Securely storing authentication keys

Many third-party APIs require a key, allowing them to prevent abuse. If they issue you a key, it's very important that you not commit the key into a public repository, as this will allow others to steal your key.

Section 65.1: Storing authentication keys with Figaro

Add `gem 'figaro'` to your Gemfile and run `bundle install`. Then run `bundle exec figaro install`; this will create config/application.yml and add it to your .gitignore file, preventing it from being added to version control.

You can store your keys in application.yml in this format:

```
SECRET_NAME: secret_value
```

where SECRET_NAME and secret_value are the name and value of your API key.

You also need to name these secrets in config/secrets.yml. You can have different secrets in each environment. The file should look like this:

```
development:
  secret_name: <%= ENV["SECRET_NAME"] %>
test:
  secret_name: <%= ENV["SECRET_NAME"] %>
production:
  secret_name: <%= ENV["SECRET_NAME"] %>
```

How you use these keys varies, but say for example `some_component` in the development environment needs access to `secret_name`. In config/environments/development.rb, you'd put:

```
Rails.application.configure do
  config.some_component.configuration_hash = {
    :secret => Rails.application.secrets.secret_name
  }
end
```

Finally, let's say you want to spin up a production environment on Heroku. This command will upload the values in config/environments/production.rb to Heroku:

```
$ figaro heroku:set -e production
```

Chapter 66: Authenticate Api using Devise

Devise is authentication solution for Rails. Before going any further i would like to add quick note on API. So API does not handle sessions (is stateless) which means one that provide response after you request, and then requires no further attention, which means no previous or future state is required for the system to work hence whenever we requesting to the server need to pass authentication details with all API and should tell Devise not to store authentication details.

Section 66.1: Getting Started

So first we will create rails project and setup device

create a rails application

```
rails new devise_example
```

now add devise to gem list

> you can find a file named 'Gemfile' at the root of rails project

Then run `bundle install`

Next, you need to run the generator:

```
rails generate devise:install
```

Now on console you can find few instructions just follow it.

Generate devise model

```
rails generate devise MODEL
```

Then run `rake db:migrate`

For more details go to: [Devise Gem](#)

Authentication Token

Authentication token is used to authenticate a user with a unique token, So Before we proceed with the logic first we need to add `auth_token` field to a Devise model

Hence,

```
rails g migration add_authentication_token_to_users

class AddAuthenticationTokenToUsers < ActiveRecord::Migration
  def change
    add_column :users, :auth_token, :string, default: ""
    add_index :users, :auth_token, unique: true
  end
end
```

Then run `rake db:migrate`

Now we are all set to do authentication using `auth_token`

In app`/`controllers`/`application_controllers`.rb`

First this line to it

```
respond_to :html, :json
```

this will help rails application respond with both html and json

Then

```
protect_from_forgery with: :null
```

will change this `:null` as we are not dealing with sessions.

now we will add authentication method in application_controller

So, by default Devise uses email as unique field we can also use custom fields, for this case we will be authenticating using user_email and auth_token.

```
before_filter do
  user_email = params[:user_email].presence
  user       = user_email && User.find_by_email(user_email)

  if user && Devise.secure_compare(user.authentication_token, params[:auth_token])
    sign_in user, store: false
  end
end
```

> Note: Above code is purely based on your logic i am just trying to explain the working example

On line 6 in the above code you can see that i have set `store: false` which will prevent from creating a session on each requests hence we achieved stateless re

Chapter 67: Integrating React.js with Rails Using Hyperloop

This topic covers integrating React.js with Rails using the Hyperloop gem

Other approaches not covered here are using the react-rails or react_on_rails gems.

Section 67.1: Adding a simple react component (written in ruby) to your Rails app

1. Add the hyperloop gem to your rails (4.0 - 5.1) Gemfile
2. `bundle install`
3. Add the hyperloop manifest to the application.js file: // app/assets/javascripts/application.js ... //= hyperloop-loader
4. Create your react components, and place them in the hyperloop`/`components directory # app/hyperloop/components/hello_world.rb class HelloWorld < Hyperloop::Component after_mount do every(1.second) { mutate.current_time(Time.now) } end render do "Hello World! The time is now: #{state.current_time}" end end
5. Components act just like views. They are "mounted" using the `render_component` method in a controller: # somewhere in a controller: ... def hello_world render_component # renders HelloWorld based on method name end

Section 67.2: Callbacks

```ruby
# all react callbacks are supported using active-record-like syntax

class SomeCallBacks < Hyperloop::Component
  before_mount do
    # initialize stuff - replaces normal class initialize method
  end
  after_mount do
    # any access to actual generated dom node, or window behaviors goes here
  end
  before_unmount do
    # any cleanups (i.e. cancel intervals etc)
  end

  # you can also specify a method the usual way:
  before_mount :do_some_more_initialization
end
```

Section 67.3: Declaring component parameters (props)

```ruby
class Hello < Hyperloop::Component
  # params (= react props) are declared using the param macro
  param :guest
  render do
    "Hello there #{params.guest}"
  end
end

# to "mount" Hello with guest = "Matz" say
  Hello(guest: 'Matz')
```

```
# params can be given a default value:
  param guest: 'friend' # or
  param :guest, default: 'friend'
```

Section 67.4: HTML Tags

```
# HTML tags are built in and are UPCASE
class HTMLExample < Hyperloop::Component
  render do
    DIV do
      SPAN { "Hello There" }
      SPAN { "Welcome to the Machine!" }
    end
  end
end
```

Section 67.5: Event Handlers

```
# Event handlers are attached using the 'on' method
class ClickMe < Hyperloop::Component
  render do
    DIV do
      SPAN { "Hello There" }
      A { "Click Me" }.on(:click) { alert('you did it!' }
    end
  end
end
```

Section 67.6: States

```
# States are read using the 'state' method, and updated using 'mutate'
# when states change they cause re-render of all dependent dom elements

class StateExample < Hyperloop::Component
  state count: 0  # by default states are initialized to nil
  render do
    DIV do
      SPAN { "Hello There" }
      A { "Click Me" }.on(:click) { mutate.count(state.count + 1) }
      DIV do
        "You have clicked me #{state.count} #{'time'.pluralize(state.count)}"
      end unless state.count == 0
    end
  end
end
```

Note that states can be shared between components using Hyperloop::Stores

Chapter 68: Change a default Rails application enviornment

This will discuss how to change the environment so when someone types `rails s` they boot in not development but in the environment they want.

Section 68.1: Running on a local machine

Normally when rails environment is run by typing. This just runs the default environment which is usually `development`

```
rails s
```

The specific environment can be selected by using the flag `-e` for example:

```
rails s -e test
```

Which will run the test environment.

The default environment can be changed in terminal by editing the `~/.bashrc` file, and adding the following line:

```
export RAILS_ENV=production in your
```

Section 68.2: Running on a server

If running on a remote server that is using Passenger change apache.conf to to the environment you want to use. For example this case you see `RailsEnv production`.

```
<VirtualHost *:80>
  ServerName application_name.rails.local
  DocumentRoot "/Users/rails/application_name/public"
  RailsEnv production ## This is the default
</VirtualHost>
```

Chapter 69: Rails -Engines

Parameters	Purpose
--mountable	option tells the generator that you want to create a "mountable" and namespace-isolated engine
--full	option tells the generator that you want to create an engine, including a skeleton structure

Engines can be considered miniature applications that provide functionality to their host applications. A Rails application is actually just a "supercharged" engine, with the Rails::Application class inheriting a lot of its behavior from Rails::Engine.

Engines are the reusable rails applications/plugins. It works like a Gem. Famous engines are Device, Spree gems which can be integrated with rails applications easily.

Section 69.1: Famous examples are

Generating simple blog engine

```
rails plugin new [engine name] --mountable
```

Famous engines examples are

Device (authentication gem for rails)

Spree (Ecommerce)

Chapter 70: Adding an Amazon RDS to your rails application

Steps to create an AWS RDS instance and configure your database.yml file by installing the required connectors.

Section 70.1: Consider we are connecting MYSQL RDS with your rails application

Steps to create MYSQL database

1. Login to amazon account and select RDS service
2. Select `Launch DB Instance` from the instance tab
3. By defaul MYSQL Community Edition will be selected, hence click the **SELECT** button
4. Select the database purpose, say `production` and click **next** step
5. Provide the `mysql version, storage size, DB Instance Identifier, Master Username` **and** `Password` and click **next** step
6. Enter `Database Name` and click `Launch DB Instance`
7. Please wait until all the instance gets created. Once the instance gets created you will find an Endpoint, copy this entry point (which is referred as hostname)

Installing connectors

Add the MySQL database adapter to your project's gemfile,

```
gem 'mysql2'
```

Install your added gems,

```
bundle install
```

Some other database adapters are,

- gem `'pg'` for PostgreSQL
- gem `'activerecord-oracle_enhanced-adapter'` for Oracle
- gem `'sql_server'` for SQL Server

Configure your project's database.yml file Open your config/database.yml file

```
production:
  adapter: mysql2
  encoding: utf8
  database: <%= RDS_DB_NAME %>   # Which you have entered you creating database
  username: <%= RDS_USERNAME %> # db master username
  password: <%= RDS_PASSWORD %> # db master password
  host: <%= RDS_HOSTNAME %>     # db instance entrypoint
  port: <%= RDS_PORT %>         # db post. For MYSQL 3306
```

Chapter 71: Payment feature in rails

This document pretend to introduce you, with a complete example, how you can implement different payment methods with Ruby on Rails.

In the example, we will cover Stripe and Braintree two very well-known payment platforms.

Section 71.1: How to integrate with Stripe

Add Stripe gem to our `Gemfile`

```
gem 'stripe'
```

Add `initializers/stripe.rb` file. This file contains the necessary keys for connecting with your stripe account.

```
require 'require_all'

Rails.configuration.stripe = {
    :publishable_key => ENV['STRIPE_PUBLISHABLE_KEY'],
    :secret_key      => ENV['STRIPE_SECRET_KEY']
}

Stripe.api_key = Rails.configuration.stripe[:secret_key]
```

How to create a new customer to Stripe

```
Stripe::Customer.create({email: email, source: payment_token})
```

This code creates a new customer on Stripe with given email address and source.

`payment_token` is the token given from the client-side that contains a payment method like a credit card or bank account. More info: Stripe.js client-side

How to retrieve a plan from Stripe

```
Stripe::Plan.retrieve(stripe_plan_id)
```

This code retrieves a plan from Stripe by its id.

How to create a subscription

When we have a customer and a plan we can create a new subscription on Stripe.

```
Stripe::Subscription.create(customer: customer.id, plan: plan.id)
```

It will create a new subscription and will charge our User. It's important to know what really happens on Stripe when we subscribe a user to a plan, you will find more info here: Stripe Subscription lifecycle.

How to charge a user with a single payment

Sometimes we want to charge our users just a single time, for do that we will do the next.

```
Stripe::Charge.create(amount:   amount, customer: customer, currency: currency)
```

In that case, we are charging our user one time for given amount.

Common errors:

- The amount must be sent in integer form, that means, 2000 will be 20 units of currency. [Check this example](#)

- You cannot charge a user in two currencies. If the user was charged in EUR at any moment in the past you cannot charge the user in USD.

- You cannot charge user without source (payment method).

Chapter 72: Rails on docker

This tutorial will start with Docker installed and with a Rails app

Section 72.1: Docker and docker-compose

First of all, we will need to create our `Dockerfile`. A good example can be found on this [blog](#) by Nick Janetakis.

This code contains the script that will be executed on our docker machine at the moment of start.For this reason, we are installing all the required libraries and ends with the start of Puma (RoR dev server)

```dockerfile
# Use the barebones version of Ruby 2.3.
FROM ruby:2.3.0-slim

# Optionally set a maintainer name to let people know who made this image.
MAINTAINER Nick Janetakis <nick.janetakis@gmail.com>

# Install dependencies:
# - build-essential: To ensure certain gems can be compiled
# - nodejs: Compile assets
# - libpq-dev: Communicate with postgres through the postgres gem
RUN apt-get update && apt-get install -qq -y --no-install-recommends \
      build-essential nodejs libpq-dev git


# Set an environment variable to store where the app is installed to inside
# of the Docker image. The name matches the project name out of convention only.
ENV INSTALL_PATH /mh-backend
RUN mkdir -p $INSTALL_PATH

# This sets the context of where commands will be running in and is documented
# on Docker's website extensively.
WORKDIR $INSTALL_PATH

# We want binstubs to be available so we can directly call sidekiq and
# potentially other binaries as command overrides without depending on
# bundle exec.
COPY Gemfile* $INSTALL_PATH/

ENV BUNDLE_GEMFILE $INSTALL_PATH/Gemfile
ENV BUNDLE_JOBS 2
ENV BUNDLE_PATH /gembox

RUN bundle install

# Copy in the application code from your work station at the current directory
# over to the working directory.
COPY . .

# Ensure the static assets are exposed to a volume so that nginx can read
# in these values later.
VOLUME ["$INSTALL_PATH/public"]

ENV RAILS_LOG_TO_STDOUT true


# The default command that gets run will be to start the Puma server.
CMD bundle exec puma -C config/puma.rb
```

Also, we will use docker-compose, for that, we will create `docker-compose.yml`. The explanation of this file will be more a docker-compose tutorial than an integration with Rails and I will not cover here.

```
version: '2'

services:
  backend:
    links:
      - #whatever you need to link like db
    build: .
    command: ./scripts/start.sh
    ports:
      - '3000:3000'
    volumes:
      - .:/backend
    volumes_from:
      - gembox
    env_file:
      - .dev-docker.env
    stdin_open: true
    tty: true
```

Just with these two files you will have enough to run `docker-compose up` and wake up your docker

Appendix A: Reserved Words

You should be careful using these words for variable, model name, method name or etc.

Section A.1: Reserved Word List

- ADDITIONAL_LOAD_PATHS
- ARGF
- ARGV
- ActionController
- ActionView
- ActiveRecord
- ArgumentError
- Array
- BasicSocket
- Benchmark
- Bignum
- Binding
- CGI
- CGIMethods
- CROSS_COMPILING
- Class
- ClassInheritableAttributes
- Comparable
- ConditionVariable
- Config
- Continuation
- DRb
- DRbIdConv
- DRbObject
- DRbUndumped
- Data
- Date
- DateTime
- Delegater
- Delegator
- Digest
- Dir
- ENV
- EOFError
- ERB
- Enumerable
- Errno
- Exception
- FALSE
- FalseClass
- Fcntl
- File
- FileList
- FileTask

- FileTest
- FileUtils
- Fixnum
- Float
- FloatDomainError
- GC
- Gem
- GetoptLong
- Hash
- IO
- IOError
- IPSocket
- IPsocket
- IndexError
- Inflector
- Integer
- Interrupt
- Kernel
- LN_SUPPORTED
- LoadError
- LocalJumpError
- Logger
- Marshal
- MatchData
- MatchingData
- Math
- Method
- Module
- Mutex
- Mysql
- MysqlError
- MysqlField
- MysqlRes
- NIL
- NameError
- NilClass
- NoMemoryError
- NoMethodError
- NoWrite
- NotImplementedError
- Numeric
- OPT_TABLE
- Object
- ObjectSpace
- Observable
- Observer
- PGError
- PGconn
- PGlarge
- PGresult
- PLATFORM

- PStore
- ParseDate
- Precision
- Proc
- Process
- Queue
- RAKEVERSION
- RELEASE_DATE
- RUBY
- RUBY_PLATFORM
- RUBY_RELEASE_DATE
- RUBY_VERSION
- Rack
- Rake
- RakeApp
- RakeFileUtils
- Range
- RangeError
- Rational
- Regexp
- RegexpError
- Request
- RuntimeError
- STDERR
- STDIN
- STDOUT
- ScanError
- ScriptError
- SecurityError
- Signal
- SignalException
- SimpleDelegater
- SimpleDelegator
- Singleton
- SizedQueue
- Socket
- SocketError
- StandardError
- String
- StringScanner
- Struct
- Symbol
- SyntaxError
- SystemCallError
- SystemExit
- SystemStackError
- TCPServer
- TCPSocket
- TCPserver
- TCPsocket
- TOPLEVEL_BINDING

- TRUE
- Task
- Text
- Thread
- ThreadError
- ThreadGroup
- Time
- Transaction
- TrueClass
- TypeError
- UDPSocket
- UDPsocket
- UNIXServer
- UNIXSocket
- UNIXserver
- UNIXsocket
- UnboundMethod
- Url
- VERSION
- Verbose
- YAML
- ZeroDivisionError
- @base_path
- accept
- Acces
- Axi
- action
- attributes
- application2
- callback
- category
- connection
- database
- dispatcher
- display1
- drive
- errors
- format
- host
- key
- layout
- load
- link
- new
- notify
- open
- public
- quote
- render
- request
- records

- responses
- save
- scope
- send
- session
- system
- template
- test
- timeout
- to_s
- type
- URI
- visits
- Observer

Database Field Names

- created_at
- created_on
- updated_at
- updated_on
- deleted_at
- (paranoia
- gem)
- lock_version
- type
- id
- #{table_name}_count
- position
- parent_id
- lft
- rgt
- quote_value

Ruby Reserved Words

- alias
- and
- BEGIN
- begin
- break
- case
- class
- def
- defined?
- do
- else
- elsif
- END
- end
- ensure
- false

- for
- if
- module
- next
- nil
- not
- or
- redo
- rescue
- retry
- return
- self
- super
- then
- true
- undef
- unless
- until
- when
- while
- yield
- _ *FILE* _
- _ *LINE* _

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

abhas	Chapter 40
Abhinay	Chapter 10
Adam Lassek	Chapters 2, 5, 6, 9, 12, 13, 38, 40 and 42
Ahsan Mahmood	Chapter 25
Aigars Cibuļskis	Chapter 5
Ajay Barot	Chapters 1 and 13
Akshay Borade	Chapter 57
Ali MasudianPour	Chapter 16
Andy Gauge	Chapter 2
AnoE	Chapter 3
Antarr Byrd	Chapter 10
Arslan Ali	Chapter 54
ArtOfCode	Chapter 20
Ashish Bista	Chapter 41
Aswathy	Chapter 1
Atul Khanduri	Chapters 7 and 14
Avdept	Chapter 13
Awais Shafqat	Chapter 35
B Liu	Chapter 37
B8vrede	Chapter 33
Bijal Gajjar	Chapter 3
br3nt	Chapters 7 and 13
Brian	Chapter 58
buren	Chapter 5
cl3m	Chapter 2
Cuisine Hacker	Chapter 20
Cyril Duchon	Chapters 2 and 17
D	Chapter 3
Dénes Papp	Chapter 32
danirod	Chapter 4
Darpan Chhatravala	Chapters 1 and 48
Darshan Patel	Chapter 1
DawnPaladin	Chapter 65
Deep	Chapters 21, 27 and 47
Deepak Kabbur	Chapter 69
Deepak Mahakale	Chapters 1, 2, 5 and 12
dgilperez	Chapters 10, 12 and 25
Dharam	Chapters 2, 15 and 32
dnsh	Chapter 13
dodo121	Chapter 12
Don Giovanni	Chapter 43
elasticman	Chapter 4
Emre Kurt	Chapter 73
esthervillars	Chapter 2
Fabio Ros	Chapters 2, 13 and 46
fiedl	Chapter 18
Fire	Chapters 2 and 21
Flambino	Chapter 7
fool	Chapter 56

You may also like

AngularJS
Notes for Professionals
100+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Ruby
Notes for Professionals
200+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

CSS
Notes for Professionals
200+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

HTML5
Notes for Professionals
100+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

HTML5
Canvas
Notes for Professionals
100+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Git
Notes for Professionals
100+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Java
Notes for Professionals
900+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

JavaScript
Notes for Professionals
400+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

jQuery
Notes for Professionals
50+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books