Directory structure:
Gemfile - contains the gems needs for the application or specific packages that are needed for the application. I think it is like composer
app/directory has all the models, controllers and views
config/directory contains config files

The routing for rails works like this:
http request -> rails router (routers.rb) -> controller -> View is generated and fed back in

The command for generating a controller is as follows:
rails generate controller Pages home about

this adds the pages to the pages_controllers.rb and creates simple view files such as
app/view/pages/about.html.erb

Instance variables can be passed around using the syntax @item1

The model is the part where the application interacts with the database. For the todo list app, it will save and read the todo entries from a database. To create a model:

rails generate model Todo todo_item:string

This command also builds a migration file which is like a schema for the database. Then, by doing rake db:migrate we can apply the changes to the database. So, migrations are similar to the version control software git. The db:migrate is like the commit command on git.

In case something goes wrong, we have a fallback command: rake db:rollback

Ruby is very similar to python because has the same architecture of a scripting language. Therefore, it has a great console like python and it can be accessed by 'rails console'. In the console, we can access a few things, such as get all the todos: Todo.all or create new todo_item:

t = Todo.new
t.todo_item = "pick up milk"
t.save

An alternative creation method would be: Todo.create(:todo_item => "Do homework")

The magic of this is that we can create SQL queries without getting bugged into the arduous code for writing queries themselves. We can instead use the function provided by rails:
Todo.find_by_field_name("Do homework")

So, in order to create a new function we have to:
Create a controller
    Add the code to the controller (todos_controller.erb)
    #Add a route if it needs it (routes.erb)
    #Create a view if necessary

To apply a css framework like bootstrap, we just need to paste the bootstrap.css inside the
app/assets/stylesheets folder and it is instantly applied. When I refreshed the browser, the bootstrap was
already applied and I didn't have to link it.

The way rails generates a view is as follows:
1. Layout is contained inside the application.html.erb and can contain a dynamic head and footer
   The content itself is placed as <%= yield %>
2. Assets pipeline and helper methods:
   <%= stylesheet_link_tag "application", :media => "all" %>
   <%= javascript_include_tag "application" %>
These two lines add any css or js inside the folders.
application.css should ideally contain css that is loaded with every page

Just like laravel, rails seems to have quite a collection of helpers, which are like small functions that
help the developer to be more productive

I have found that some of the little tutorials do not cover a lot of information and I have found a free
course on coursera called Web Application Architectures and it specifically teaches Rails. This is a real
course offered by the University of New Mexico

Notes from lectures:
web 1.0 - pretty much a client/server model - static webpages, no separation of data from separation
(read-only webpages)
   Browser wars happened - Netscape vs Microsoft IE
web 2.0 and web 3.0 are better organized to deal with complex architectures, read-write, adding meta-
data(data about data),
   approaching a billion webpages, 2.5 billion people use the internet

Design pattern - abstract template in order to solve a specific design problem that occurs while building
real systems.
n-tier architecture - separating the system into different pieces that each tier has specific functionality
Advantages:
   abstraction provides a means to deal with complexity
   tiers can be upgraded easily because they are separate - easy to scale

One of common is 3-tier:
   presentation - user interface
   application (logic) - retrieves data
      business logic tier - model
      data access tier - controller - accessing database and passing to business logic
   Data tier - source of data

Rake - RubyMake, similar to unix make
Criticism:
   Rails doesn't scale as it should - issues at Twitter
   Magic - we don't really understand much of what is going underneath
      there's a trade-off for productivity and understanding everything

The difference between scaffold and model is that the scaffold generates a controller, view and routes
like magic while model allows for more customization.

Philosophy of rails:
1. Convention over Configuration - there is standard procedure to do things well and it should be followed. Only unique features should be customized
2. Don't repeat yourself (DRY) - data should be repeated
3. Agile Development - incremental software development, get something working and add more and more
Extreme programming - agile approach centers on test-driven development.
TDD - build the test and build code that satisfies it
BDD - behavior driven language - describe in natural language - building a code that satisfies the behavior
   RSpect and Cucumber - BDD tools for Ruby

Centralized version control - if someone checkouts a file, it gets locked and other can't edit it
For distributed version control, snapshots are taken of the whole repo and compared when a change is committed.

commit -a       adds and commits in a single command
push <remote> <branch>   to a remote repository
fetch          retrieves a remote repo and need to checkout the branch
clone <remote>

The Database:
Relational databases - stores relationships between data
   id is the primary key of a table and acts as a unique identifier
normalize - separating databases and avoiding redundant data
schema is the structure of the tables in a database
to manage many-to-many relationship, we create a "joint table" and it only stores foreign keys
   ex: addresses_people: has person_id, address_id

Active Record - rails component that creates CRUD sql queries, creates ORM to conveniently get objects.

3 environments in rails: development, test and production.
To change env: rails server -e production

The ActiveRecord::Base can be inherited to get the functions. If we create class Post inheriting from it, it will assume that there will be a database class posts. Note that it pluralizes the name. Sample queries:
   Post.all, Post.first, Post.find_by(1), Post.find_by_title("First post")

Interactive Ruby Shell (IRB) is an interpreter, just like how python has an interpreter by calling python in shell.
However, 'rails console' opens up the IRB and loads the project so that we can debug the app

To create an association or a foreign key:
1. one-to-one: has_one (model with no foreign key) and belongs_to (model with foreign key)
   has_many :comments, dependent: :destroy   <-- to class Post
when a post is deleted, all of the comments are also deleted

belongs_to :post  

<-- to class Comment, note the singular post, not posts

2. many-to-one: has_many and belongs_to

3. many-to-many: has_and_belongs_to_many

Data validation:
Without validation, the app is exposed to SQL injection, cross-site scripting and buffer overflow attacks

1. Using javascript and html5 in the user's browser
2. Server-side validation - put it in the model using ActiveRecord

These "hooks" are called callbacks and the CRUD operations trigger the Validations callback. We define the validations in our models. Example:

class Person < ActiveRecord::Base
         validates_presence_of :name  //if name isn't provided, object isn't stored
         validates_numericality_of :age :only_integer => true  //only integers for age
         validates_length_of :password, :in => 8..20

Introduction to Ruby by Yukihiro Matsumoto

Philosophy: more powerful than Perl and more OO than Python

more focused on the efficiency and the productivity of the programmer

Ruby is designed to make programmers happy

Ruby:

scripting language

imperative (procedure-oriented) programming

OOP - everything is object

Functional programming

Classes:

CamelCase, ex: MyClass

for methods: my_method

@boo  //instance variable

The last execution line is returned without a return statement

'self' keyword is available

only single inheritance is supported

classes are never closed and methods can always be added

access methods: public, private (can only by invoked from self) and protected

        attr_accessor :boo  //creates an instance var boo and creates getter and setter methods
        attr_reader - creates only getter, attr_writer - only setter

all assignments in ruby are done by reference not value

Naming conventions:

my_name - local var

@name - instance var

@@name - class var

$Name - global var

A_CONST - constant

MyClass - class names
Strings:
"360 degrees is #{2*Math::PI}" - expressions calculation
`date` executes the string command
strings are mutable meaning that loops create new strings each time
string[6..15]

Regular expressions - flexible method of matching strings:
syntax - /pattern/modifiers
to match, use the =~ operator
ex: phone.gsub!(/\D/, "") removes non-numbers from phone

Expressions:
if exp1
  code
end
elsif exp2
  code
end
else
  code
end
Shorthand: code if expression (?:)

Loops:
until expression
  code
end

for var in collection do
  body
end

while condition do
  body
do
  until condition do

Last character of a method name indicates the behavior:
? boolean
! changes state

Collection classes - arrays, hash, sets
methods - sort, reverse, first, last, <<, push, pop, include?
Hash - associative array
  phone = {:home => 1, :mobile => 2}
  phone.key(1)
  phone.value?(1)

Iterators - invokes a block of code applying to each element in collection:
ex: a.each {|element| puts element}  # prints out all of the elements
Middleware - software glue that holds the client and server together. Rack - middleware stack
Rack - interface for connecting with the web servers

HTTP - application layer protocol used to distribute resources. In order to build and debug web applications, understanding of the HTTP protocol is vital. HTTP itself is a stateless protocol meaning that what the user has requests before wasn't kept track of and it made dynamic web apps hard to develop. Then, it added cookies, sessions and URL parameters to compensate.
HTTP request message - request line + header + message body
The resource is identified by URI - Universal Resource Identifier
request types - HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH

MVC Design Pattern - makes applications much easier to scale and maintain
Model - gives meaning to data
View - renders the model
Controllers - communicates between model and view

Rails Controllers:
ex- get 'products/:id' => 'catalog#view'
connects to the view method of CatalogController class, assigning 10 to params[:id]
Rails controllers are RESTful(Representational State Transfer) -- use resource routing
Fundamental philosophy: clients should communicate with servers through stateless connections
the servers will keep a long-term state of the resources
the client will be able to modify those resources through CRUD
The process is to identify the resource and applying CRUD operation on that resource
In contrast, RPC-based web apps: we need to find out the specific resources and procedures the server advertises while REST has specific simple verbs that can operate over resources.
In rails, the 'resources :posts' will create seven different routes in your application that link to methods: index, new, create, show, edit, update, destroy

The View Module
Every instance variables inside the controller is available to the view (@posts)
to create a different layout, create a layout file with the name of the controller and place it in the layouts folder.
ex: posts.html.erb

A high quality website has:
a visually appealing layout
an intuitive navigation structure - principle of least surprise
Create a wireframe
Create a navigation structure that helps you visualize
AJAX - dynamically change content by making asynchronous http requests
The easiest way to do this is through jQuery, rails contains the jquery and jquery_ujs automatically
Any forms and links that have the attribute: data-remote="true", will be submitted using jQuery's ajax method

```
<%= form_for([@post, Comment.new], remote:true) do |f| %>
  Wow that was easier than I thought. So now, the comments will be submitted asynchronously
```

Ajax process:
1. Identify the pieces of page that need to be dynamically updated
2. Create partial templates for the pieces you want to load dynamically
   `_comment.html.erb`
3. Modify the form to use Ajax (remote: true)
4. Modify the controller to respond to ajax requests (respond_to format.js)
5. Create the js code that will be returned and executed in the webpage