Web basics

Web server vs App server

Web server handles the request and its response static content only. While the app server renders the dynamic contents. Web servers can serve multiple apps .

Life cycle of Request

Client --> ISP(internet service provider) **--> DNS** (Domain name system (map a domain according to IP address))**--> App Server --> response**

HTTP vs HTTPS

SSL and TSL

Rails

Introduction

MVC, DRY and COC

Why Rails?

Rails Components

Active Record
For handling db record
Active Model
Handles db validations
Action Pack
Action Dispatch
Manage routes ….when we hit links or routes
Action controllller

Action Mailer
Sending mailsr

Action Mailbox
Receive mails
Active Jobs
Background jobs
Action Cable
End to end connection handle krne k liye
Like messenger handler...
Active Storage
Cloud data managing.
Active Support
To run the ruby code in our rails application.

Directory Structure

Lib: extended modules, task(custom rake tasks), payment method config.
Bin: scripts for starting our framework
Initializers: 3rd party gems initialize(devise)
Config: our whole configuration (db config)

config directory and its files

It covers the configuration and initialization features available to Rails applications.

config/application.rb

It allows you to specify the various settings that you want to pass down to all of the components.

config/environments/

It allows you to specify the various settings that you want to pass down to all of the components of a specific environment (development, production, test).

config/initializers

After loading the framework and any gems in your application, Rails turns to loading initializers. An initializer is any Ruby file stored under config/initializers in your application. You can use initializers to hold configuration settings that should be made after all of the frameworks and gems are loaded, such as options to configure settings for these parts.
There is no guarantee that your initializers will run after all the gem initializers, so any initialization code that depends on a given gem having been initialized should go into a config.after_initialize block.

config/locales/

Files in the config/locales directory are used for internationalization and are automatically loaded by Rails. If you want to use locales other than English, add the necessary files in this directory.

boot.rb

In a standard Rails application, there's a Gemfile which declares all dependencies of the application. config/boot.rb sets ENV['BUNDLE_GEMFILE'] to the location of this file. If the Gemfile exists, then bundler/setup is required. The require is used by Bundler to configure the load path for your Gemfile's dependencies.

Environment.rb

This file is the common file required by config.ru (rails server) and Passenger. This is where these two ways to run the server meet; everything before this point has been Rack and Rails setup.

Puma.rb

Puma was built for speed and parallelism. Puma is a small library that provides a very fast and concurrent HTTP 1.1 server for Ruby web applications.

Spring.rb

Spring is a Rails application preloader. It speeds up development by keeping your application running in the background so you don't need to boot it every time you run a test, rake task or migration.

Routes.rb

Routes are defined in app/config/routes.rb.

config/database.yml

Using the config/database.yml file you can specify all the information needed to access your database.

```
development:
  adapter: postgresql
  database: blog_development
  pool: 5
```

This will connect to the database named `blog_development` using the `postgresql` adapter. This same information can be stored in a URL and provided via an environment variable.

Rake Tasks

db:migrate vs db:schema:load

- **db:migrate** runs (single) migrations that have not run yet.
- **db:migrate:up** runs one specific migration
- **db:migrate:down** rolls back one specific migration
- **db:migrate:status** shows current migration status.
- **db:rollback** rolls back the last migration.
- **db:forward** advances the current schema version to the next one.
- **db:seed** (only) runs the db/seed.rb file
- **db:create** creates the database for current environment.
- **db:create:all** creates the databases for all envs.
- **db:drop** deletes the database
- **db:schema:load** creates tables and columns within the (existing) database following schema.rb
- **db:setup** does db:create, db:schema:load, db:seed
- **db:reset** does db:drop, db:setup
- **db:migrate:reset** does db:drop, db:create, db:migrate

Typically, you would use db:migrate after having made changes to the schema via new migration files (this makes sense only if there is already data in the database). db:schema:load is used when you set up a new instance of your app.

ORM

ORM is a technique that connects the rich objects of an application to tables in a relational database management system. Using ORM, the properties and relationships of the objects in an application can be easily stored and retrieved from a database without writing SQL statements directly and with less overall database access code.

Asset pipeline

How is it useful?

The first feature of the pipeline is to concatenate assets, which can reduce the number of requests that a browser makes to render a web page. Sprockets concatenates all JavaScript files into one master .js file and all CSS files into one master .css file.

The second feature of the asset pipeline is asset minification or compression. For CSS files, this is done by removing whitespace and comments. For JavaScript, more complex processes can be applied. You can choose from a set of built in options or specify your own.

The third feature of the asset pipeline is it allows coding assets via a higher-level language, with precompilation down to the actual assets. Supported languages include Sass for CSS, CoffeeScript for JavaScript, and ERB for both by default.

Fingerprinting (cache busting)?

How it is categorized

Pipeline assets can be placed inside an application in one of three locations: app/assets, lib/assets or vendor/assets.

app/assets

is for assets that are owned by the application, such as custom images, JavaScript files, or stylesheets./Projects

lib/assets

is for your own libraries' code that doesn't really fit into the scope of the application or those libraries which are shared across applications.

vendor/assets

is for assets that are owned by outside entities, such as code for JavaScript plugins and CSS frameworks. Keep in mind that third party code with references to other files also processed by the asset Pipeline (images, stylesheets, etc.), will need to be rewritten to use helpers like asset_path.

Sprockets

Sprockets is a Ruby library for compiling and serving web assets. Sprockets allows organizing an application's JavaScript files into smaller more manageable chunks that can be distributed over a number of directories and files. It provides structure and practices on how to include assets in our projects.

Using directives at the start of each JavaScript file, Sprockets can determine which files a JavaScript file depends on. When it comes to deploying your application, Sprockets then uses these directives to turn your multiple JavaScript files into a single file for better performance.

Require_tree and require_self

The require directive is used to tell Sprockets the files you wish to require.
The require_tree directive tells Sprockets to recursively include all JavaScript files in the specified directory into the output.
require_self puts the CSS contained within the file (if any) at the precise location of the require_self call.

Scss

Scss is a pre-processor scripting language.that interpret or compiled into css. It has options for nested blocks, variables, partials, modules, mixin, operators.

Coffee script

Coffee script that compiles to js and adds syntactic sugar to js,inspired by ruby python. It gives smaller code for js functionality

Jquery

Jquery is js library, designed to simplify html dom tree traverse and manipulation and event handling, css animation and ajax. It is cross platform

Bootstrap

Bootstrap is used to customize and responsive mobile first sites. Uses a responsive grid system. 12 columns.

Active records Models

Models in Rails are SQL tables with your typical columns, primary key, and foreign key
.

Convention over Configuration in Active Record

When writing applications using other programming languages or frameworks, it may be necessary to write a lot of configuration code. This is particularly true for ORM frameworks in general. However, if you follow the conventions adopted by Rails, you'll need to write very little configuration (in some cases no configuration at all) when creating Active Record models. The idea is that if you configure your applications in the very same way most of the time then this should be the default way. Thus, explicit configuration would be needed only in those cases where you can't follow the standard convention.

Callbacks

During the normal operation of a Rails application, objects may be created, updated, and destroyed. Active Record provides hooks (called callbacks) into this object life cycle so that you can control your application and its data. Callbacks allow you to trigger logic before or after an alteration of an object’s state.

Object Life Cycle

Before_validation
After_validation
Before_save
Around_save
Before_create (_update)
Around_create (_update)
After_create (_update)
After_save
After_commit/ After_rollback

after_save vs after_update

The after_save callback is called both when a record has been created and updated.

The after_update callback is called on a persisted record respectively.

after_commit

When data is persisted and trancaction is complete

What is mass assignment

Mass Assignment is the name Rails gives to the act of constructing your object with a parameters hash. It is "mass assignment" in that you are assigning multiple values to attributes via a single assignment operator.

The following snippets perform mass assignment of the name and topic attribute of the Post model:

Post.new(:name => "John", :topic => "Something")
Post.create(:name => "John", :topic => "Something")
Post.update_attributes(:name => "John", :topic => "Something")
Post.new(:name => "John", :topic => "Something")

Post.create(:name => "John", :topic => "Something")

Post.update_attributes(:name => "John", :topic => "Something")

Attr_accessor vs att_accessible
attr_accessor is a Ruby method that makes a getter and a setter.

attr_accessible is a Rails method that allows you to pass in values to a mass assignment: new(attrs) or update_attributes(attrs).

Here's a mass assignment:

Order.new({ :type => 'Corn', :quantity => 6 })

You can imagine that the order might

Attr_accessor vs att_accessible

attr_accessor is a Ruby method that makes a getter and a setter.
attr_accessible is a Rails method that allows you to pass in values to a mass assignment: new(attrs) or update_attributes(attrs).

Here's a mass assignment:
Order.new({ :type => 'Corn', :quantity => 6 })

You can imagine that the order might also have a discount code, say :price_off. If you don't tag :price_off as attr_accessible you stop malicious code from being able to do like so:

Order.new({ :type => 'Corn', :quantity => 6, :price_off => 30 })
Even if your form doesn't have a field for :price_off, if it's in your model it's available by default. This means a crafted POST could still set it. Using attr_accessible white lists those things that can be mass assigned.

Require vs Permit

The `require` method ensures that a specific parameter is present, and if it's not provided, the `require` method throws an error. It returns an instance of `ActionController::Parameters` for the key passed into `require`.
The `permit` method returns a copy of the parameters object, returning only the permitted keys and values. When creating a new ActiveRecord model, only the permitted attributes are passed into the model.

What is a dirty object?

An object becomes dirty when it has gone through one or more changes to its attributes and has not been saved. ActiveModel::Dirty gives the ability to check whether an object has been changed or not.

Method Chaining

A feature of using where is the ability to method chain. Method chaining is only allowed in a statement if the previous method returns an ActiveRecord::Relation, like where and all. Methods that only return a single object have to be at the end of the statement.

Entry.where(user_id: 1).find_by(completed: false) # => #<ActiveRecord::
Relation #<Entry id: 1, user_id: 1, completed: false>, #<Entry id: 3,
user_id: 1, completed: false>...]>

Nested Attributes

Nested attributes is something built into ActiveRecord and there's just a few short steps to add it to your project. When added, it allows you to create rows in multiple tables/databases at the same time.
Accepts_nested_attributes_for

https://medium.com/@adam.n.conway/nested-attributes-a9bebd91e1ca

Delete vs destroy

delete(id_or_array) *public*

Deletes the row with a primary key matching the `id` argument, using a SQL `DELETE` statement, and returns the number of rows deleted. Active Record objects are not instantiated, so the object's callbacks are not executed, including any `:dependent` association options.

You can delete multiple rows at once by passing an Array of `ids`.

Note: Although it is often much faster than the alternative, `#destroy`, skipping callbacks might bypass business logic in your application that ensures referential integrity or performs other essential jobs.

destroy(id) *public*

Destroy an object (or multiple objects) that has the given id. The object is instantiated first, therefore all callbacks and filters are fired off before the object is deleted. This method is less efficient than ActiveRecord#delete but allows cleanup methods and other actions to be run.

This essentially finds the object (or multiple objects) with the given id, creates a new object from the attributes, and then calls destroy on it.

Parameters

- `id` - Can be either an Integer or an Array of Integers.

Active Records Validation

Validate vs Validates

validates is used for normal validations presence , length , and the like.
validate is used for custom validation methods validate_name_starts_with_a , or whatever crazy method you come up with. These methods are clearly useful and help keep data clean.

Size vs count

Size and **count** both return the number of items in an array. ... **count** is always going to run a query in the database while **size** will return the number of items in the collection based on the objects currently in the object graph.

Scopes

Scoping allows you to specify commonly-used queries which can be referenced as method calls on the association objects or models. With these scopes, you can use every method previously covered such as where, joins and includes. All scope bodies should return an

ActiveRecord::Relation or nil to allow for further methods (such as other scopes) to be called on it.

To define a simple scope, we use the scope method inside the class, passing the query that we'd like to run when this scope is called:

```
class Article < ApplicationRecord
  scope :published, -> { where(published: true) }
end
```

```
Article.published # => [published articles]
```

Active Records Queries

Find() vs Find_by()

find()
The **find** method retrieves the first object that matches the specified primary key (generally the id).
If a record is not found, it will raise an ActiveRecord::RecordNotFound exception.

Model.find(id)

record found
User.find(4) # => #<User id: 4, name: "Batman">

record not found
User.find(53) # ActiveRecord::RecordNotFound (Couldn't find
User with 'id'=53)
find_by()

Another way to locate a single model instance is **find_by**. The find_by method returns the first record that matches a given condition. The condition is the key value pair argument of what you're looking for. Optionally, you can pass in several arguments as well. If a record is not found, nil is returned. Say you want to find a user that has a pet cat, your Active Record query would like would:
Model.find_by(key: value)

record found & single argument
User.find_by(pet: 'cat') # => #<User id: 1, pet: “cat”, location: "SF">

record found & multiple arguments

User.find_by(pet: 'cat', location: 'NY') # => #<User id: 5, pet: "cat",
location: "NY">

record not found
User.find_by(pet: 'iguana') # => nil

Find() vs where()

.find() and .find_by() return single instances of an object, .where() returns what is called an “ActiveRecord::Relation object”, essentially an array of instances with some extra information attached. We can use chaining methods with where but cannot use it with find.

Exist vs present

Aggregate functions

Group

Select vs pluck

Select by columns, return active record object, while pluck return array.
With Select we can use chaining methods(by using where clause only), while in pluck we can't.

eager loading vs lazy loading

Eager Loading
Eager loading is the solution to N+1 problem. (User-to-friends example)

One way to improve performance is to cut down on the number of SQL queries. You can do this through eager loading.

User.find(:all, :include => :friends)
Here you are firing only two queries :

1) One for all users.

2) One for all friends of users.

Pro: is that everything's ready to go.

Con: you are using up space/memory.

Lazy Loading :

When you have an object associated with many objects like a User has many Friends and you want to display a list as in Orkut you fire as many queries as there are friends, plus one for the object itself.

users = User.find(:all)
Then query for each user friend , like :

```
users.each do |user|
  friend = Friend.find_by_user_id(user.id)
end
```

Here

1) One query for all users.

2) N query for N no. of users friends.

Pro of lazy loading: you don't hit the database until you need to.

Con: You'll be hitting the database N + 1 times.....unless you select exactly the column you want and you alias it. e.g.

joins vs includes

Includes uses eager loading whereas joins uses lazy loading.
https://medium.com/@swapnilggourshete/rails-includes-vs-joins-9bf3a8ada00

Locking

Optimistic Locking

Optimistic Locking assumes that a database transaction conflict is very rare to happen. It uses a version number of the record to track the changes. It raises an error when another user tries to update the record while it is locked.
Just add a **lock_version** column to the table you want to place the lock and Rails will automatically check this column before updating the record.

```
c1 = Client.find(1)
c2 = Client.find(1)

c1.first_name = "Michael"
c1.save

c2.name = "should fail"
```

```
c2.save # Raises an ActiveRecord::StaleObjectError
```

Pessimistic Locking

Pessimistic locking assumes that database transaction conflict is very likely to happen. It locks the record until the transaction is done. If the record is currently locked and the other user makes a transaction, that second transaction will wait until the lock in the first transaction is released.

```
user = User.lock.find(1) #lock the record
user.name = 'ryan'
user.save!
```

Active Records Migrations

Migrations are a convenient way to alter your database schema over time in a consistent and easy way. They use a Ruby DSL so that you don't have to write SQL by hand, allowing your schema and changes to be database independent.

Up and down methods

The up method is called when migrating “up” the database – forward in time – while the down method is called when migrating “down” the database – or, back in time. In other words, the up method is a set of directions for running a migration, while the down method is a set of instructions for reverting a migration. This implies that the code in these two methods should fundamentally do the opposite things of one another.

Change method

The change method is pretty standard when it comes to migrations partly because it’s a newer addition to Rails. Just like up and down, the change method is defined on the ActiveRecord::Migration class. In fact, it does exactly what up and down accomplish together. The change method is usually able to automatically figure out the inverse operation you provide it; for example, if you call create_table inside of the change method, when you run rake db:rollback, it will drop_table. The same goes for add_column and remove_column.

So, if the change method can do all of these things in one go (rather than in two methods), why do we sometimes see an up and down method defined together in a migration file?

Well, there are many times when we might want Active Record to be smart and figure out when to drop a column or table. But other times, it might not be as clear.
For example, what if we wanted a migration that just created or fixed data? We wouldn’t want ActiveRecord to try to figure out whether to add or remove a column…or worse, drop our table!

Or what if we wanted to remove columns when we migrated up, and add columns when we migrated down? We'd have to specify that explicitly in our up and down methods.

Reversible Migrations

Reversible migrations are migrations that know how to go down for you. You simply supply the up logic, and the Migration system figures out how to execute the down commands for you.

To define a reversible migration, define the change method in your migration like this:

```
class TenderloveMigration < ActiveRecord::Migration[5.0]
  def change
    create_table(:horses) do |t|
      t.column :content, :text
      t.column :remind_at, :datetime
    end
  end
end
```

This migration will create the horses table for you on the way up, and automatically figure out how to drop the table on the way down.

Some commands cannot be reversed. If you care to define how to move up and down in these cases, you should define the up and down methods as before.

If a command cannot be reversed, an ActiveRecord::IrreversibleMigration exception will be raised when the migration is moving down.

Transactional Migrations

On databases that support transactions with statements that change the schema, migrations are wrapped in a transaction. If the database does not support this then when a migration fails the parts of it that succeeded will not be rolled back. You will have to rollback the changes that were made by hand.

Active Records Callbacks

before/after_commit vs before/after_create

before/after_validation, around_save etc

Get old values of attributes in callbacks, check if attribute changed, append errors(invalidate record)

var_name_changes, var_name_was

Active Records Associations

Associations are basically defining the relationship between models.
In Associations, we are executing either a JOIN SQL query (if there's a through-table) or a normal SELECT query. Look at the code below for example:

User.find(1).subscriptions
SELECT * FROM subscriptions WHERE user_id = 1 LIMIT 1

These two lines of code do exactly the same thing, the Rails code is being turned into its equivalent SQL query.

How to achieve association

Association can be achieved using below given keywords depending upon the condition.

- belongs_to
- has_one
- has_many
- has_many :through
- has_one :through
- has_and_belongs_to_many

Now you're probably noticing the :through keyword that is being used by has_many and has_one. This is used when there's a third model (table) involved.

Types of associations

One to one:

When you set up one-to-one relations you are saying that a record contains exactly one instance of another model. For example, if each user in your application has only one profile, you can describe your models as:

```
class User < ApplicationRecord      class Profile < ApplicationRecord
  has_one :profile                          belongs_to :user
end                                 end
```

One of the models in the given relation will have has_one method invocation and another one will have _ belongs_to. It is used to describe which model contains a foreign key reference to the other one, in your case, it is the profiles model.

One to Many:

A one-to-many association is the most common used relation. This association indicates that each instance of the model A can have zero or more instances of another model B and model B belongs to only one model A.

Let's check it out via example. Let’s say you want to create an application where users can write multiple stories, your models should look like this:

```
class User < ApplicationRecord      class Story < ApplicationRecord
  has_many :stories                         belongs_to :user
end                                 end
```

Here, deciding which model will have has_many and which will have belongs_to is more important than in one-to-one relations, because it changes the logic of your application. In both cases, the second model contains one reference to the first model in the form of a foreign key.

The second model doesn't know about the first model relation to it - it is not aware if the first model has a reference to more than one of them or just to one.

Many to Many:

Many-to-many associations are a bit more complex and can be handled in two ways, "has and belongs to many" and "has many through" relations.

Has and Belongs to Many:

A has_and_belongs_to_many association creates a direct many-to-many connection with another model. It is simpler than the other one, as it only requires calling has_and_belongs_to_many from both models.
Example: Let's say, a user can have many different roles and the same role may contain many users, your models would be like this:

```
class User < ApplicationRecord
  has_and_belongs_to_many :roles
end
```

```
class Role < ApplicationRecord
  has_and_belongs_to_many :users
end
```

You will need to create a join table for this association to work. It is a table that connects two different models. The join table is created with the rails function create_join_table :user, :role in a separate migration.

```
class CreateUserRoles < ActiveRecord::Migration
  def change
    create_table :user_roles, id: false do |t|
      t.references :user, index: true, foreign_key: true
      t.references :role, index: true, foreign_key: true
    end
  end
end
```

This is a very simple approach, but you don't have the direct access to related objects, you can only hold references to two models and nothing else.

Has_many Through:

Another way to define a many-to-many association is to use the has_many through association type. Here you should define a separate model, to handle that connection between two different ones.
Using the example of has_and_belongs_to_many association, this time the three models should be written like this:

```
class User < ApplicationRecord
  has_many :user_roles
  has_many :roles, through: :user_roles
```

```
end

class UserRoles < ApplicationRecord
  belongs_to :user
  belongs_to :role
end

class Role < ApplicationRecord
  has_many :user_roles
  has_many :users, through: :user_roles
End
```

This association will enable you to do things like **user.role** and to get a list of all connected second model instances. It will also enable you to get access to data specific to the relation between first and second models.

Polymorphic associations:

Polymorphic associations are the most advanced associations available to us. You can use it when you have a model that may belong to many different models on a single association.

Let's imagine you want to be able to write comments for users and stories. You want both models to be commentable. Here's how this could be declared:

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true
end

class Employee < ApplicationRecord
  has_many :comment, as: :commentable
end

class Product < ApplicationRecord
  has_many :comment, as: :commentable
End
```

You can think of a polymorphic belongs_to declaration as setting up an interface that any other model can use. To declare the polymorphic interface you need to declare both a foreign key column and a type column in the model. You should run the migration once you are done.

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.text :body
```

```
      t.integer :commentable_id
      t.string :commentable_type
      t.timestamps
    end

    add_index :comments, :commentable_id
  end
End
```

This migration can be simplified by using references:

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.text :body
      t.references :commentable, polymorphic: true, index: true
      t.timestamps
    end
  end
end
```

STI and ENUMS

Dependent

Active Records Routing

Shallow routing

From **Jamis Buck's blog** post the resources should never be nested more than 1 level deep. Buck's blog post suggests an advanced method for keeping your routes and URLs from getting out of control.

Adding the ‘shallow: true’ parameter to your nested resource will define routes at the base level, ‘/songs/’, for four of our RESTful routes — **show, edit, update and destroy**, while leaving the remaining routes — **index, new, create**— at the nested level. So how is this useful to us?
In my model, with artists and songs, we are now able to use the routes starting with ‘/songs/:id’ to view or modify existing song0.0s on an individual basis. Recall that songs belong to artists, so upon creation, we still need to initialize a song with an artist relationship. However, once a song is created, it doesn’t matter to the artist relationship if we would like to edit or delete it. And so, we don’t need to follow a lengthy route, such as ‘/artists/:artist_id/songs/:id/edit’ to make our changes.

Member routes vs Collection routes

A member route will require an ID, because it acts on a member. A collection route doesn't because it acts on a collection of objects. Preview is an example of a member route, because it acts on (and displays) a single object. Search is an example of a collection route, because it acts on (and displays) a collection of objects.

PUT vs Patch

when you want to update a resource with a PUT request, you have to send the full payload as the request whereas with PATCH , you only send the parameters which you want to update.

PUT vs Post

The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly has side effects of creating the same resource multiple times.

Resource vs Resources

Sometimes, you have a resource that clients always look up without referencing an ID. For example, you would like /profile to always show the profile of the currently logged in user. In this case, you can use a singular resource to map /profile (rather than /profile/:id) to the show action.

A good way to see it is that the resource does not have an index action, since it's supposed to be just one.

Helper methods

A helper is a method that is (mostly) used in your Rails views to share reusable code. Rails comes with a set of built-in helper methods.

One of these built-in helpers is **time_ago_in_words.**

Render vs redirect

Render tells Rails which view or asset to show a user, without losing access to any variables defined in the controller action. Redirect is different. The redirect_to method tells your browser to send a request to another URL.

Mount

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

scopes in details

concerns in rails

Find vs find_by, where,

join, includes, left_outer_join,

where.not,

pluck vs select,

group in DB,

aggregate functions(sum, avg, min, max),

first_or_create etc

after_commit vs after_create

before_validation

Get old values of attributes in callbacks

So if you have an attribute total, you will have a total_changed? method and a total_was method that returns the old value.

Scopes

Scopes are custom queries that you define inside your Rails models with the scope method.

Every scope takes two arguments:

A name, which you use to call this scope in your code
A lambda, which implements the query
It looks like this:

```
class Fruit < ApplicationRecord
  scope :with_juice, lambda { where("juice > 0") }
end
```

When To Use Scopes?

Ok, scopes are cool, but when should you use them?

Let's see an example.

```
1. def index
2.  @books = Book.where("LENGTH(title) > 20")
3. end
```

This is an index controller action that wants to display books with titles longer than 20 characters.

It's fine.

But if you want to use this query in other places, you're going to have duplicated code.Duplicated code makes your project harder to maintain.Let's move this query into a scope.

Like this:

```
1. class Book
2.  scope :with_long_title,->{where("LENGTH(title)20")}
3. End
```

Now our controller action looks like this:

```
def
  @books = Book.with_long_title
end
```

How to Use Rails Scopes With Arguments:

You may want to introduce a variable into a scope so you can make it more flexible.
Here's how:

```
class Book
  scope :with_long_title, ->(length)  { where("LENGTH(title) > ?", length) }
end
```

Here "?" is a placeholder for length.

Data seeding (seeds.rb):

With the file db/seeds.rb, the Rails gods have given us a way of feeding default values easily and quickly to a fresh installation. This is a normal Ruby program within the Rails environment. You have full access to all classes and methods of your application.
So you do not need to enter everything manually with rails console in order to make the records created in the section called "create" available in a new Rails application, but you can simply use the following file db/seeds.rb:
Country.create(name: 'Germany', population: 81831000)
Country.create(name: 'France', population: 65447374)
Country.create(name: 'Belgium', population: 10839905)
Country.create(name: 'Netherlands', population: 16680000)

You then populate it with data via **rake db:seed**. To be on the safe side, you should always set up the database from scratch with rake db:setup in the context of this book and then automatically populate it with the file db/seeds.rb.

The db/seeds.rb is a Ruby program. Correspondingly, we can also use the following approach as an alternative:

```
country_list = [
  [ "Germany", 81831000 ],
  [ "France", 65447374 ],
  [ "Belgium", 10839905 ],
  [ "Netherlands", 16680000 ]
]

country_list.each do |name, population|
  Country.create( name: name, population: population )
end
```

Controllers

Callbacks

Methods

#

- _insert_callbacks,
- _normalize_callback_options

A

- after_action,
- append_after_action,
- append_around_action,
- append_before_action,
- around_action

B

- before_action

P

- prepend_after_action,
- prepend_around_action,
- prepend_before_action

S

- skip_after_action,
- skip_around_action,
- skip_before_action

Service objects

Service objects are plain old Ruby objects (PORO's) that do one thing.
They encapsulate a set of business logic, moving it out of models and controllers and into a more focused setting.
Here's what a RegisterUser service object might look like:

```
class RegisterUser
  def initialize(user)
    @user = user
  end
  def execute
    return nil unless @user.save
    send_welcome_email
    notify_slack
    if @user.admin?
      log_new_admin
    else
      log_new_user
    end
    @user
  end
  # private methods
end
```

Our service object takes a newly instantiated user on initialize, and will either return a saved version of that user, or nil if there were problems saving.
And our slim controller:

```
class UserController < ApplicationController
  def create
    user = RegisterUser.new(User.new(user_params)).execute
    if user
      redirect_to new_user_welcome_path
    else
      render 'new'
    end
  end
  # private methods
```

```
end
```

Concerns

The Concern is a tool provided by the ActiveSupport lib for including modules in classes, creating mixins.

```
module Emailable
  include ActiveSupport::Concern
  def deliver(email)
    # send email here...
  end
end
class Document
  include Emailable
  def archive
    @archived = true
    deliver({to: 'me@mydomain.com', subject: 'Document archived', body: @content})
  end
end
```

Sounds great, right? Any class including our Emailable concern would be able to send emails. Unfortunately, not every example in a common Rails project is as clear as the one explained above.

Rails service objects vs concerns

The chief difference between ActiveSupport::Concerns and service objects is that concerns are modules that are intended to be mixed into other classes, while service objects are classes that can be instantiated. Concerns are used when there is shared functionality among several classes.

Views

Partial vs Views vs Layout

Helpers

Collections, without loop rendering

Form_for vs Form_with vs Form_tag

form_tag

Form_tag generates an HTML form for us and lets you specify options you want for your form.

form_for

form_for method follows RESTful conventions on its own. It accepts the instance of the model as an argument where it makes assumptions for you (which is why it can be seen to be preferred over form_tag)

What's with the form_with !?

The form_with view helper came out in 2017 with the idea to unify the interface of what form_for AND form_tag can do (so it is expected that form_with will be the one more commonly applied). When you pass it a model, it will act the same way as form_for would.

Middlewares

Jobs in rails

Webpacker

Starting with Rails 6, Webpacker is the default JavaScript compiler. It means that all the JavaScript code will be handled by Webpacker instead of the old assets pipeline aka Sprockets. Webpacker is different from asset pipeline in terms of philosophy as well as implementation.

webpacker is a gem which wraps webpack - the popular JavaScript tool used for managing and bundling JavaScript code - and provides helpers to use the webpack in our Rails applications. In simple words it provides Rails a way of using webpack. This is a very simple definition for a tool which is very powerful but that is enough for us as of now.

Webpacker wraps webpack in a Ruby gem and provides helpers to use the output from Webpacker in the Rails application.

Rubocop (linter) gem

A Ruby static code analyzer and formatter, based on the community Ruby style guide. Out of the box it will enforce many of the guidelines outlined in the community Ruby Style Guide. Apart from reporting the problems discovered in your code, RuboCop can also automatically fix many of them.

Brakeman gem

Brakeman is a command-line tool that analyzes the source code of Ruby on Rails applications to find potential security vulnerabilities.

Rubycritic gem

RubyCritic is a gem that wraps around static analysis gems such as Reek, Flay and Flog to provide a quality report of your Ruby code.

Bullet gem

help to kill N+1 queries and unused eager loading.

Overcommit gem

overcommit is a tool to manage and configure Git hooks.

Byebug gem

Byebug is a simple to use and feature rich debugger for Ruby. Byebug is also fast because it is developed as a C extension and reliable because it is supported by a full test suite. The debugger permits the ability to understand what is going on inside a Ruby program while it executes and offers many of the traditional debugging features such as:

- Stepping: Running your program one line at a time.
- Breaking: Pausing the program at some event or specified instruction, to examine the current state.
- Evaluating: Basic REPL (read–eval–print loop) functionality, although pry does a better job at that.
- Tracking: Keeping track of the different values of your variables or the different lines executed by your program.

Pundit gem

Figaro gem

Pagination

Different options for pagination in Rails

- **Kaminari**
- **Will_paginate**
- **pagy**

Frontend pagination vs backend pagination.

For Server Side Pagination:

1. your request time and data are reduced, as only a selected no of rows will be sent by the server.
2. browser required less memory hence faster to process like filter, map, reduce etc.(only for one page)

For Client Side Pagination:

1. As all data is present on client machine user can easily switch between back and forth.
2. filter, search, map, reduce is possible on whole data.
3. server get few requests as for search, filter, etc needed extra request and many iterations to the server.