- **-Rails introduction**
- **COC DRY and AGILE**

// Convention over Configuration // Don't repeat Yourself //

- **Scaffolding**

// Scaffolding creates 5 things
// Models, Views, Controllers, Routes, Testing

- **Active model and Active Record diff**

// Active Model contains all the classes that cannot be mapped on the database // Validation //
// Active Record contains the classes that can be mapped to the database (implements ORM)

- **What is MVC**

MVC is Model, View, Controller. It is an architectural structure method which divides the functionality into three logical components.

- **ORM mapping in Rails**

// Object relational mapping // means the mapping of the database tables to the class objects. Implemented by Active Records

- **Difference in lib/assets, vendor/assets, app/assets**

// vendor/assets is for assets that are owned by outside entities, such as code for JavaScript plugins and CSS frameworks.
// 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.
// app/assets is for assets that are owned by the application, such as custom images, JavaScript files, or stylesheets.

- **Lib vs gem difference**

// Lib is any piece of code where gem is a ruby written code with a .gemspec file,
// Can be created by gem build name.gemspec

- **Public folder**

// Any assets under public will be served as static files by the application or web server when config.public_file_server.enabled is set to true. You should use app/assets for files that must undergo some preprocessing before they are served.
In production, Rails precompiled these files to public/assets by default. The precompiled copies are then served as static assets by the web server. The files in app/assets are never served directly in production.

- **What is Rake task**

// Rake is a task runner, you can run multiple predefined tasks by rails like db:migrate, db:seed, db:rollback

// If you want to run custom rake tasks you can add them to the lib/tasks directory with the extension of .rake and not .rb, To view all the rake tasks you can do rake -vT.

- **DB rollback 3 steps back**

Rolls back 3 Migrations

- **seeding , why seeding?**

// To populate the db with data

- **Gemfile.lock. Delete gemfile.lock**

Gemfile is where you tell which gems are required and bundler will install, in gemfile.lock the bundler writes the versions of gems that are installed
// When deleted gemfile.lock, it created itself.

- **Sass vs scss, css vs scss (mixin, variable, custom functions)**

SASS Syntactically Awesome Style Sheets
Strict Indentation, No semi colons, no brackets
SCSS Not that strict syntax, no indentations, uses semicolons and brackets

- **Background jobs** how many by default (0, you can declare 5 jobs in 1 queue)

//

- **Turbolinks**

Turbolinks is a rails feature available as a gem and active by default. Intercepts all the clicks that navigates to another page of the app and makes the request via AJAX replacing the body with the received content

- **Transpilation**

// Taking source code in one language and converting it to another language is called transpilation

- **Hash Indifferent Access**

// The hash keys are considered as a string and the index you will try to access will be treated as a key and not the index integer
// Implements a hash where keys :foo(symbol) and "foo"(string) are considered to be the same.

- **Active model vs active record callbacks**

// Active Model is the model which does not contain any relations with the database whereas their callbacks have to be defined // On the other hand the active records are the models which are being stored in the database and they have pre defined callbacks just like ‘before_update’, ‘before_save’

- **indexes , which column and why?**

// Indexes are used to speed up the searching in the data table and is defined with the column while making up the migration email:string:index

- **Up, down and change, difference**

// Down defines our migrations to which we want to rollback to and Up defines the migration we want to put on when we call migrate. Change defines the changes we are making to the database.

- **Where can't u use change (where SQL query is called -> where there is concern is used)**

// You cant use change where you are writing some SQL query which is out of rails convention, such ass using a SoftDelete, the rails wont know what will be the inverse of the query So we have to manually tell, Or go to the Up Down migration.

- **Validate vs validates**

// validates is used for the predefined validations that the rails has provides and on the other hand the validate is used to define our own validations or custom validations

- **How does custom validation fail in Rails?**

// Adding an error message in the errors array will do the trick.

- **Sequence of callbacks in Model**

// Validation -> Save/Destroy -> After Save/Destroy -> Commit

- **After_save and commit**

// after_save is triggered when an object is saved to the database successfully and commit is called when an object is created/ updated or deleted and saved successfully

- **Around callbacks**

// Around call back is triggered with the before and ends when the object is saved. Then the after callback is triggered.

- **Commit and save difference**

// The save callback is invoked on create and update where commit is invoked where a transaction is completed.
A transaction is started when a save, update, destroy or delete is called on the object and once all the steps are performed successfully and the object is saved/updated/deleted to the database, a commit is called to sign off the completion of the transaction.

- **Old values, DIRTY method**

// Dirty method is used to see the changes that have happened to an object and what its previous states were. Using was, changed?, rollback!

- **After_destroy callback, do we have reference for the object?**

// After_destroy is triggered when the object is destroyed and removed. And for the reference for the object we have self.Object

- **Why do we use serializers?**

// To change the object of the class to a hash/json object

- **What if we dont want either fat or slim (concern)**

// When our controller is getting fat, one of the reasons maybe, code repetition, to solve that we can use concerns to make our controller less fat
// Arguably one of the most important ways to write clear and concise code in Ruby on Rails, the motto “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

- **Controller callbacks**

// The callbacks are triggered when a controller is called to be executed, which include after_action, around_action, before_action

- **Require vs permit**

// Require returns error if the params passed are nil or the object is of different type || checks if the passed object is present in the params and permit returns only the values for the symbols that are passed in arguments

- **Permit vs permit!**

// permit only takes the arguments with the symbols already passed to permit whereas the permit! Takes no arguments and returns all the data which sometime may be useful and maybe sometime open up mass assignment vulnerabilities

- **How to handle JSON or HTML request in controller**

- **CRSF token**

//CSRF token is generated by the server and sent to the client in the http response and then is used to encrypt the data with the randomly generated number.

- **Groups in gemfile**

// Group keyword is used to create the groups of gems in the gemfile So we can use bundler to tell the bundler command which group of gems we want to install for specific environment.

- **Tilde arrow and <> difference in gemfile**

// Tilde arrow is used to define the version of the gemfile we want to add to our rails application. And < and > are used to define the version greater than or less than the mentioned version.

- **Action Controller**

// Action controller is the C of MVC and is responsible for making the sense of the request and returning the appropriate output. Controller is called when the routes defined determine which controller is used for the required request

- **Sprockets**

// Sprockets is a asset manager used for compiling and serving the web assets

- **ERB**

// ERB is embedded ruby code, which is used in html extension to use ruby code in html file and manipulate the data coming from the backend to be manipulated and viewed by the application.

- **ActionController::Base**

// Base class of action controller, when inherited by any controller, adds some important functionalities into the class that inherits from the base class, like handling requests.

- **Fat vs slim controller**

// Fat vs skinny controller means that all the business logic which is response related should be present in the controller and all the other business logic should be defined in the model.

- **Routing ins and outs**

// Rails routing is a two-way piece of machinery – it both connects incoming HTTP requests to the code in your application's controllers, and helps you generate URLs without having to hard-code them as strings.

- **Routs.rb => Controller**

// When a url is requested, it goes into the routes.rb file and look for the method to execute from the controller, both of them are places in the routes.rb file with the naming convention of controller#method

- **Nested Routes**

// Nested routes are used when we are creating routes for the models having a relationship // One controller is used for the nested routes which is from the child class (don’t say actual words, just for understanding)

- **Action Form Helpers**

// The helpers which help in creating forms in html.erb file in the erb are action Form helpers, they can create the forms with the class variables and generate the labels and the text fields for them.

- **Action View**

// Action Controller is responsible for CRUD and other functionalities including communication with models/DB on the other hand Action View is responsible for compiling the response. Action views are written in embedded ruby and html

- **Render**

// In most of the cases Render is used to render the response to the website, Render can be used in multiple ways which includes rendering a Action View, a json, xml or just some output from a symbol

- **Shallow Nesting**

// Shallow nesting is where we use nested routes but this time we use the only: keyword aur nest only the required, we can use “shallow: true” keyword if not wanting to specify which routes to be nested

- **Namespaces vs Scope**

// Namespace
You may wish to organize groups of controllers under a namespace. Most commonly, you might group a number of administrative controllers under an Admin:: namespace, and place these controllers under the app/controllers/admin directory. You can route to such a group by using a

```
namespace :admin do
  resources :articles, :comments
end
```

// Scope
If you want to route /articles (without the prefix /admin) to Admin::ArticlesController, you can specify the module with a scope block

```
scope module: 'admin' do
  resources :articles, :comments
end
```

- **Mount**

// If you would prefer to have your Rack application receive requests at the root path instead, use mount:

```
  mount AdminApp, at: '/admin'
```

// If you have an engine that you may want to route in your application like using sidekiq, you can use mount to connect them with your application

- **Route Concerns**

// to avoid repeating the same defined routes and making reusable routes for them.

- **Patch VS Put**

// PUT means replace the entire resource with given data (so null out fields if they are not provided in the request),
// PATCH means replace only specified fields. For the Table API, however, PUT and PATCH mean the same thing.

- **Partials**

// Partial templates - usually just called "partials" - are another device for breaking the rendering process into more manageable chunks. With a partial, you can move the code for rendering a particular piece of a response to its own file.

<%= render ‘menu’ %> will call the _menu.html.erb partial and puts the view where it is rendered

- **.touch**

// Touch is used to update the timestamps of the object, when it was updated at

- **Destroy vs Delete**

// Destroy removes the instances and also its associated children record and runs the callbacks(like before_destroy) on the other hand delete doesn't run any callbacks

- **Form_for Vs Form_with Vs Form_tag**

// form_for prefers the argument that you're passing in to be an active record object. This will easily make a create or edit form. We use form_for with a specific model and
.

- **Web Server Vs Application Server**

// The main difference between Web server and application server is that web server is meant to serve static pages e.g. HTML and CSS, while Application Server is responsible for generating dynamic content by executing server side code

- **/app vs /lib**

// You put all of your important business logics in the app folder which include your MVC and it is used directly by your application, So what you put in lib folder, you can put all your generic stuff that can be reusable maybe for you or maybe you can create a gem of it and others can use it as well

- **Avoiding Queries N+1**

//
@users = User.limit(50) // Controller // BAD //creates N+1 queries

@users = User.includes(:house).limit(50) // Controller // Good //

```
<% @users.each do |user|%>
   <%= user.house.address %>
<% end %>
```

- **Law of demeter**

// The law of demeter states that a model can only talk to the immediate associated models, which can be achieved by using delegate keyword and can tell the model after association which of the attributes can be used by the model

```
class Project > ActiveRecord::Base
   belongs_to :creator
   delegate :name, :company, :to => :creator, :prefix => true
end
```

- **Passing data from views to partials**

// Use locals keywords and pass data in the form of a hash.

- **after_initialize and after_find**

// The after_initialize callback will be called whenever an Active Record object is instantiated, either by directly using new or when a record is loaded from the database. It can be useful to avoid the need to directly override your Active Record initialize method.
// The after_find callback will be called whenever Active Record loads a record from the database. after_find is called before after_initialize if both are defined.
The after_initialize and after_find callbacks have no before_* counterparts, but they can be registered just like the other Active Record callbacks.

- **Active Record CallBacks**
  - **Creating an Object**
  1. before_validation
  2. after_validation
  3. before_save
  4. around_save
  5. before_create
  6. around_create

7. after_create
8. after_save
9. after_commit / after_rollback

- **Updating an Object**

1. before_validation
2. after_validation
3. before_save
4. around_save
5. before_update
6. around_update
7. after_update
8. after_save
9. after_commit / after_rollback

- **Destroying an Object**

1. before_destroy
2. around_destroy
3. after_destroy
4. after_commit / after_rollback

- **Running Callbacks**

  The following methods trigger callbacks:

  1. create
  2. create!
  3. destroy
  4. destroy!
  5. destroy_all
  6. destroy_by
  7. save
  8. save!
  9. save(validate: false)
  10. toggle!
  11. touch
  12. update_attribute
  13. update
  14. update!
  15. Valid?

The callbacks with the create ! operator throws exceptions when any error occurs on the other hand the create returns false if any error occurs

Additionally, the after_find callback is triggered by the following finder methods:

1. all
2. first
3. find
4. find_by
5. find_by_*
6. find_by_*!
7. find_by_sql
8. Last

- **Active Model Callbacks**

// extend ActiveModel::Callbacks
Is used to add callbacks functionality in active models and then to define the callbacks we use define_model_callbacks :create :save :initializer :update

And we can define after_create and before_create and execute a specific code for the callback

- **How to send data from views to layouts**

// You can use
 to specify where the view will be sending data and content_for :content will define what the actual view will be placed to layout where we can put the data

- **Resource Vs Resources**

// Resource is used when we are handling the request for a single resource

- **The Types of Associations**

// Rails supports six types of associations:
1. belongs_to
2. has_one
3. has_many
4. has_many :through
5. has_one :through
6. Has_and_belongs_to_many

- **Belongs_to**

// A belongs_to association sets up a connection with another model, such that each instance of the declaring model "belongs to" one instance of the other model. For example, if your application includes authors and books, and each book can be assigned to exactly one author, you'd declare the book model this way:

```
class Book < ApplicationRecord
  belongs_to :author
end

Class Author < ApplicationRecord
  has_many :books || has_one :book
end
```

- **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
```

- **Precision VS Scale**

// The precision represents the total number of digits in the number, whereas scale represents the number of digits following the decimal point. ... To specify the precision and scale, simply pass those as options to your column definition.

rails generate migration AddDetailsToProducts 'price:decimal{5,2}' supplier:references{polymorphic}

- **HTTP Request Structure**

// Http request contains a method (GET/POST)
// The path of the URL request.
// Version of the HTTP

```
before_validation :before_valid
  # after_validation :after_valid
  # before_save :before_sav
  # around_save :around_sav
  # before_create :before_creat
  # around_create :around_creat
  # after_create :after_creat
```

- **HTTP Headers**

// HTTP headers are included in the message to provide the recipient with the information about the message,

- **HTTP Message**

// The HTTP message contains the header and the content we're sending, It is in a hash format.

- **Action Cable**

// Action cable integrates the websockets with your rails application. Which means you can use websockets instead of HTTP response, request protocol. Websockets provide you with a full duplex TCP connection.

- **Include vs Extend**

// In simple words, the difference between include and extend is that 'include' is for adding methods only to an instance of a class and 'extend' is for adding methods to the class but not to its instance.

- **Associations Types**
    1. Has_one
    2. Has_many
    3. Has_one :through
    4. Has_many :through
    5. Belongs_to
    6. has_many_and_belongs_to

- **When to use has_many :through and has_many_and_belongs_to**

// The simplest rule of thumb is that you should set up a has_many :through relationship if you need to work with the relationship model as an independent entity. If you don't need to do anything with the relationship model, it may be simpler to set up a has_and_belongs_to_many relationship (though you'll need to remember to create the joining table in the database).
You should use has_many :through if you need validations, callbacks, or extra attributes on the join model

.

- **:dependent option**

// If you set the :dependent option to:

1. **:destroy**, when the object is destroyed, destroy will be called on its associated objects.
2. **:delete**, when the object is destroyed, all its associated objects will be deleted directly from the database without calling their destroy method.
3. **:destroy_async**: when the object is destroyed, an ActiveRecord::DestroyAssociationAsyncJob job is enqueued which will call destroy on its associated objects. Active Job must be set up for this to work.
4. **:restrict_with_exception:** Restrict with exception raises an exception when there is an associated record found.
5. **:restrict_with_error:** Adds an error message to the object if there is any associated record found.
6. **:nullify:** Nullify nulls out the foreign keys of the associated object.

- **Explicit Foreign Key**

// By convention, Rails assumes that the column used to hold the foreign key on this model is the name of the association with the suffix _id added. The :foreign_key option lets you set the name of the foreign key directly/explicitly:

```
class Book < ApplicationRecord
  belongs_to :author, class_name: "Patron",
                      foreign_key: "patron_id"
end
```

- **Through Associations**

// The through associations come in handy when we are creating a link between two classes and we can generate another model for their relation, just like patient and doctors have appointments in common and also have appointment dates of their own, but these are not required everywhere just like a relationship between an assembly and a part, many assemblies have many parts and many parts are in many assemblies, SO we don't use through their and we can use has_many_and_belongs_to

- **Scope in Active Records**

// scope :not_available, -> {where("available = ?", false)}

Scopes are used to define queries in the models,
Default Scopes, The default scopes are applied on all the queries that will be performed on the model default_scope

Named Scopes, The named scopes are defined using the scope keyword and they the name they will be called by
scope :available, ->{where("available = ?", true) }

We can also use scopes in the controller and pass the data to the views appropriately

- **Take**

// By using Take as model.take will get you the first record it founds without any implicit ordering and if you pass a value to it, that value will act as a limit on the number of records to be fetched

- **Unscoped**

// Unscoped is used to get the data independent of all the default scopes that are running on the data

- **Enums**

// enum status: [:shipped, :being_packed, :complete, :cancelled]

- **Important Retrievers**
    a. Annotate (Used to add comment to the Sql Query
    b. Find (Used to find the record against id)
    c. Create_with (Used to set the value for an attribute)
    d. Distinct (Used to get all the records distinct // No Repetitions)
    e. Eager_load (Used to force Left Outer Join on the args)
    f. Extending (
    g. extract_associated
    h. From (Specifies the table from which the records will be fetched)
    Doctor.select.from
    i. Group (Allows to specify a group attribute)
    j. Having (Allows to specify a Having clause)
    k. Includes (Used to get the data with the presence of given model)
    l. joins
    m. left_outer_joins
    n. Limit (Gives the limit of the records to be fetched)
    o. lock
    p. none
    q. offset
    r. optimizer_hints

s. order
t. preload
u. readonly
v. references
w. reorder
x. reselect
y. reverse_order
z. Select (Generates the sql Select, can pass the attributes you want to select from the table)
aa. Where (Generates the sql where clause, can pass the condition to perform on the records)

- **Select Vs Pluck Vs Map**

// Map/Collect
f = Author.all.select(:id).map(&:id)
Will return an array of Active records and then run a loop to get the ids.

// Pluck
f = Author.all.pluck(:id)
Will return the array of the attribute we just passed

- **Preload**

// Preload loads the data in two separate queries, For example if you're doing something like

Author.preload(:books).to_a
It will first go to the author and pick up all the primary keys and then run the second query on the books table and will find all the books with author_id in the array returned in the first query.

We cannot use the where clause on books, whereas it will work perfectly fine on the Author table

- **Includes**

// Includes also performs two queries just like pre loading but it is smarter than the preloading

- **Custom Validation**

// Custom validation classes can be created by inheriting the ActiveModel::Validator and these classes must implements the validate method

- **Assets 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.
The asset pipeline is implemented by the “sprockets-rails” gem, and is enabled by default. You can disable it while creating a new application by passing the --skip-sprockets option.

- **Fingerprinting**

// Fingerprinting is a technique that makes the name of a file dependent on the contents of the file. When the file contents change, the filename is also changed. For content that is static or infrequently changed, this provides an easy way to tell whether two versions of a file are identical, even across different servers or deployment dates.
The technique Sprockets uses for fingerprinting is to insert a hash of the content into the name, usually at the end. For example a CSS file global.css
Global-908e25f4bf641868d8683022a5b62f54.css

- **Precompilation**

// In precompilation, asset pipeline 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.

- **Contcatination**

// Concatenation is a concept to concatenate all the javascript files into one or all the css files to one, Assets pipeline gives us this first feature.

- **Minification**

// Sprockets provides us with the minification, in which the code is compressed or minified, Assets pipeline gives us this second feature

- **Preprocessing**

// After the minification of the files, they are precompiled and are put in the Public/Assets and are used from there to save the response time, the asset pipelines just help us breakdown all the js and css to their relevant folders for our understanding and then

- **Members VS Collections (Routes)**

//

- **Separate seed in multiple files**
- **When should indexes be avoided?**
    1. Indexes should not be used on small tables.
    2. Tables that have frequent, large batch updates or insert operations.
    3. Indexes should not be used on columns that contain a high number of NULL values.
    4. Columns that are frequently manipulated should not be indexed.

- **Pagination**

// Pagination is a process of separating print or digital media into discrete pages. Just like when we see page numbers in a bar on a website with some digital content