RUBY ON RAILS - NOTES

Ruby On Rails - Quick Fire 🔥

1. What is the difference between include and extend in Ruby?

include:

This is used to mix a module's methods as instance methods into a class. It makes the methods from the module available to all instances of the class.

```
module Greet
  def hello
    puts "Hello!"
  end
end

class Person
  include Greet
end

p = Person.new
p.hello  # => "Hello!"
```

extend:

This makes the module's methods available as class methods, not instance methods. The module's methods are mixed into the class as class methods, and not for the instances of the class.

```
module Greet
 def hello
   puts "Hello!"
```

```
  end
end

class Person
  extend Greet
end

Person.hello  # => "Hello!"
(class method)
```

So, the difference is:

- include - adds module methods as instance methods.
- extend - adds module methods as class methods.

2. What is the purpose of the before_action and after_action callbacks in Rails controllers? Can you give an example of how you would use them?

In Ruby on Rails, before_action and after_action are controller callbacks that allow you to run methods before or after an action is executed in the controller.

before_action: Runs a method before an action is executed. It's typically used to perform tasks like authentication, authorization, or preparing data before the controller action is run.

Example:

```
class ArticlesController < ApplicationController
  before_action :authenticate_user, only: [:edit, :update]

  def edit
    # action code
  end

  private
```

```
  def authenticate_user
    redirect_to login_path unless user_signed_in?
  end
end
```

In this example, authenticate_user will run before the edit or update actions to check if the user is signed in.

after_action: Runs a method after an action has been executed. It's commonly used for things like logging or modifying the response.

Example:

```
class ArticlesController < ApplicationController
  after_action :log_action, only: [:create, :destroy]

  def create
    # action code
  end

  def destroy
    # action code
  end

  private

  def log_action
    Rails.logger.info "Action was performed."
  end
end
```

Here, log_action will run after the create or destroy actions are executed, logging the event.

3. What are "strong parameters" 💪 in Rails, and why is it necessary? Can you give an example?

Strong Parameters is a Rails feature introduced in Rails 4 to prevent mass assignment vulnerabilities. It's used to explicitly whitelist which attributes of a model can be updated via user input. Without strong parameters, a user might send unwanted data in a form (like updating a user's role), which could lead to security issues.

Why is it necessary?

Before Rails 4, mass assignment allowed attackers to assign any attribute of a model, even sensitive ones like admin or is_active, if they were not protected. Strong Parameters ensures only the intended attributes are allowed for mass assignment.

Example:

Let's say you have a User model with attributes name, email, role, and is_admin.

Without strong parameters, if a user sends a form to update their data, they could potentially update role or is_admin (which should be protected).

```
class UsersController <
ApplicationController

  def update

    @user = User.find(params[:id])

    # This is unsafe!

    @user.update(params[:user])

  end

end
```

With strong parameters, we explicitly allow only the name and email to be updated:

```
class UsersController < ApplicationController
  def update
    @user = User.find(params[:id])
    # Using strong parameters to whitelist allowed fields
    if @user.update(user_params)
      redirect_to @user
    else
      render :edit
    end
  end

  private
  def user_params
    # Only permit name and email
    params.require(:user).permit(:name, :email)
  end
end
```

In this case, even if the form includes role or is_admin, they won't be updated because they are not permitted in user_params.

4. What is the difference between find and find_by in Rails? When would you use each? 🔍

find:

- Purpose: The find method is used to retrieve a record by its primary key (usually id).
- Behavior: It raises an ActiveRecord::RecordNotFound exception if the record with the given ID is not found.

Example:

Fetch a record by its primary key (id)

```
user = User.find(1)  # Finds the user with id = 1
```

If no user with id = 1 exists, it will raise:

ActiveRecord::RecordNotFound

find_by:

- Purpose: The find_by method is used to find a record by any column (not just id), and you can specify any attribute as the search criterion.
- Behavior: It returns nil if no record is found (instead of raising an exception like find does).

Example:

Fetch a record by any attribute (e.g., email)

```
user = User.find_by(email:
'john.doe@example.com')
```

If no user with the email 'john.doe@example.com' exists, it will return nil, and not raise an exception.

Key Differences:

1. Primary Key vs Other Attributes:
    - find: Can only use the primary key (id) to find a record.
    - find_by: Can use any column/attribute to find a record.

2. Exception Handling:
    - find: Raises ActiveRecord::RecordNotFound if no record is found.
    - find_by: Returns nil if no record is found.

When to use each:

- Use find when you are sure you are looking for a record by its primary key (id) and want to handle the case where the record isn't found through exception handling.
- Use find_by when you are searching by non-primary key columns, or when you prefer to get nil if no match is found instead of raising an exception.

Example Scenario:

If you want to find a user by id, use find:

```
user = User.find(5)  # Finds the user with id = 5
```

If you want to find a user by their email, use find_by:

```
user = User.find_by(email: 'jane.doe@example.com')  # Finds the
first user with that email.
```

5. What is the role of migrations in Rails? How do you create and run a migration? Can you give an example of a migration to add a new column to a table?

What is a migration in Rails?

In Rails, migrations are a way to alter the database schema over time in a consistent and version-controlled manner. They allow you to make changes to

your database schema, such as creating or modifying tables, adding or removing columns, creating indexes, and more.

Migrations are part of the ActiveRecord framework and ensure that the changes to the database are consistent across different environments (development, testing, production). They also provide a way to rollback or revert changes, which is important for maintaining a flexible and reliable development process.

How do you create and run a migration?

1. Creating a Migration

To create a migration, you use the Rails command:

rails generate migration MigrationName

This generates a migration file in the db/migrate directory. The migration file will have a timestamp in the filename, which helps keep track of the order in which migrations are created.

For example:

```
rails generate migration AddAgeToUsers age:integer
```

This will generate a migration file like:

```
class AddAgeToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :age, :integer
  end
end
```

- AddAgeToUsers: This is the name of the migration.
- age:integer: This adds a new column age of type integer to the users table.

2. Running the Migration

To apply the migration (i.e., actually make the change to the database), you run the following command:

```
rails db:migrate
```

This will execute all the pending migrations in the db/migrate directory, updating the database schema as defined in those migrations.

3. Rolling Back a Migration

If you want to undo the last migration (or any specific migration), you can run:

```
rails db:rollback
```

This will undo the last migration that was applied. If you want to roll back multiple steps, you can use the STEP option:

```
rails db:rollback STEP=3  # Rolls back the
last 3 migrations
```

Example of a Migration to Add a Column

Let's say you want to add a column called age to the users table:

Generate the migration:

rails generate migration AddAgeToUsers age:integer

1. Migration File Generated:

```
class AddAgeToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :age, :integer
  end
end
```

Run the Migration:

```
rails db:migrate
```

2. This will add the age column (of type integer) to the users table in your database.

Summary of the Migration Process:

1. Create a migration file with rails generate migration.
2. Write the changes to be made (e.g., adding columns, creating tables).
3. Run the migration with rails db:migrate to apply the changes to the database.
4. Rollback migrations with rails db:rollback if needed.

Migrations: Rails migrations handle schema changes over time. Each migration modifies the database schema by adding/removing tables, columns, indexes, and so on.

Versioning: Rails keeps track of migrations in the schema_migrations table (a table in your database). This table stores which migrations have been applied, ensuring that migrations are applied in the correct order.

You can see the list of migrations that have been run in the schema_migrations table:

```
SELECT * FROM schema_migrations;
```

When you run rails db:migrate, Rails looks at the schema_migrations table to determine which migrations still need to be applied.

6. What are Rails validations, and how do you use them in models? Can you provide an example of a validation in a model?

Rails Validations:

In Rails, validations are used to ensure that the data entered into your models is correct and meets certain criteria before it gets saved to the database. Validations are defined inside the model and are typically used to check conditions like required fields, format, uniqueness, numericality, and more.

How to Use Validations:

You define validations within your model class using built-in Rails methods. For example, you can use validates for a variety of conditions.

Commonly Used Validations:

Presence Validation: Ensures that a field is not empty or nil.

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

This means that the name field must not be empty before the record can be saved.

Uniqueness Validation: Ensures that a field's value is unique across the table.

```
class User < ApplicationRecord
  validates :email, uniqueness: true
end
```

This ensures that no two users can have the same email.

Format Validation: Ensures that a field's value matches a specific format, often used with regular expressions.

```
class User < ApplicationRecord
  validates :email, format: { with:
URI::MailTo::EMAIL_REGEXP }
end
```

This validates that the email field contains a valid email address format.

Length Validation: Ensures that a string's length is within a specified range.

```
class User < ApplicationRecord
  validates :password, length: { minimum: 8 }
end
```

1. This validates that the password field is at least 8 characters long.

Numericality Validation: Ensures that a field is a number.

```
class Product < ApplicationRecord
  validates :price, numericality: true
end
```

2. This ensures that the price field contains a numerical value.

Inclusion Validation: Ensures that a field's value is within a specified set of values.

```
class User < ApplicationRecord
  validates :status, inclusion: { in: ['active', 'inactive',
'suspended'] }
end
```

3. This validates that the status is one of the allowed values.

Example of a Model with Validations:

```
class User < ApplicationRecord
  # Validates that the name is not empty
  validates :name, presence: true

  # Validates that the email is unique
  validates :email, presence: true, uniqueness: true

  # Validates that the email is in a proper format
```

```
  validates :email, format: { with: URI::MailTo::EMAIL_REGEXP
}

  # Validates that the password is at least 8 characters long
  validates :password, length: { minimum: 8 }

  # Ensures the age is a number
  validates :age, numericality: true
end
```

How Validations Work:

When you attempt to save or update a model with invalid data (e.g., a missing name or a wrongly formatted email), Rails will prevent the record from being saved to the database. You can check whether the model is valid using the .valid? method, and you can get the errors using .errors.

Example:

```
user = User.new(email: 'invalid_email', password: '123', age:
'abc')
user.valid?  # => false
user.errors.full_messages  # => ["Email is invalid", "Password is
too short", "Age is not a number"]

  class CustomerParams
    include ActiveModel::Validations
    attr_accessor :name, :pass, :email_id

    validates :name, presence: true
    validates :pass, presence: true
    validates :email_id, presence: true

    def initialize params
      @name = params['name']
      @pass = params['pass']
      @email_id = params['email_id']
    end
  end

validate_request = CustomerParams.new(request)
```

```
raise "Invalid request. Reason:
#{validate_request.reason_for_failure}" unless
validate_request.valid?
```

7. What is the difference between has_many and belongs_to associations in Rails? Can you provide an example of each?

Associations in Rails:

Rails uses ActiveRecord associations to establish relationships between different models. These associations make it easy to set up relationships between database tables and work with related data.

has_many Association:

- Definition: A has_many association is used to set up a one-to-many relationship, where one model can have many related records.
- Example: A User might have many Posts. This means a single user can create multiple posts, but each post belongs to one user.

Example:

```
# user.rb (model)
class User < ApplicationRecord
  has_many :posts
end

# post.rb (model)
class Post < ApplicationRecord
  belongs_to :user
end
```

- In this case, the User model has a has_many relationship with the Post model.

- Each Post will have a foreign key (user_id) that links it back to the User who created it.

belongs_to Association:

- Definition: A belongs_to association is used to set up the inverse of a has_many relationship. It's used on the model that belongs to the other model in the relationship. It indicates that a record of this model is owned by another model.

Example:

In the example above, the Post model belongs to the User because each post is associated with a single user.

```
# post.rb (model)
class Post <
ApplicationRecord
  belongs_to :user
end
```

Summary of the Relationship:

1. has_many:
    - Used in the parent model (the one that "owns" the records).
    - Specifies that a single record can have many associated records.
    - Example: A User has many Posts.
2. belongs_to:
    - Used in the child model (the one that "belongs" to another record).
    - Specifies that a record is associated with a single record from another model.
    - Example: A Post belongs to a User.

How the Database Looks:

- In the posts table, there will be a user_id column, which is a foreign key pointing to the users table. This column tells Rails which user each post belongs to.

Schema Example:

```
# users table
create_table :users do |t|
  t.string :name
  t.timestamps
end

# posts table
create_table :posts do |t|
  t.string :title
  t.text :body
  t.references :user, foreign_key: true   # This is the 'user_id'
column
  t.timestamps
end
```

Example of Usage in Rails:

- To create a new post for a user:

```
user = User.find(1)
post = user.posts.create(title: "My First Post", body: "This is the
body of the post")
```

- To access a user's posts:

```
user = User.find(1)
user.posts  # Returns an array of posts belonging to that user
```

- To get the user of a post:

```
post = Post.find(1)
post.user  # Returns the user who created this post
```

8. What is the purpose of the application.rb file in a Rails project, and what are some key configurations you can find in it?

The application.rb file in a Rails project is a central configuration file for your entire application. It's located in the config directory (config/application.rb), and it plays a crucial role in setting up and configuring various aspects of the Rails framework.

Purpose of application.rb:

- The application.rb file is responsible for initializing the Rails application and loading necessary configuration settings.
- It helps define the overall behavior of your application, including things like:
    - Loading middleware
    - Specifying default settings for various components of the Rails framework
    - Setting configuration for environments (e.g., development, production)
    - Including gems and modules

Key Components in application.rb

Here's what you typically find in the config/application.rb file:

1. Class Definition

The class Application inherits from Rails::Application, which means it's the main class that Rails uses to load your application's configuration.

```
module MyApp
  class Application < Rails::Application
    # Configuration for the application
  end
end
```

2. Configuration Block

Inside the Application class, you'll find various configurations for the app. You can modify these to change how your application behaves.

Example:

```
module MyApp
  class Application < Rails::Application
    # Specifies the default locale
    config.i18n.default_locale = :en

    # Load custom directories to be included during asset compilation
    config.assets.paths << Rails.root.join('app', 'assets', 'fonts')

    # Set timezone for the application
    config.time_zone = 'UTC'

    # Enable or disable caching
    config.cache_store = :memory_store

    # Add additional autoload paths for custom directories
    config.autoload_paths += %W(#{config.root}/lib)

    # Use default logging format
    config.log_level = :debug
  end
end
```

3. Configuration Options You Can Modify:
config.i18n.default_locale: Set the default language or locale for your application.

Example:

```
config.i18n.default_locale = :fr  # Default language is French
```

config.time_zone: Set the default time zone for your application.

Example:

```
config.time_zone = 'Eastern Time (US & Canada)'
```

config.active_record.default_timezone: Sets the timezone for ActiveRecord timestamps (e.g., storing in UTC vs local time).

Example:

```
config.active_record.default_timezone = :utc
```

- config.assets.paths: Customize asset loading paths, for example, to include fonts or images stored outside of the default app/assets directory.
- config.cache_store: Define which cache store Rails should use. Options include :memory_store, :file_store, and more.
- config.middleware: You can add, remove, or modify middleware used by your Rails app (Rails uses middleware to process HTTP requests and responses).

4. Autoloading Paths:

Rails automatically loads files in specific directories, like app/models and app/controllers, but you can add other directories to be autoloaded.

Example:

```
config.autoload_paths += %W(#{config.root}/lib)
```

This adds the lib directory to the list of directories Rails will autoload classes from.

Important Settings in application.rb:

- Environments: Some settings are set by default for specific environments (e.g., config/environments/development.rb, config/environments/production.rb). The settings in application.rb serve as the base, and those environment-specific files can override them.

Middleware Configuration: You can add custom middleware or reorder default middleware.

Example:

```
config.middleware.use Rack::Attack  # Example of using custom
middleware
```

When to Modify application.rb:

- Modify this file when you want to change global application settings that affect the entire app, regardless of environment.
- It's commonly modified for settings like locale, time zone, autoload paths, and caching.

Example of Custom Configuration:

Let's say you want to use a different default time zone and locale for your application:

```
module MyApp
  class Application < Rails::Application
    # Set the default time zone to 'Pacific Time'
    config.time_zone = 'Pacific Time (US & Canada)'

    # Set the default locale to Spanish
    config.i18n.default_locale = :es
  end
end
```

Conclusion:

The config/application.rb file is essential for configuring how your Rails application behaves on a global level. It's where you can set various global application settings such as time zone, locale, and middleware.

9. What are "scopes" in Rails, and how do you define and use them in models? Can you provide an example of a scope?

In Rails, scopes are a way to define reusable queries that can be used across your application. They are essentially custom query methods that allow you to define common query logic and reuse it in a clean and concise way. Scopes help you avoid repeating the same query logic in multiple places, making your code more maintainable and readable.

How to Define a Scope

You define scopes inside your model using the scope method. A scope is essentially a class method that returns an ActiveRecord relation (which can be chained with other queries or scopes).

Syntax of a Scope:

```
scope :scope_name, -> { query_logic }
```

- scope_name: The name of the scope (the method you will call on the model).
- query_logic: The logic of the query (e.g., filtering records based on a condition).

Examples of Scopes:

1. Simple Scope Example:

Let's say you have a User model and you want to define a scope that finds all active users:

```
class User < ApplicationRecord
  # Define a scope for active users
  scope :active, -> { where(active: true) }
end
```

- Now, you can call User.active to retrieve all active users:

```
active_users = User.active
```

2. Scope with Parameters:

Scopes can also take parameters to make them more dynamic.

For example, let's say you want a scope that finds users who are active and have a specific role:

```
class User < ApplicationRecord
  # Define a scope with a parameter (e.g., finding active users with a specific role)
  scope :active_with_role, ->(role) { where(active: true, role: role) }
end
```

Now, you can call the scope with a parameter like this:

```
admin_users = User.active_with_role('admin')
```

3. Chaining Scopes:

One of the key benefits of scopes is that they can be chained together to build more complex queries.

For example, if you have the following scopes:

```
class User < ApplicationRecord
  scope :active, -> { where(active: true) }
  scope :with_role, ->(role) { where(role: role) }
end
```

You can chain them like this:

```
admin_active_users = User.active.with_role('admin')
```

This will generate a query like:

```
SELECT * FROM users WHERE active = true AND role = 'admin'
```

4. Scope with order and limit:

You can define scopes to include sorting and limiting:

```
class User < ApplicationRecord
  # Scope to get the most recent users
  scope :recent, -> { order(created_at: :desc).limit(5) }
end
```

This scope will return the 5 most recent users:

```
recent_users = User.recent
```

Why Use Scopes?

1. Cleaner Code: Instead of repeating the same query logic multiple times, you can define it once as a scope and reuse it.
2. Chainable: Scopes return ActiveRecord relations, which means you can chain them together to build complex queries.
3. Readability: Scopes make your queries more readable and self-documenting. For instance, User.active is much clearer than writing User.where(active: true) everywhere.

Example Use Case:

Let's say you have a Post model and want to define scopes for published posts and posts created within the last week:

```
class Post < ApplicationRecord
  # Scope to get published posts
  scope :published, -> { where(published: true) }
```

```
  # Scope to get posts created in the last 7 days
  scope :recent, -> { where("created_at >= ?", 7.days.ago) }
end
```

Now you can chain these scopes to get published posts created in the last week:

```
recent_published_posts = Post.published.recent
```

Conclusion:

- Scopes are a powerful way to define reusable query logic in your Rails models.
- They help keep your code DRY (Don't Repeat Yourself) and make it more readable.
- You can define simple or complex queries, take parameters, and chain scopes to create powerful query logic.

10. What is the difference between pluck and select in Rails, and when would you use one over the other?

pluck vs. select in Rails

Both pluck and select are used to query data from the database, but they behave differently in terms of the results they return and how they interact with the database.

1. pluck Method

- Purpose: pluck is used to retrieve a specific column (or columns) directly from the database. It bypasses ActiveRecord objects and returns an array of values.
- When to Use: Use pluck when you only need certain columns and don't need the entire ActiveRecord object with all its methods and associations.

Example:

Suppose we have a User model with columns: id, name, and email

```
names = User.pluck(:name)
```

This will return an array of names: ["Alice", "Bob", "Charlie"]

Plucking multiple columns

```
names_and_emails = User.pluck(:name, :email)
```

This will return an array of arrays: [["Alice", "alice@example.com"], ["Bob", "bob@example.com"]]

- Performance: pluck is more efficient when you only need certain columns from the database, as it retrieves just the necessary data without creating full ActiveRecord objects. This can be especially useful for large datasets.

2. select Method

- Purpose: select is used to filter or choose specific columns to be included in the ActiveRecord query result. The result is still an ActiveRecord relation, meaning it returns ActiveRecord objects, not just raw values.
- When to Use: Use select when you want to work with ActiveRecord objects but only want specific fields (columns) to be loaded into memory.

Example:
Suppose we have a User model with columns: id, name, and email

```
users = User.select(:name, :email)
```

This returns an ActiveRecord relation with User objects that only include the 'name' and 'email' attributes.

- Performance: select allows you to limit the columns that are loaded into the ActiveRecord objects, but it still creates objects and includes the overhead of ActiveRecord methods and callbacks.

Key Differences:

Feature	pluck	select
Return Value	Array of values (not ActiveRecord objects)	ActiveRecord relation with objects
Use Case	When you need specific column(s) without the overhead of ActiveRecord objects	When you need ActiveRecord objects but want to limit the fields returned
Performance	Faster for large datasets since it avoids creating ActiveRecord objects	Slower than pluck because it still creates ActiveRecord objects and applies model-level methods
Example	User.pluck(:name)	User.select(:name, :email)

Example Comparison:

Let's say we have a User model with columns id, name, email, and created_at.

Using pluck (returns an array of values):

```
names = User.pluck(:name)
```

Result: ["Alice", "Bob", "Charlie"]

1. Using select (returns ActiveRecord objects):

```
users = User.select(:name, :email)

users.each do |user|
  puts user.name # Accessing ActiveRecord attributes
end
```

Result: ActiveRecord objects with only 'name' and 'email' columns loaded

2. When to Use Which?

- Use pluck when you just need data values from the database (e.g., for a list or for calculations) and don't need ActiveRecord methods or functionality like associations.
- Use select when you need to work with ActiveRecord objects but want to limit the number of columns loaded into memory for performance reasons.

11. Can you explain the purpose of validates_uniqueness_of and how it works in Rails? How does it differ from the validates method with uniqueness: true?

validates_uniqueness_of vs validates :attribute, uniqueness: true

In Rails, both validates_uniqueness_of and validates :attribute, uniqueness: true are used to ensure that the value of an attribute in a model is unique within the database, but there are some differences in how they are used.

1. validates_uniqueness_of

validates_uniqueness_of is a class method that can be used to validate the uniqueness of an attribute. This method is older and was often used in older versions of Rails. It is no longer commonly used in favor of the more modern syntax with validates.

Example:

```
class User < ApplicationRecord
  validates_uniqueness_of :email
end
```

This ensures that the email attribute is unique across all User records.

2. validates :attribute, uniqueness: true

This is the more modern and preferred syntax introduced in Rails 3.1 and onwards. It is a more concise way of specifying validations. It uses the validates method with uniqueness: true as an option.

Example:

```
class User < ApplicationRecord
  validates :email, uniqueness: true
end
```

This also ensures that the email is unique in the User model.

Key Differences:

- Syntax: The main difference is in how the validation is specified. validates_uniqueness_of is a class method, whereas validates with uniqueness: true is more flexible and is now the preferred way to add validations.

- Configuration Options: validates :attribute, uniqueness: true has more built-in options for customization and is more consistent with the overall Rails validates syntax.

- Deprecation: validates_uniqueness_of is considered older syntax and may not be as well-supported in future Rails versions, whereas the validates :attribute, uniqueness: true style is the recommended approach.

Important Notes:

- Both methods do not guarantee that the value is unique in the database. They only check the model's values at the application level.
- Race Conditions: If two requests are trying to create records with the same value at the same time, these validations won't catch it. To avoid race conditions, it's important to have a unique constraint in the database (via migrations) to ensure uniqueness at the database level as well.

Example with a database constraint:

You would also typically add a unique constraint on the column in your database to ensure uniqueness at that level:

```
add_index :users, :email, unique: true
```

This ensures that the email column in the users table has a unique constraint.

Summary:

- validates_uniqueness_of is the older syntax, and validates :attribute, uniqueness: true is the modern, recommended syntax.
- Use the modern validates :attribute, uniqueness: true to ensure uniqueness in your Rails models, and make sure to also enforce uniqueness at the database level for integrity.

12. DB Locking in Rails 🔒

Locking in Rails:

In Rails, locking mechanisms help prevent race conditions when two or more processes attempt to modify the same record in the database simultaneously. Rails provides a couple of ways to handle locking: Optimistic Locking and Pessimistic Locking.

Let's discuss both types of locking, with examples, and then explain what the default behavior is.

---

1. Optimistic Locking

Optimistic Locking assumes that most operations will not result in conflicts. It doesn't lock the record when reading but checks whether the record has been modified by another process before saving it.

In Optimistic Locking, you need a lock_version column on the model that will automatically track changes to the record. This column is updated automatically each time a record is updated.

Steps to implement Optimistic Locking:

1. Migration to add lock_version:
    - This column is usually an integer and is used to track the version of the record.

```
class AddLockVersionToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :lock_version, :integer, default: 0, null:
false
  end
end
```

2. Model:
    - Rails automatically handles the lock_version column. You don't need to manually change it — Rails does it when you update a record.

```
class User < ApplicationRecord
  # No special code needed for optimistic locking, just add the
lock_version column
end
```

3. Using Optimistic Locking:
    - When you try to save a record, Rails will check if the lock_version has changed. If it has, a ActiveRecord::StaleObjectError will be raised.

Example:

Fetching a user

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

Simulating a scenario where another user also modifies the same record
Another process updates the user before we save

Trying to update the user:

```
begin
  user.update!(name: 'New Name')
rescue ActiveRecord::StaleObjectError => e
  puts "Conflict detected: #{e.message}"
end
```

- What Happens: If another transaction has updated the record between the time we fetched the user and the time we tried to save it, an ActiveRecord::StaleObjectError will be raised.

---

2. Pessimistic Locking

Pessimistic Locking locks the record for exclusive use by a transaction. While the record is locked, other processes or transactions cannot modify or read it until the transaction is completed. This ensures that no other process can interfere with the changes you're making.

Pessimistic locking is particularly useful when you expect conflicts to happen frequently, and you want to prevent other transactions from working with the same data.

Using Pessimistic Locking:

Rails provides a lock method to perform pessimistic locking on a record. It uses SQL SELECT FOR UPDATE to lock the row.

Example:

Using pessimistic locking to lock a specific user record

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

This record is locked for the duration of the transaction.
Other transactions cannot modify this record until the lock is released.

Update the user safely

```
user.update!(name: 'New Locked Name')
```

What Happens: The user record with ID 1 is locked until the transaction is complete, preventing any other transactions from modifying or accessing it.

SQL Equivalent: This would translate to a query like:

```
SELECT * FROM users WHERE id = 1 FOR UPDATE;
```

Default Locking Behavior in Rails

By default, Rails does not apply any locking when you query records. This means that:

- When you read records (e.g., User.find(1)), they are not locked.
- Other processes can update the same record simultaneously, leading to potential conflicts.

However, optimistic locking is a feature you can enable by adding the lock_version column and relying on the ActiveRecord::StaleObjectError to handle conflicts.

Default behavior:

- No Locking (except for transactions). If you're not using includes, eager_load, or lock, Rails will not lock records by default. Multiple processes can read and modify the same record concurrently.

Transaction:

ActiveRecord::Base.transaction do

Perform some database operations

```
user = User.create!(name: "Alice")

post = Post.create!(title: "First Post", user_id: user.id)
```

If any exception is raised, the entire transaction will be rolled back

```
raise ActiveRecord::Rollback if some_condition
```

If no exceptions occur, the changes will be committed to the database

end

Summary of Locking in Rails:

Locking Type	Description	Default Behavior
Optimistic Locking	Uses a lock_version column to track changes and raise errors if concurrent changes are detected.	Not enabled by default; needs lock_version column.

Pessimistic Locking	Locks records using SELECT FOR UPDATE to prevent concurrent modifications.	Not enabled by default; needs .lock in query.
Default	No locking by default; records can be read and modified concurrently.	No locking unless explicitly specified.

Example Use Case:

Imagine you're building a financial application where multiple users are updating their balances. Pessimistic Locking would be useful to ensure that no two users can update the balance at the same time.

On the other hand, in a collaborative document editor, Optimistic Locking could be used to handle changes in a way where users can save their progress, and if someone else has made changes in the meantime, they'll get an error and can decide how to handle it.

---

13. What is the difference between includes, eager_load, and joins in Rails ActiveRecord? When would you use each one? 🔽

1. includes (Eager Loading)

- Purpose: includes is used to eagerly load associated records to avoid N+1 query problems (where an additional query is made for each record in a collection). It loads the associated records in separate queries.

- How it works: Rails will issue multiple queries, but it ensures that the associated records are loaded in one extra query instead of running additional queries for each associated record. It's particularly useful when

you need to access associations (like has_many or belongs_to) and want to load them up-front to avoid unnecessary database calls.

- Use Case: Use includes when you know you'll need the associated records and want to avoid N+1 queries but don't need to perform complex conditions on the associated tables.

Example:

Fetching users and their posts, avoiding N+1 queries

```
users = User.includes(:posts).where(active: true)
users.each do |user|
  puts user.posts.count
end
```

In this example, Rails will execute two queries:

1. One for users where active = true.
2. One for posts related to the users.

2. eager_load (Full Eager Loading)

- Purpose: eager_load is a form of eager loading that performs LEFT OUTER JOINs between the main model and its associations in a single query, fetching all the data at once.

- How it works: Instead of issuing separate queries like includes, eager_load loads the records in a single SQL query using JOIN statements. This can be more efficient if you need to filter or sort based on the associated table's fields.

- Use Case: Use eager_load when you need to join the associated tables (to filter, order, or select based on fields from the associated table) and want all the data in one query.

Example:

Fetching users and their posts with a join (using eager_load)

```
users = User.eager_load(:posts).where('posts.created_at > ?',
1.month.ago)
users.each do |user|
  puts user.posts.count
end
```

Here, Rails will perform a single query with a JOIN that combines users and posts.

3. joins (Inner Join)

- Purpose: joins performs an INNER JOIN between the main model and its associated models. This means that it only fetches records that have a matching record in the associated table (i.e., a record must exist in the associated table for the main record to be included).
- How it works: It doesn't fetch the associated data like includes or eager_load; instead, it just combines tables based on a SQL JOIN. You can filter based on fields from the associated table, but it doesn't load the associated records into the main model.
- Use Case: Use joins when you want to filter or query the main model using conditions on the associated table (without needing to load the actual data from the associated table).

Example:

Fetching users who have posts created within the last month

```
users = User.joins(:posts).where('posts.created_at > ?',
1.month.ago)
```

This query will join the users table and the posts table based on the foreign key relationship, and it will return only the users who have posts created within the last month. It doesn't load the posts records into the users objects; it just uses the JOIN to filter users.

When to Use Each One:

- includes: Use this when you need to load the associated records, and you want to avoid N+1 queries. It will load the records in separate queries.
    - Example: Displaying a list of users with their posts.
- eager_load: Use this when you want to eagerly load associations and need to filter or order based on fields from the associated table, requiring a single query with a JOIN.
    - Example: Displaying users with their posts, and filtering them by the post's created_at date.
- joins: Use this when you only need to filter or query based on the associated model's fields, without needing to load the associated records.
    - Example: Finding users who have posts created after a certain date, but not loading the posts into the users.

Example Comparison:

Let's say you want to display all users who have at least one post with the title "New Post".

With includes:

```
users = User.includes(:posts).where(posts: { title: 'New Post' })
```

- This will execute two queries: one for users and one for posts.

With eager_load:

```
users = User.eager_load(:posts).where(posts: { title: 'New Post' })
```

- This will execute one query with a LEFT OUTER JOIN between users and posts.

With joins:

```
users = User.joins(:posts).where(posts: { title: 'New Post' })
```

- This will execute one query with an INNER JOIN between users and posts, and only users with at least one post titled "New Post" will be returned.

Summary:

Method	Description	Use Case
includes	Eager loading with separate queries to avoid N+1 query problem.	When you need to load associated records without filtering or joining.
eager_load	Eager loading with LEFT OUTER JOIN, all data in one query.	When you need to filter or sort by associated model's fields.
joins	Performs an INNER JOIN between tables.	When you want to filter or query based on associated table's fields only.

Preload:

Example Scenario:

Consider an e-commerce application where we have the following two models:

- Order: An order placed by a customer.
- Customer: The customer who placed the order.

Models and Associations:

```
Order Model:
class Order < ApplicationRecord
  belongs_to :customer
  has_many :order_items
end


Customer Model:
class Customer < ApplicationRecord
  has_many :orders
end


OrderItem Model:
class OrderItem < ApplicationRecord
  belongs_to :order
end
```

Problem: N+1 Query Issue

Let's say we want to fetch all the orders and for each order, we want to show the customer who placed it and all order_items associated with the order.

If you fetch orders and access the customer and order_items for each order without preloading these associations, you'll end up with N+1 queries:

- 1 query to fetch all orders.
- N additional queries to fetch the customer for each order (1 per order).
- N additional queries to fetch the order_items for each order (1 per order).

This leads to many unnecessary database queries and performance issues. To avoid this, you can use preload.

Solution: Using preload

You can use preload to efficiently load the associated customer and order_items in separate queries, reducing the total number of queries and preventing the N+1 query problem.

Here's how to do it:

```
# Preload customers and order_items for all orders
orders = Order.preload(:customer, :order_items)

# Accessing orders and their associations without causing
additional queries
orders.each do |order|
  puts "Order ID: #{order.id}"
  puts "Customer Name: #{order.customer.name}"  # No extra query
for customer
  order.order_items.each do |item|
    puts "Order Item ID: #{item.id}"  # No extra query for order
items
  end
end
```

What Happens Behind the Scenes:

1. preload(:customer, :order_items):
    - This tells Rails to load the customer and order_items associations in separate queries rather than using a JOIN.
    - Rails will run:
        - One query to load all the orders.
        - One query to load all the customers associated with those orders.
        - One query to load all the order_items associated with those orders.
2. Reduced Queries:
    - Instead of running N+1 queries (where N is the number of orders), you'll have:
        - 1 query to fetch all orders.
        - 1 query to fetch all customers related to those orders.
        - 1 query to fetch all order items related to those orders.
3. No Extra Queries During Access:
    - When you access the customer or order_items for each order in the loop, Rails won't perform additional queries because the data has already been preloaded.

SQL Queries Generated:

Let's break down what SQL queries are generated when you use preload:

Query 1 (fetching all orders):

```
SELECT "orders".* FROM "orders";
```

Query 2 (fetching all customers for the orders):

```
SELECT "customers".* FROM "customers" WHERE "customers"."id" IN
(...);
```

Query 3 (fetching all order items for the orders):

```
SELECT "order_items".* FROM "order_items" WHERE
"order_items"."order_id" IN (...);
```

Key Benefits of Using preload:

- Avoids N+1 Queries: Instead of running a separate query for each associated record, preload reduces the number of queries to just three (one for each association), no matter how many orders you have.
- Improves Performance: By reducing the number of database queries, the overall performance of the application improves, especially when dealing with large datasets.
- Separation of Queries: Since preload uses separate queries (unlike eager_load), it doesn't use SQL JOINs. This can be more efficient when you have large tables or don't need to filter or aggregate data based on the associated records.

Example with N+1 Queries (Without preload):

```
# Without preload (leads to N+1 queries)
orders = Order.all

orders.each do |order|
  puts "Order ID: #{order.id}"
```

```
  puts "Customer Name: #{order.customer.name}"  # This triggers a
query for each order
  order.order_items.each do |item|
    puts "Order Item ID: #{item.id}"  # This triggers a query for
each order
  end
end
```

Here, for every order, the application will make an additional query to fetch the associated customer and order_items, leading to a total of N+1 queries (1 for the orders and N additional queries for each associated record).

Summary:

- preload helps to load associations in separate queries (rather than JOINs), thus avoiding N+1 query problems.
- It performs one query for the primary model (orders), one for each of the preloaded associations (customers and order items).
- This approach improves performance by reducing the total number of queries and making data retrieval more efficient when associations are needed.

Both preload and includes in Rails are used to handle eager loading of associations, but they work a bit differently and have different use cases. Let's break down the differences between them.

preload vs includes

1. Basic Purpose:

Both preload and includes are used to eager load associations to avoid N+1 query problems, but they differ in how they handle the loading of data.

- preload: Forces Rails to always load associations in separate queries.
- includes: Rails tries to determine whether to load the associations using a JOIN (single query) or using separate queries based on the context and the query being built.

2. SQL Queries:

- preload: Always uses separate queries for the associations.

- includes: Rails will decide whether to use a JOIN or separate queries based on the context of the query and the conditions involved.

3. How Rails Decides (when using includes):

When you use includes, Rails uses different strategies depending on whether you're filtering, ordering, or selecting fields from the included associations.

- When includes leads to a JOIN: If you include conditions or filters on the included association (like where or order), Rails will switch to using a JOIN because it needs to filter or order based on the joined data.
- When includes leads to separate queries: If no conditions are applied to the included associations (just fetching them), Rails will run separate queries, much like preload.

4. Performance Considerations:

- preload: When you are confident that you only want to load associations using separate queries (without any filtering or joining on them), use preload. It guarantees no JOINs are used.
- includes: If you want to sometimes load associations with JOINs (when applying conditions to them), includes is more flexible. It may optimize performance by using JOINs where necessary but can still fall back to separate queries when it doesn't need to join.

Example 1: Using preload

```
# Using preload, it will load associations in separate queries
orders = Order.preload(:customer, :order_items)

orders.each do |order|
  puts "Order ID: #{order.id}"
  puts "Customer Name: #{order.customer.name}"  # No extra query
for customer
  order.order_items.each do |item|
    puts "Order Item ID: #{item.id}"  # No extra query for order
items
  end
end
```

- This will always generate 3 SQL queries:
    - 1 for orders.
    - 1 for customers.
    - 1 for order items.

No JOIN is used here.

Example 2: Using includes (Simple Case)

Using includes, it will load associations using separate queries if no filtering is done

```
orders = Order.includes(:customer, :order_items)

orders.each do |order|
  puts "Order ID: #{order.id}"
  puts "Customer Name: #{order.customer.name}"  # No extra query
for customer
  order.order_items.each do |item|
    puts "Order Item ID: #{item.id}"  # No extra query for order
items
  end
end
```

- This will also generate 3 SQL queries (separate queries):
    - 1 for orders.
    - 1 for customers.
    - 1 for order items.

No JOIN is used in this case because no filtering, ordering, or conditions were applied.

Example 3: Using includes with Filtering

Using includes with conditions (e.g., filtering orders)

```
orders = Order.includes(:customer, :order_items).where(customer: {
status: 'active' })

orders.each do |order|
  puts "Order ID: #{order.id}"
  puts "Customer Name: #{order.customer.name}"  # No extra query
for customer
  order.order_items.each do |item|
    puts "Order Item ID: #{item.id}"  # No extra query for order
items
  end
```

```
end
```

- This time, Rails will likely use a JOIN because you are applying a filter condition (where(customer: { status: 'active' })) that involves the customer association.

In this case, the query might look like this:

```
SELECT orders.*, customers.* FROM orders
LEFT OUTER JOIN customers ON orders.customer_id = customers.id
WHERE customers.status = 'active';
```

And then it will also fetch the order_items in a separate query.

Example 4: Using includes with Ordering

```
# Using includes with ordering (order by customer name)
orders = Order.includes(:customer,
:order_items).order('customers.name ASC')

orders.each do |order|
  puts "Order ID: #{order.id}"
  puts "Customer Name: #{order.customer.name}"  # No extra query
for customer
  order.order_items.each do |item|
    puts "Order Item ID: #{item.id}"  # No extra query for order
items
  end
end
```

If you apply an order condition on the included association (customers.name ASC), Rails will use a JOIN because it needs to sort by customer.name.

The query might look like:

```
SELECT orders.*, customers.* FROM orders
LEFT OUTER JOIN customers ON orders.customer_id = customers.id
ORDER BY customers.name ASC;
```

Summary of Differences:

Feature	preload	includes
Query Strategy	Always uses separate queries for associations.	Can use separate queries or JOINs based on context.
Used For	Simple eager loading without filtering or conditions.	Flexible eager loading, can optimize with JOINs if needed.
When to Use	When you don't need any filtering, ordering, or complex joins.	When you might need to filter or order by associations or want flexibility.
Performance	Guarantees no JOIN and can be more efficient for large datasets without conditions.	Optimizes queries by using JOIN when conditions or ordering are involved.

When to Use Each:

- Use preload when you're sure you don't need to filter, order, or aggregate based on the included associations. This is most useful when you want to load data without worrying about conditions and you prefer separate queries to avoid unnecessary complexity.

- Use includes when you need to load associations but might also need to filter or order based on those associations. Rails will intelligently choose whether to use JOIN or separate queries based on the context, providing flexibility and performance optimizations.

Final Notes:

- includes gives you more flexibility, but it may use JOIN when conditions require it, which could lead to inefficient results if not used correctly.
- preload is more predictable in that it always uses separate queries, ensuring no unwanted joins but could be less efficient if a JOIN could be used instead.

If you run this code:

```
orders = Order.preload(:customer, :order_items).where(customer: { status: 'active' })
```

You will get an error, not just incorrect data.

Why the Error Occurs:

The error happens because you're trying to filter by an association (customer.status) while using preload, but preload does not perform a JOIN on the customer table.

14. Bang (!) Methods vs Normal Methods in Ruby:

In Common Methods

In Ruby, bang (!) methods are used to indicate a destructive operation—meaning they modify the object in place instead of returning a new object. Normal methods, on the other hand, return a modified copy while leaving the original object unchanged.

1️⃣ Normal Methods (Non-Bang)

- Do not modify the original object.
- Return a new object with the changes.
- Safer to use because they don't mutate the data.

```
str = "hello"
new_str = str.upcase # Returns a new object

puts str      # Output: "hello" (original string remains unchanged)
puts new_str  # Output: "HELLO" (new string)
```

2️⃣ Bang (!) Methods

- Modify the original object in place.
- Return the modified object (or sometimes nil if the operation fails).
- Use with caution because they change existing data.

```
str = "hello"
str.upcase!  # Modifies the original string

puts str  # Output: "HELLO" (original string is modified)
```

In ORM methods:

In Rails ActiveRecord, bang (!) methods are used to indicate exception-raising versions of common database operations. These methods modify records in the database and raise an exception if something goes wrong, unlike their non-bang counterparts, which return false on failure.

Method	Without ! (Normal)	With ! (Bang)

save	Returns true or false	Raises ActiveRecord::RecordInvalid if validation fails
create	Returns object (even if invalid)	Raises ActiveRecord::RecordInvalid if validation fails
update	Returns true or false	Raises ActiveRecord::RecordInvalid if validation fails
destroy	Returns object (even if not destroyed)	Raises ActiveRecord::RecordNotDestroyed if deletion fails

create vs create!

✅ Example (create)

```
user = User.create(name: nil) # Assuming name is required
puts user.persisted? # false (Record not saved)
```

💡 create returns an unsaved object if validation fails.

🚨 Example (create!)

```
user = User.create!(name: nil) # Assuming name is required
```

💥 Output (if validation fails)

```
ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

- Use create! when failure should stop execution with an error.

15. Proc Vs Lambda Vs Def

Difference Between Proc, Lambda, and def Method in Ruby

Feature	Proc	Lambda	def Method
Definition	Proc.new { } or proc { }	lambda { } or -> { }	Defined using def
Return Behavior	Returns from the enclosing scope when return is used	Returns only from the lambda itself	Returns from the method when called
Parameter Handling	Does not enforce the exact number of arguments	Strictly enforces the exact number of arguments	Strictly enforces arguments
Usage	Can be stored in a variable and passed around	Can be stored in a variable and passed around (acts like a method)	Defined within a class/module and bound to an object

---

1. Proc Example (Flexible Parameters, Returns Early)

```
proc_example = Proc.new { |x, y| puts "Sum: #{x + y}" }
proc_example.call(2, 3)  # Output: Sum: 5
proc_example.call(2)     # Output: Sum: 2 (y is nil, treated as 0)


Proc Exits Entire Method When return is Used
def test_proc
  my_proc = Proc.new { return "Proc Exited Early" }
  my_proc.call
  puts "This line won't execute"
end

puts test_proc  # Output: "Proc Exited Early"
```

---

2. Lambda Example (Strict Parameters, Local Return)

```
lambda_example = ->(x, y) { puts "Sum: #{x + y}" }
lambda_example.call(2, 3)  # Output: Sum: 5
lambda_example.call(2)     # Error: wrong number of arguments
(given 1, expected 2)


Lambda Does Not Exit the Enclosing Method
def test_lambda
  my_lambda = -> { return "Lambda Only Exits Itself" }
  puts my_lambda.call
  puts "This line will execute"
end

puts test_lambda
```

Output:

Lambda Only Exits Itself
This line will execute

---

3. def Method (Standard Function Definition)

```
def add(x, y)
  x + y
end

puts add(3, 5)  # Output: 8
```

- Unlike Proc and Lambda, methods are tied to objects (instance or class methods).
- Methods enforce parameters strictly like Lambda.

Key Takeaways

1. Use Proc when you want flexibility with arguments and don't mind it returning from the enclosing method.
2. Use Lambda when you need strict argument checking and want it to return only from itself.
3. Use def Methods for defining reusable functions inside classes or modules.

16. Mixins in Ruby

Mixins in Ruby

A Mixin in Ruby is a way to add functionality to a class without using inheritance, achieved by including modules.

Why Use Mixins?

1. Avoids multiple inheritance (since Ruby only supports single inheritance).
2. Encapsulates reusable behavior in modules that can be shared across different classes.
3. Keeps code DRY by defining reusable methods once in a module.

Example: Mixin for Logging

```
# Define a module (Mixin)
module Logger
  def log(message)
    puts "[LOG]: #{message}"
  end
end

# Class including the module
class User
  include Logger  # Adds Logger methods to User class

  def initialize(name)
    @name = name
  end

  def greet
    log("User #{@name} logged in.")  # Using log method from module
    puts "Hello, #{@name}!"
  end
end

# Usage
user = User.new("Alice")
user.greet
```

Output:

[LOG]: User Alice logged in.
Hello, Alice!

Key Mixin Methods in Ruby

Method	Purpose

include	Adds module methods as instance methods
extend	Adds module methods as class methods
prepend	Adds methods before the class's own methods

Example: extend vs include

```
module Greetings
  def say_hello
    puts "Hello!"
  end
end

class Person
  include Greetings   # Adds say_hello as an instance method
end

class Robot
  extend Greetings    # Adds say_hello as a class method
end

# Usage
p1 = Person.new
p1.say_hello  # Works, since it's an instance method

Robot.say_hello  # Works, since it's a class method
```

Deep Dive into prepend, Method Lookup, and Real-World Module Structures in Ruby

---

1. Understanding prepend

How is prepend Different from include?

- include adds module methods after the class's own methods.
- prepend adds module methods before the class's own methods (higher priority).

Example: include vs prepend

```
module Greetings
  def hello
    puts "Hello from Greetings!"
  end
end

class Person
  include Greetings
  def hello
    puts "Hello from Person!"
  end
end

p = Person.new
p.hello  # Output: Hello from Person! (Class method takes priority)
```

Now, if we use prepend instead:

```
class Person
  prepend Greetings
  def hello
    puts "Hello from Person!"
  end
end

p = Person.new
p.hello  # Output: Hello from Greetings! (Module method takes
priority)
```

✅ Use prepend when you want to override class methods before they execute.

---

2. Method Lookup in Ruby

Order of Method Resolution (Method Lookup Path)

Ruby searches for a method in this order:

1. Singleton class (if defined in object)
2. Prepend modules (if any)
3. Class itself
4. Include modules (in reverse order)
5. Superclass
6. Kernel Module (included in Object)
7. BasicObject (root class)

Example of Method Lookup

```
module Mixin1
  def greet
    puts "Hello from Mixin1!"
  end
end

module Mixin2
  def greet
    puts "Hello from Mixin2!"
  end
end

class Base
  def greet
    puts "Hello from Base!"
  end
end

class Person < Base
  prepend Mixin1  # Takes the highest priority
  include Mixin2  # Included after class methods
end

p = Person.new
p.greet
```

Output:

Hello from Mixin1! # prepend makes Mixin1 take priority

✅ If Mixin1 wasn't prepended, Person would call its own method first.

---

3. Real-World Module Structures

Example: Logging System with prepend for Method Wrapping

Imagine an app where every method call needs to be logged.

```
module Logger
  def process
    puts "[LOG]: Starting process..."
    super  # Calls the original method after logging
    puts "[LOG]: Process completed."
  end
end

class Task
  prepend Logger

  def process
    puts "Executing task..."
  end
end

t = Task.new
t.process
```

Output:

[LOG]: Starting process...
Executing task...
[LOG]: Process completed.

✅ This is useful for adding logging, profiling, or auditing to methods.

---

4. When to Use include, extend, and prepend

Method	Use Case	Behavior
include	Instance methods	Methods are available to instances of the class
extend	Class methods	Methods are available on the class itself
prepend	Method overriding	Methods are inserted before the class's own methods

---

Final Thought

- Use include for adding instance methods.
- Use extend to define class-level utilities.
- Use prepend when you need to override or intercept methods before they execute.

17. Method Lookup in Ruby

Order of Method Resolution (Method Lookup Path)

Ruby searches for a method in this order:

1. Singleton class (if defined)
2. Prepend modules (if any)
3. Class itself
4. Include modules (in reverse order)

5. Superclass
6. Kernel Module (included in Object)
7. BasicObject (root class)

```
module Mixin1
  def greet
    puts "Hello from Mixin1!"
  end
end
module Mixin2
  def greet
    puts "Hello from Mixin2!"
  end
end

class Base
  def greet
    puts "Hello from Base!"
  end
end

class Person < Base
  prepend Mixin1  # Takes the highest priority
  include Mixin2  # Included after class methods
end
```

```
p = Person.new

p.greet
```

Hello from Mixin1! # prepend makes Mixin1 take priority

18. Callbacks in Rails

Callbacks in Rails

Callbacks are methods that execute automatically at specific points in an object's lifecycle, such as before saving, updating, or deleting a record.

Feature	destroy	delete
Triggers callbacks?	✅ Yes (before_destroy, after_destroy)	❌ No
Deletes from DB?	✅ Yes	✅ Yes
Checks dependent records?	✅ Yes (dependent: :destroy)	❌ No
Soft delete possible?	✅ Yes (with gems like paranoia)	❌ No

Performance	⬇ Slower (because of callbacks)	⬆ Faster (direct DB delete)

Callback	Runs Before / After
before_save	Before saving to the database
after_save	After saving (both create & update)
before_create	Before creating a new record
after_create	After creating a new record
before_update	Before updating an existing record
after_update	After updating an existing record
before_destroy	Before deleting a record
after_destroy	After deleting a record

Types of Callbacks

Example: Using Callbacks in a Model

```
class User < ApplicationRecord
  before_save :normalize_name
  after_create :send_welcome_email

  private

  def normalize_name
    self.name = name.downcase.capitalize
  end

  def send_welcome_email
    UserMailer.welcome_email(self).deliver_now
  end
end
```

✅ before_save ensures names are formatted correctly before saving.
✅ after_create sends an email only after a new user is created.

Why Use Callbacks?

- Automate common tasks (e.g., formatting data, sending notifications).
- Enforce business logic at the model level.
- Reduce duplication (instead of adding the same logic in multiple controllers).

Skipping Callbacks in Rails
Sometimes, you might want to bypass callbacks for specific scenarios, like bulk inserts or test cases. Here's how you can skip callbacks in Rails.

Skipping Callbacks in Rails

Sometimes, you might want to bypass callbacks for specific scenarios, like bulk inserts or test cases. Here's how you can skip callbacks in Rails.

1️⃣ Using save(validate: false) to Skip Validations (But Not Callbacks)

```
user = User.new(name: "John Doe")
user.save(validate: false)  # Skips model validations but runs
callbacks
```

✅ Useful when importing large datasets where validation isn't required.
❌ Does NOT skip before_save or after_create callbacks.

2️⃣ Using update_column / update_all to Skip Callbacks

```
user.update_column(:name, "New Name")  # Updates DB directly,
skipping callbacks
```

✅ Skips callbacks (before_save, after_save, etc.)
✅ Useful for performance optimization in batch updates.
❌ Does NOT trigger validations or after_commit hooks.

For multiple records:

```
User.update_all(name: "Default Name")  # Skips callbacks &
validations
```

3️⃣ Using skip_callback to Disable Callbacks Temporarily

```
class User < ApplicationRecord
  skip_callback :save, :before, :normalize_name  # Disables
before_save :normalize_name
end
```

✅ Permanently removes a specific callback from execution.
❌ Alters the model globally, so use cautiously.

4️⃣ Using a Custom Flag to Manually Skip Callbacks

Modify your model:

```
class User < ApplicationRecord
  attr_accessor :skip_callbacks  # Virtual attribute

  before_save :normalize_name, unless: -> { skip_callbacks }

  private

  def normalize_name
    self.name = name.downcase.capitalize
  end
end


Then, when saving:

user = User.new(name: "john doe")
user.skip_callbacks = true  # Disables before_save
user.save
```

✅ Flexible: You can enable/disable callbacks dynamically.
✅ Scoped: Only affects the current instance, not the entire model.

When to Skip Callbacks?

Scenario	Method
Bulk updates	update_all
Avoid validations but keep callbacks	save(validate: false)

Skip specific callbacks permanently	skip_callback
Skip callbacks dynamically per instance	Custom flag (attr_accessor :skip_callbacks)

Final Thought

- Use update_column/update_all for fast updates without callbacks.
- Use skip_callback carefully, as it modifies model behavior permanently.
- Use a custom flag (skip_callbacks) for instance-level control.

---

19. CSRF

CSRF (Cross-Site Request Forgery) in Rails – Explained Clearly

---

🔹 What is CSRF?

CSRF (Cross-Site Request Forgery) is a type of cyber attack where a malicious website tricks an authenticated user into making unintended requests to another website.

🔹 How Does a CSRF Attack Work?

Scenario: Bank Account Transfer

1. You log in to your bank website (Bank. com) and have an active session.
2. Without logging out, you visit a malicious website (Hacker. com).

Hacker. com contains a hidden request, like:

```
<form action="https://bank.com/transfer" method="POST">
  <input type="hidden" name="amount" value="10000">
  <input type="hidden" name="to_account" value="HACKER_ACCOUNT">
</form>
<script>
  document.forms[0].submit();  // Auto-submits the form when page
loads
</script>
```

3. Since you're already logged in, your browser sends the request with your session cookies.
4. The bank processes the request as if you submitted it—money is transferred!

🚨 Danger: The request was forged by the attacker without your consent.

🔹 How Does Rails Prevent CSRF?

Rails protects against CSRF by using a secret authenticity token that must be included with every non-GET request.

Step 1: Rails Automatically Adds an Authenticity Token

Every form helper in Rails automatically includes a hidden CSRF token:

```
<%= form_with(url: "/transfer") do |form| %>
  <%= form.hidden_field :authenticity_token %>
  <%= form.text_field :amount %>
  <%= form.submit "Transfer" %>
<% end %>
```

🔹 What is this token?

- It is a randomly generated, unique string for each session.
- Stored on the server and validated when a request is made.

Step 2: Rails Verifies the CSRF Token

When the form is submitted, Rails checks if the authenticity token is valid before processing the request.

This is handled by this line in ApplicationController:

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception  # Default CSRF protection
end
```

🛑 If the token is missing or incorrect, Rails rejects the request with a 403 Forbidden error.

🔹 CSRF Protection in APIs (When to Disable?)

By default, Rails protects all POST, PUT, PATCH, and DELETE requests.

However, for APIs that don't use sessions (e.g., JSON-based APIs), CSRF protection is not needed.
For APIs, we disable it using:

```
class ApplicationController < ActionController::API
  include ActionController::RequestForgeryProtection
  protect_from_forgery with: :null_session
end
```

✅ This prevents CSRF while allowing APIs to function properly.

🔹 When to Disable CSRF? (Use With Caution ⚠️)

If you receive CSRF errors in certain cases (e.g., webhook requests from third-party services like Stripe), you can skip CSRF protection for specific actions:

```
skip_before_action :verify_authenticity_token, only: [:webhook]
```

🔹 Only disable CSRF when necessary! Disabling it for all actions makes your app vulnerable.

🔹 Summary

Feature	Explanation
CSRF Attack	Tricks a logged-in user into performing unintended actions (e.g., transferring money).
Rails Protection	Uses a hidden authenticity token to verify requests.
Enforcement	protect_from_forgery with: :exception blocks invalid requests.
APIs	protect_from_forgery with: :null_session for API requests.
When to Disable?	Only for webhooks or APIs that don't use sessions.

20. has_many :through

🔹 has_many :through Association in Rails

The has_many :through association is used to set up a many-to-many relationship between two models via a third "join" model.

🔹 When to Use has_many :through?

Use has_many :through when you need extra data in the join table.
For example:

- A Doctor has many Patients through Appointments.
- The Appointments table holds additional details (like date, status, etc.) that we need.

🔹 Example: Doctor, Patient, and Appointment

1️⃣ Database Schema

```
class CreateDoctorsPatientsAppointments <
ActiveRecord::Migration[6.1]
  def change
    create_table :doctors do |t|
      t.string :name
      t.timestamps
    end

    create_table :patients do |t|
      t.string :name
      t.timestamps
    end

    create_table :appointments do |t|
      t.references :doctor, foreign_key: true
      t.references :patient, foreign_key: true
      t.date :appointment_date
      t.timestamps
    end
  end
end
```

2️⃣ Defining Models

```
class Doctor < ApplicationRecord
  has_many :appointments
  has_many :patients, through: :appointments  # Many patients
through appointments
end

class Patient < ApplicationRecord
  has_many :appointments
  has_many :doctors, through: :appointments  # Many doctors through
appointments
end

class Appointment < ApplicationRecord
  belongs_to :doctor
  belongs_to :patient
end
```

3️⃣ Querying the Relationship

```
# Create records
doctor = Doctor.create(name: "Dr. Smith")
patient = Patient.create(name: "John Doe")

# Creating an appointment (join table record)
appointment = Appointment.create(doctor: doctor, patient: patient,
appointment_date: Date.today)

# Fetch all patients of a doctor
doctor.patients  # => Returns [patient]

# Fetch all doctors of a patient
patient.doctors  # => Returns [doctor]

# Fetch all appointments of a doctor
doctor.appointments
```

Feature	has_many :through	has_and_belongs_to_many
Requires a join table?	✅ Yes, but it's a separate model	✅ Yes, but no model needed
Can store extra data in the join table?	✅ Yes (e.g., appointment date)	❌ No
More flexible?	✅ Yes	❌ No
Recommended for complex relationships?	✅ Yes	❌ No

- Why Use has_many :through Instead of has_and_belongs_to_many?

---

21. Polymorphic Associations

- Polymorphic Associations in Rails (Complete Guide)
- What is a Polymorphic Association?

A polymorphic association allows a model to belong to multiple other models using a single association. Instead of using separate foreign keys like post_id or photo_id, Rails stores:

- commentable_id → The ID of the associated record.
- commentable_type → The model name ("Post", "Photo").

🔹 Example: A Comment Model for Post and Photo

1️⃣ Migration for posts Table

```
class CreatePosts < ActiveRecord::Migration[6.1]
  def change
    create_table :posts do |t|
      t.string :title
      t.text :body
      t.timestamps
    end
  end
end
```

2️⃣ Migration for photos Table

```
class CreatePhotos < ActiveRecord::Migration[6.1]
  def change
    create_table :photos do |t|
      t.string :url
      t.timestamps
    end
  end
end
```

3️⃣ Migration for comments Table (Polymorphic)

```
class CreateComments < ActiveRecord::Migration[6.1]
```

```
  def change
    create_table :comments do |t|
      t.text :content
      t.references :commentable, polymorphic: true, index: true  #
Polymorphic reference
      t.timestamps
    end
  end
end
```

- Model Definitions

Post Model

```
class Post < ApplicationRecord
  has_many :comments, as: :commentable  # A post can have many
comments
end
```

Photo Model

```
class Photo < ApplicationRecord
  has_many :comments, as: :commentable  # A photo can have many
comments
end
```

Comment Model (Polymorphic)

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true  # Belongs to either
Post or Photo
end
```

- Database Schema After Migration

id	title	body	created_at	updated_at
1	"Rails Guide"	"Learn Rails"	Timestamp	Timestamp

id	url	created_at	updated_at
1	"image1.jpg"	Timestamp	Timestamp

id	content	commentable_id	commentable_type	created_at
1	"Nice post!"	1	"Post"	Timestamp
2	"Cool photo"	1	"Photo"	Timestamp

- How commentable Works?

1️⃣ Creating a Comment for a Post

```
post = Post.find(1)
post.comments.create(content: "Nice post!")
```

- commentable_id: 1
- commentable_type: "Post"

2️⃣ Creating a Comment for a Photo

```
photo = Photo.find(1)
photo.comments.create(content: "Cool photo")
```

- commentable_id: 1

- commentable_type: "Photo"

3️⃣ Fetching Comments for a Post

```
post = Post.find(1)
post.comments  # Returns all comments where commentable_type =
"Post" AND commentable_id = 1
```

4️⃣ Fetching Comments for a Photo

```
photo = Photo.find(1)
photo.comments  # Returns all comments where commentable_type =
"Photo" AND commentable_id = 1
```

5️⃣ Finding the Associated Model (Post or Photo) from a Comment

```
comment = Comment.find(1)
comment.commentable  # Returns the associated Post or Photo object
```

🔹 When to Use Polymorphic Associations?

✅ When multiple models need to share a common association (e.g., comments, tags, likes).
✅ When you don't want separate foreign keys for different models.
✅ When you want to avoid complex join tables.

22. Preventing SQL Injection in Rails

🔹 Preventing SQL Injection in Rails

SQL injection occurs when user input is directly interpolated into SQL queries, allowing attackers to manipulate database queries.

🔹 How Rails Prevents SQL Injection by Default

Rails Active Record automatically uses parameterized queries, which prevent direct SQL injection.

✅ Safe Example (Using ? Placeholder)
User.where("email = ?", params[:email]) # Uses bound parameters

The ? placeholder prevents injection by treating input as data, not SQL code.

🚨 Unsafe Example (Direct String Interpolation)

```
User.where("email = '#{params[:email]}'")  # ⚠️ Vulnerable to injection!
```

If params[:email] contains malicious SQL, it gets executed as part of the query.

🔹 Best Practices to Prevent SQL Injection in Rails

1️⃣ Always Use ActiveRecord Query Methods

✅ Use where, find_by, and pluck Safely

```
User.where("email = ?", params[:email]) # Safe
User.find_by(email: params[:email])     # Safe
```

🚨 Avoid Direct Interpolation

```
User.where("email = '#{params[:email]}'")  # ⚠️ Unsafe!
```

2️⃣ Use Hash Syntax in ActiveRecord

```
User.where(email: params[:email])  # Safe and readable
```

3️⃣ Prevent Injection in find_by_sql and Raw Queries

🚨 Unsafe: Direct String Interpolation

```
User.find_by_sql("SELECT * FROM users WHERE email =
'#{params[:email]}'")  # ⚠️ Unsafe!
```

✅ Safe: Use Parameterized Queries

```
User.find_by_sql(["SELECT * FROM users WHERE email = ?",
params[:email]])  # Safe
```

4️⃣ Sanitize User Input for Dynamic Queries

When dynamic queries are necessary, use sanitize_sql_for_conditions:

```
User.where(User.sanitize_sql(["email = ?", params[:email]]))
```

5️⃣ Use AREL for Complex Queries

Arel allows safe dynamic queries without string interpolation.

```
users = User.arel_table
User.where(users[:email].eq(params[:email]))  # Safe
```

🔹 Summary of Best Practices

Method	Safe?	Explanation

User.where("email = ?", params[:email])	✅	Uses bound parameters
User.where(email: params[:email])	✅	Uses hash syntax
User.find_by_sql(["SELECT * FROM users WHERE email = ?", params[:email]])	✅	Uses parameterized query
User.where("email = '#{params[:email]}'")	❌	Vulnerable to injection
User.find_by_sql("SELECT * FROM users WHERE email = '#{params[:email]}'")	❌	Vulnerable to injection
User.where(User.sanitize_sql(["email = ?", params[:email]]))	✅	Sanitized query

🚨 What Happens If You Don't Follow SQL Injection Prevention in Rails?

If you fail to prevent SQL injection, attackers can manipulate your database queries, leading to data breaches, data loss, and system compromise. Here's a breakdown of what happens when unsafe code is used.

1️⃣ Example: Unsafe Query

🚨 Direct String Interpolation (Vulnerable Code)

```
User.where("email = '#{params[:email]}'")  # ⚠️ Unsafe!
```

👀 What an Attacker Can Do

If a hacker enters this in the form:

' OR 1=1 --

The query becomes:

```
SELECT * FROM users WHERE email = '' OR 1=1 --'
```

Since 1=1 is always true, it returns all users instead of just one.

🔴 Consequences

- Leaking all user accounts
- Bypassing authentication (e.g., logging in without credentials)

2️⃣ Example: Data Deletion Attack

If a developer writes:

```
User.where("email = '#{params[:email]}'").delete_all  # ⚠️ Unsafe!
```

An attacker inputs:

' OR 1=1 --

The resulting SQL:

```
DELETE FROM users WHERE email = '' OR 1=1 --'
```

🔴 Consequences

- Deletes all users
- Permanent data loss if no backups

3️⃣ Example: Gaining Admin Access

If your login system checks:

```
User.where("email = '#{params[:email]}' AND password =
'#{params[:password]}'").first
```

An attacker enters:

' OR '1' = '1'; --

The resulting SQL:

```
SELECT * FROM users WHERE email = '' OR '1' = '1' --' AND password
= ''
```

Since '1' = '1' is always true, it logs in the attacker as the first user in the database (often an admin).

🔴 Consequences

- Full admin access to the system
- Stealing user data
- Deleting/modifying database records

4️⃣ Example: Exploiting the Database

If an app uses raw SQL:

```
User.find_by_sql("SELECT * FROM users WHERE email =
'#{params[:email]}'")  # ⚠️ Unsafe!
```

An attacker enters:

```
'; DROP TABLE users; --
```

The resulting SQL:

```
SELECT * FROM users WHERE email = ''; DROP TABLE users; --'
```

🔴 Consequences

- Deletes entire users table
- Brings down the application
- Massive data loss

🚀 How to Stay Safe?

✅ Use Parameterized Queries

```
User.where("email = ?", params[:email])  # Safe!
```

✅ Use Hash Syntax

```
User.where(email: params[:email])  # Safe!
```

✅ Sanitize Input for Raw Queries

```
User.sanitize_sql(["email = ?", params[:email]])
```

📌 Key Takeaways

- 🚨 Direct string interpolation in SQL queries is dangerous
- 🔴 Hackers can bypass login, steal data, delete tables, or get admin access
- ✅ Always use parameterized queries or ActiveRecord methods

23. Single Table Inheritance (STI) in Rails

🔹 What is Single Table Inheritance (STI)?

STI (Single Table Inheritance) is a Rails feature that allows multiple models to share a single database table while behaving like separate classes in the application.

✅ When to Use STI?

- When you have multiple models that share common attributes but have some unique behaviors.
- Instead of creating separate tables for each model, you use one table with a "type" column.

🔹 How Does STI Work?

STI works by storing all child classes in a single database table with a special column named type that tells Rails which subclass an object belongs to.

🎯 Example Scenario: Employees in a Company

Imagine you have a system where there are different types of employees:

- Manager
- Developer
- Designer
  Each type of employee has some shared attributes (name, salary) but may have unique methods.

1️⃣ Define the Parent Class (Employee)

```
class Employee < ApplicationRecord
  self.inheritance_column = :type  # Tells Rails to use 'type' for
STI
end

or

class Employee < ActiveRecord::Migration[6.1]
  def change
    create_table :users do |t|
      t.string :name
      t.integer :salary
      t.string :type  # This column differentiates subclasses
      t.timestamps
    end
  end
end
```

2️⃣ Define the Child Classes

```
class Manager < Employee
  def bonus
    salary * 0.2  # Managers get 20% bonus
  end
end

class Developer < Employee
  def bonus
    salary * 0.15  # Developers get 15% bonus
  end
end

class Designer < Employee
```

```
  def bonus
    salary * 0.1  # Designers get 10% bonus
  end
end
```

🔹 Each subclass behaves like a separate model but shares the same employees table.

3️⃣ Database Schema for STI

Run a migration:

rails g migration CreateEmployees name:string salary:integer type:string
rails db:migrate

Generated employees table:

id	name	salary	type
1	Alice	80000	Manager
2	Bob	60000	Developer
3	Carol	50000	Designer

4️⃣ Creating Records

```
Manager.create(name: "Alice", salary: 80000)
Developer.create(name: "Bob", salary: 60000)
```

```
Designer.create(name: "Carol", salary: 50000)
```

5️⃣ Querying STI Models

```
Employee.all         # Returns all employees (Managers, Developers,
Designers)
Manager.all          # Returns only managers
Developer.all        # Returns only developers
```

6️⃣ Using Unique Methods

```
manager = Manager.find_by(name: "Alice")
manager.bonus    # => 16000  (20% of 80000)

developer = Developer.find_by(name: "Bob")
developer.bonus  # => 9000  (15% of 60000)
```

🔹 Pros & Cons of STI

✅ Pros	❌ Cons
Avoids duplicate tables for similar models	Can lead to a large table with many NULL fields
Easier to manage common logic in a single table	If subclasses have very different attributes, it becomes inefficient

Simpler querying and relationships	Hard to enforce table constraints for each subclass

---

🔹 When NOT to Use STI?

❌ When child models have very different attributes (too many NULL fields).
✅ Instead, use Polymorphic Associations or Separate Tables.

🚀 Summary

- STI allows multiple models to share a single table.
- It uses a "type" column to differentiate records.
- Each subclass can have custom methods and behavior.
- Best suited for similar models with small variations.

24. Concerns in Rails

Concerns in Rails

Concerns are just Rails-specific mixins but structured in a way that integrates better with ActiveRecord and Rails conventions.

🔹 What are Concerns?

- Concerns are modules in Rails used to organize and reuse shared logic across models or controllers.
- They keep code DRY (Don't Repeat Yourself) by extracting common methods into separate files.
- They are typically stored in the app/models/concerns/ or app/controllers/concerns/ directories.

🔹 Why Use Concerns?

1. Avoid code duplication (extract shared logic).
2. Keep models and controllers clean & maintainable.

3. Easily include logic in multiple places.

📝 Example: Using Concerns in a Model

Scenario

We have User and Order models. Both log changes before saving, so instead of repeating the logic, we extract it into a Concern.

Step 1: Create a Concern

Create a new file:
📂 app/models/concerns/loggable.rb

```
module Loggable
  extend ActiveSupport::Concern

  included do
    before_save :log_changes
  end

  def log_changes
    puts "Logging changes for #{self.class.name} - #{self.id ||
'New Record'}"
  end
end
```

Breakdown:

- extend ActiveSupport::Concern → Helps define methods and hooks (before_save).
- included do ... end → Runs before_save when included in a model.
- log_changes → Method to log changes before saving.

Step 2: Include Concern in Models

Modify User and Order models:

📂 app/models/user.rb

```
class User < ApplicationRecord
  include Loggable
end
```

📂 app/models/order.rb

```
class Order < ApplicationRecord
  include Loggable
end
```

Step 3: Test in Rails Console

```
user = User.create(name: "John Doe")
order = Order.create(order_number: "12345")
```

Output:
Logging changes for User - New Record
Logging changes for Order - New Record

📌 Benefit:
✅ User and Order reuse the same log_changes method without duplicating code.

🔹 Using Concerns in Controllers

Scenario

We want to track page visits for multiple controllers (ProductsController, OrdersController).

Step 1: Create a Controller Concern

📂 app/controllers/concerns/trackable.rb

```
module Trackable
  extend ActiveSupport::Concern

  included do
    before_action :track_visit
  end

  def track_visit
    puts "User visited #{controller_name}##{action_name}"
  end
end
```

Step 2: Include in Controllers

📂 app/controllers/products_controller.rb

```
class ProductsController < ApplicationController
  include Trackable

  def index
    render plain: "Products Page"
  end
end
```

📂 app/controllers/orders_controller.rb

```
class OrdersController < ApplicationController
  include Trackable

  def index
    render plain: "Orders Page"
  end
end
```

Step 3: Test

Visit:
🔗 http://localhost:3000/products
🔗 http://localhost:3000/orders

📌 Output in Logs:

User visited products#index
User visited orders#index

🔹 Summary Table

Feature	Purpose	Example
Concerns in Models	Reuse common logic across models	Loggable module for logging changes
Concerns in Controllers	Share actions across multiple controllers	Trackable module for tracking visits

📌 Concerns keep Rails code modular, clean, and reusable! 🚀

25. Devise in Rails: Authentication Made Easy

What is Devise?

- Devise is a full-featured authentication solution for Rails applications.
- It provides user registration, login, password recovery, session management, and more out-of-the-box.
- It follows the "Rails Engine" pattern, meaning it works like a mini app inside your Rails project.

🔹 Installing Devise in Rails

Step 1: Add Devise to Gemfile

```
gem 'devise'
```

Then, run:

```
bundle install
```

Step 2: Install Devise

```
rails generate devise:install
```

This creates an initializer at:
📂 config/initializers/devise.rb

Step 3: Generate the User Model

```
rails generate devise User
rails db:migrate
```

This adds a users table with authentication fields like email, encrypted password, and reset password tokens.

Step 4: Add Devise Routes

Modify config/routes.rb:

```
Rails.application.routes.draw do
  devise_for :users
  root "home#index"  # Your home page
end
```

- Using Devise in a Controller

Devise provides helper methods for authentication.

Example: Protecting a Page

📂 app/controllers/dashboard_controller.rb

```
class DashboardController < ApplicationController
  before_action :authenticate_user!

  def index
    render plain: "Welcome to your dashboard,
#{current_user.email}!"
  end
end
```

Here, authenticate_user! ensures only logged-in users can access the dashboard#index action.

Example: Accessing Current User

```
if user_signed_in?
  puts "Current User: #{current_user.email}"
else
  puts "User not signed in"
end
```

🔹 Customizing Devise Views

Generate Devise views:

```
rails generate devise:views
```

This creates templates in:
📂 app/views/devise/registrations/
📂 app/views/devise/sessions/

You can now modify these for custom styling and behavior.

🔹 Customizing User Model

Devise provides many built-in modules:

```
class User < ApplicationRecord
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable
end
```

- :database_authenticatable → Handles authentication.
- :registerable → Allows user sign-ups.
- :recoverable → Provides password reset functionality.
- :validatable → Enforces email/password validations.

🔹 Adding Extra Fields to Users

Let's add a username to the User model.

Step 1: Create Migration

```
rails generate migration AddUsernameToUsers username:string
rails db:migrate
```

Step 2: Permit the New Parameter

Modify app/controllers/application_controller.rb:

```
before_action :configure_permitted_parameters, if:
:devise_controller?

protected

def configure_permitted_parameters
  devise_parameter_sanitizer.permit(:sign_up, keys: [:username])
end
```

Now, users must enter a username while signing up.

- Summary Table

Feature	Purpose	Example
User Authentication	Login, Signup, Logout	devise_for :users in routes
Protect Routes	Restrict pages to logged-in users	before_action :authenticate_user!
Current User	Access logged-in user data	current_user.email
Custom Fields	Add extra attributes like username	Use devise_parameter_sanitizer

Most Used Rails Commands 💾

1. rails new project_name
2. rails new pn -d mysql / sqlite3
3. rails db:create
    - This creates the database for the current Rails environment (e.g., development or test).
    - Created database 'myapp_development'

4. rails db:migrate
5. rails db:rollback

6. rails s
7. rails dbconsole
8. rails g model Post title:string content:text
9. rails g model Posts index show
10. rails g migration AddViewsToPosts views:integer
11. rails routes
12. rails routes -c controller_name
13. gem install rails

How to Seed Data in Rails (rails db:seed)?

How to Seed Data in Rails (rails db:seed)?

Seeding in Rails allows you to populate your database with initial data using the db/seeds.rb file. This is useful for setting up default values, testing data, or demo environments.

1️⃣ Steps to Seed Data in Rails

1. Open db/seeds.rb
Add the data you want to insert. Example:

```
# Create some users
User.create!([
  { name: "Alice", email: "alice@example.com" },
  { name: "Bob", email: "bob@example.com" }
])

# Create categories
Category.create!(name: "Technology")
Category.create!(name: "Science")
```

2. Run the Seed Command

```
rails db:seed
```

- This executes the db/seeds.rb file and inserts data into your database.

2️⃣ Running Seeds with Specific Environment
If you want to seed for production, use:

```
RAILS_ENV=production rails db:seed
```

3️⃣ Running Seeds Along with Database Setup
For a fresh setup, you can run:

```
rails db:setup
```

- This runs rails db:create, rails db:migrate, and rails db:seed in one step.

4️⃣ Using find_or_create_by to Avoid Duplicates
To prevent inserting the same data multiple times, use:

```
User.find_or_create_by!(email: "alice@example.com") do |user|
  user.name = "Alice"
end
```

5️⃣ Using Seed Files per Environment
If you want different seed data for development, test, and production, create files under db/seeds/:

db/seeds/
  development.rb
  production.rb
  test.rb

Then, in db/seeds.rb, add:

load Rails.root.join("db/seeds/#{Rails.env}.rb")

Now, running rails db:seed will execute the correct file based on the environment.

6️⃣ Reset and Reseed the Database
To clear the database and reseed:

rails db:reset

or manually:

rails db:drop db:create db:migrate db:seed

Rails ActiveRecord

1. To create a table:

   rails g migration CreateLocationMappingTable

```
class CreateLocationTable < ActiveRecord::Migration[6.0]
  def change
    create_table :location do |t|
      # Integer column with default value and non-null constraint
      t.integer :loc_id, null: false, default: 0

      # String column with a limit on the length of characters
      t.string  :location_code, limit: 50, null: false

      # String column without any constraints
      t.string  :status, default: 'active'

      # Text column for larger error logs
      t.text    :error_log, limit: 500

      # DateTime column, Rails automatically provides created_at and updated_at
with timestamps
      t.timestamps

      # Add more columns with additional constraints
      t.boolean :active, default: true, null: false
      t.float   :lat, null: false
      t.float   :lng, null: false

      # You can also use decimal for precise number storage (e.g., for monetary
values)
      t.decimal :price, precision: 10, scale: 2, null: false
```

```
      # Enum type stored as string (useful for limited set of options like
'active', 'inactive')
      t.string  :state, null: false, default: 'pending'
    end

    # Indexes: You can add multiple indexes as needed
    add_index :location, :location_code
    add_index :location, :loc_id
    add_index :location, :status
    add_index :location, :error_log

    # Multi-column index
    add_index :location, [:location_code], unique: true

    # Foreign key constraints
    # Assuming loc_id refers to another table (e.g., categories table)
    add_foreign_key :location, :warehouse, column: :loc_id

    # Check constraints (to enforce certain conditions)
    # Note: Requires database support (e.g., PostgreSQL)
    add_check_constraint :location, 'status IN (\'active\', \'inactive\')', name:
'check_status'

    # Indexing foreign key columns for performance
    add_index :location, :loc_id, name: 'index_location_on_loc_id'
  end
end

# For the same column names,

    add_foreign_key :orders, :customers, column: :customer_id, primary_key: :id


rails db:migrate:up VERSION=20250226123456
rails db:migrate:down VERSION=20250226123456
```

2. Active Record

Each Rails model represents a database table, and ActiveRecord maps table columns to Ruby attributes.

rails generate model User name:string email:string age:integer

rails db:migrate

```
class User < ApplicationRecord
     validates :name, presence: true
     validates :email, uniqueness: true
     validates :age, numericality: { greater_than: 18 }
     validates:age, numericality: { greater_than: 18 }
     validates :age, numericality: { only_integer: true }
     validates :price, numericality: { greater_than_or_equal_to: 0.01 }
     validates :name, length: { minimum: 3 }
     validates :bio, length: { maximum: 500 }
     validates :username, length: { in: 3..15 }
     validates :email, format: { with: URI::MailTo::EMAIL_REGEXP }
     validates :phone_number, format: { with: /\A\d{10}\z/, message: "must be
     10 digits" }
     validates :status, inclusion: { in: %w(active inactive pending), message:
     "%{value} is not a valid status" }
     validates :username, exclusion: { in: %w(admin superuser), message:
     "%{value} is reserved" }
     validate :age_must_be_even

     def age_must_be_even
       errors.add(:age, "must be even") if age.odd?
     end

     end

     class User < ApplicationRecord
          has_many :posts
     end

     # Connecting DB

     class DBtoBase < ActiveRecord::Base
          self.abstract_class = true
          establish_connection("DB_#{Rails.env}".to_sym)
     end

     class User < ActiveRecord::Base
          has_and_belongs_to_many :groups
     end

     class Group < ActiveRecord::Base
```

```
    has_and_belongs_to_many :users
end
```

ActiveAdmin Gem

ActiveAdmin is a Rails gem for creating admin dashboards easily.

```
# Add this to Gemfile
gem 'activeadmin'

# Install the gem
bundle install

# Generate ActiveAdmin configuration
rails generate active_admin:install

# Migrate the database
rails db:migrate

# Start Rails server
rails server
```

app/admin/users.rb

```
ActiveAdmin.register User do
  permit_params :name, :email, :age

  filter :created_at, label: 'Created at', as: :date_range
  filter :location, as: :select, collection: MyTable.all.map {
|location| [location.name,location.id] }, as: :select

  index do
    selectable_column
    id_column
    column :name
    column :email
    column :age
    actions
  end
```

```
  form do |f|
    f.inputs do
      f.input :name
      f.input :email
      f.input :age
    end
    f.actions
  end


controller do
    def scoped_collection
      TableModel.get_detail
    end
 end
end
```

Rails Jobs & Mailer (ActiveJob & ActionMailer)

ActiveJob

ActiveJob allows you to run tasks in the background instead of blocking the main application thread.

By default, Rails uses an inline queue (runs jobs immediately), but in production, you should use a background worker like: Sidekiq

```
class SendWelcomeEmailJob < ApplicationJob
  queue_as : queue_adapter, eg: sidekiq

  def perform(user)
    UserMailer.welcome_email(user).deliver_now
  end
end
```

ActionMailer

ActionMailer allows Rails apps to send emails easily using SMTP, SendGrid, Postmark, or any email service.

rails generate mailer UserMailer

```
# app/mailers/user_mailer.rb
class UserMailer < ApplicationMailer
  default from: 'no-reply@myapp.com'

  def welcome_email(user)
    @user = user
    mail(to: @user.email, subject: 'Welcome to MyApp!')
  end
end
```

Create a view in app/views/user_mailer/welcome_email.html.erb:

erb

```
<h1>Welcome to MyApp, <%= @user.name %>!</h1>
<p>We're excited to have you on board.</p>
```

```
UserMailer.welcome_email(@user).deliver_now # Sends immediately
```

Setting Up a Sidekiq Worker in Rails

```
bundle add sidekiq
bundle install


# config/application.rb
module MyApp
  class Application < Rails::Application
    config.active_job.queue_adapter = :sidekiq
```

```
  end
end

rails generate sidekiq:worker EmailNotification

app/workers/email_notification_worker.rb

# app/workers/email_notification_worker.rb
class EmailNotificationWorker
  include Sidekiq::Worker
  sidekiq_options queue: ‘queue_name’, retry: 5

  def perform(user_id)
    user = User.find(user_id)
    UserMailer.welcome_email(user).deliver_now
    Rails.logger.info "Sent welcome email to #{user.email}"
  end
end


EmailNotificationWorker.perform_async(@user.id) # Calls Sidekiq
Worker
EmailNotificationWorker.perform_in(10.minutes, user.id)
```

Setting up Redis in a Rails Application

```
gem 'redis'
bundle install

mkdir -p config/initializers
touch config/initializers/redis.rb

require 'redis'

$redis_validation_data = Redis.new(:host =>
REDIS_DETAILS_CONFIG['server'], :port =>
```

```
REDIS_DETAILS_CONFIG['port'] ,:db =>
REDIS_DETAILS_CONFIG['redis_validation_data']['db_num'])

REDIS.set("my_key", "Hello Redis")  # Store value
puts REDIS.get("my_key")            # Retrieve value
```

In Redis, DB number (or database number) refers to the logical separation of data within a single Redis instance. Redis supports multiple databases, and each database is identified by a numeric index. By default, Redis has 16 databases, indexed from 0 to 15.

require VS include VS extend

In Ruby, require, include, and extend are commonly used keywords, but they serve very different purposes. Let's break them down and explain the differences:

1. require

- Purpose: Used to load external files or libraries.
- Usage: Typically used to include external Ruby files or modules, whether they're part of the Ruby standard library or third-party gems.
- Where it works: Works at the file level. It loads external files or libraries and makes them available to the current program.

Example:
Load the 'date' library from the standard library
require 'date'

# Now you can use the Date class from the 'date' library
today = Date.today
puts today

When you require a file, Ruby looks for it in the $LOAD_PATH (which includes the standard library and the directories where Ruby gems are installed).

You can also use require_relative to load a file that is relative to the current file.

Example with require_relative:
require_relative 'my_module'

include MyModule

2. include

- Purpose: Used to include a module into a class or another module.
- Usage: When you include a module, it mixes in the module's methods as instance methods into the class.
- Where it works: It works at the class or module level. When included, the module's methods are available as instance methods of the class or module where it's included.

Example:

```
module Greetable
  def greet
    puts "Hello!"
  end
end

class Person
  include Greetable
end

person = Person.new
person.greet  # => "Hello!"
```

- In the above example, Greetable is included in the Person class, making the greet method available as an instance method of Person.

- Instance Methods: When you include a module, it gives access to the module's methods as instance methods for the class or module where it's included.

3. extend

- Purpose: Used to extend a module into a class or object, making the module's methods available as class methods or singleton methods.
- Usage: When you extend a module, the module's methods become class methods for the class, or singleton methods for an individual object.

- Where it works: Works at the object or class level. When a module is extended into a class or object, the module's methods are available as class methods or singleton methods.

Example (with class-level methods):

```
module Greetable
  def greet
    puts "Hello!"
  end
end

class Person
  extend Greetable
end

Person.greet  # => "Hello!"
```

- In this case, we used extend to add greet as a class method to the Person class.
- Singleton Methods: If you extend a module into a specific object, the methods in the module will be available only on that object as singleton methods.

Example (with object-level methods):

```
module Greetable
  def greet
    puts "Hello!"
  end
end

person = Object.new
person.extend(Greetable)

person.greet  # => "Hello!"
```

- Here, we used extend on an object (person), which gives the object its own version of the greet method, making it a singleton method for person.

Summary:

- require: Loads external files or libraries, making them available in your program. It's about bringing in dependencies or modules, typically at the file level.

- include: Mixes in a module's methods as instance methods into a class or module. This allows an object of that class to access the module's methods as instance methods.

- extend: Adds a module's methods as class methods (when used in a class) or singleton methods (when used on an individual object).

Practical Examples of Each:

```
# Example of require
require 'json'  # External library

json_string = '{"name": "John", "age": 30}'
parsed_data = JSON.parse(json_string)
puts parsed_data['name']  # => "John"


# Example of include
module Walkable
  def walk
    puts "Walking..."
  end
end

class Human
  include Walkable
end

human = Human.new
human.walk  # => "Walking..."


# Example of extend
module Speakable
  def speak
```

```
    puts "Hello!"
  end
end

class Animal
  extend Speakable
end

Animal.speak  # => "Hello!"  # Class-level method

dog = Object.new
dog.extend(Speakable)
dog.speak  # => "Hello!"  # Singleton method
```

Quick Reference:

- require → Load external files (external dependencies or files).
- include → Mix a module's methods as instance methods into a class.
- extend → Mix a module's methods as class or singleton methods into a class or object.

Naming Rules in Rails

Rails follows convention over configuration, which means naming conventions are crucial for smooth development. Below are the naming rules and best practices for different parts of a Rails application.

1️⃣ Models

- Model names are singular and CamelCase.
- Model file names are snake_case and singular.

✅ Example:

Model class (CamelCase, singular)

```
class UserProfile < ApplicationRecord
end
```

File name (snake_case, singular)
app/models/user_profile.rb

2️⃣ Database Tables

- Table names are plural and snake_case.
- Rails automatically infers table names from model names.

✅ Example:

users # for User model
order_items # for OrderItem model
customer_addresses # for CustomerAddress model

Tip: Use rails g model User → This creates a users table automatically.

3️⃣ Controllers

- Controller names are plural and CamelCase.
- File names are snake_case and plural.

✅ Example:

Controller class (CamelCase, plural)

```
class UsersController < ApplicationController
end
```

File name (snake_case, plural)
app/controllers/users_controller.rb

4️⃣ Routes (URLs)

- Use snake_case and plural for resource routes.
- RESTful routes follow CRUD actions (index, show, create, update, destroy).

✅ Example (config/routes.rb):

resources :users # Generates standard RESTful routes

/users   # index
/users/:id   # show
/users/new   # new
/users/:id/edit # edit

5️⃣ Migrations

- Migration class names use CamelCase and Create/Update/TableName.
- Migration file names are snake_case and timestamped.

✅ Example:

Migration file (CamelCase)

```
class CreateUsers < ActiveRecord::Migration[7.0]
end
```

File name (snake_case with timestamp)
db/migrate/20240225123045_create_users.rb

6️⃣ Associations

- Singular for belongs_to, plural for has_many.
- Use snake_case for foreign keys.

✅ Example:

```
class Order < ApplicationRecord
  belongs_to :user  # Singular
```

```
  has_many :order_items # Plural
end

# Foreign key in migrations (snake_case)
t.references :user, foreign_key: true
```

7️⃣ Views & Partials

- View files match controller actions (index.html.erb, show.html.erb).
- Partials start with an underscore (_).

✅ Example:

app/views/users/index.html.erb # Listing all users
app/views/users/show.html.erb # Display one user
app/views/users/_form.html.erb # Partial for forms

8️⃣ Helpers

- Helper names are plural and CamelCase.
- File names are snake_case and plural.

✅ Example:

Helper class (CamelCase, plural)

```
module UsersHelper
end
```

File name (snake_case, plural)
app/helpers/users_helper.rb

9️⃣ Mailers

- Mailer class names are CamelCase, singular.
- File names are snake_case.

- Mailer views are in a directory matching the mailer name.

✅ Example:

```
class UserMailer < ApplicationMailer
end
```

app/mailers/user_mailer.rb
app/views/user_mailer/welcome_email.html.erb

🔟 Background Jobs (ActiveJob & Sidekiq)

- Job class names end in Job, use CamelCase.
- Sidekiq worker class names end in Worker.

✅ Example:

```
# ActiveJob (CamelCase, ends with Job)
class SendWelcomeEmailJob < ApplicationJob
end

# Sidekiq Worker (CamelCase, ends with Worker)
class SendWelcomeEmailWorker
  include Sidekiq::Worker
end
```

🔢 Constants & Variables

Constants

- UPPER_CASE_SNAKE_CASE.
- Defined inside modules or classes.

✅ Example:

```
class Order
```

```
  DEFAULT_STATUS = "pending"
end
```

Variables

- Use snake_case for variables and methods.

✅ Example:

```
customer_name = "John Doe"
```

Boolean Methods

- Use a ? at the end of method names.

✅ Example:

```
def active?
  status == "active"
end
```

Bang (!) Methods

- Use ! for methods that modify objects in place.

✅ Example:

```
user.save!  # Raises an error if save fails
```

📌 Summary Table

Component	Naming Convention
Models	Singular, CamelCase (User)
Database Tables	Plural, snake_case (users)
Controllers	Plural, CamelCase (UsersController)
Routes	Plural, snake_case (/users)
Migrations	CamelCase (CreateUsers), snake_case file (create_users.rb)
Associations	belongs_to :user (singular), has_many :orders (plural)
Views	Match controller actions (index.html.erb, show.html.erb)
Helpers	Plural, CamelCase (UsersHelper)
Mailers	Singular, CamelCase (UserMailer)
Jobs & Workers	SendEmailJob (ActiveJob), SendEmailWorker (Sidekiq)
Constants	UPPER_CASE_SNAKE_CASE
Variables & Methods	snake_case (customer_name)

Boolean Methods	Ends with ? (active?)
Bang Methods	Ends with ! (save!)

Metaprogramming in Ruby

Metaprogramming is a technique in Ruby where the code writes or modifies itself at runtime. It allows you to dynamically create methods, classes, and modify behavior, making Ruby extremely flexible.

---

1. Why Use Metaprogramming?

- Reduce Code Duplication – Generate methods dynamically instead of manually defining them.
- Improve Flexibility – Modify objects at runtime.
- Use DSLs (Domain Specific Languages) – Used in frameworks like Rails for defining routes (resources :users).

---

2. Example: Defining Methods Dynamically

Instead of writing multiple methods manually:

```
class Person
  def name
    "Alice"
  end

  def age
    30
  end
end
```

Use metaprogramming to create methods dynamically:

```ruby
class Person
  attr_accessor :name, :age  # Defines getter & setter methods
end

person = Person.new
person.name = "Alice"
puts person.name  # Output: Alice
```

Here, attr_accessor generates name and age getter/setter methods dynamically.

---

3. send Method (Calling Methods Dynamically)

Ruby allows calling methods dynamically using send.

```ruby
class Dog
  def bark
    "Woof!"
  end
end

dog = Dog.new
puts dog.send(:bark)  # Calls the bark method
```

Here, send(:bark) executes the method using a symbol.

---

4. method_missing (Handling Undefined Methods)

You can intercept undefined method calls and handle them dynamically.

```ruby
class Robot
  def method_missing(method, *args)
    puts "I don't know how to #{method}!"
  end
```

```
end

robot = Robot.new
robot.fly  # Output: I don't know how to fly!
robot.dance  # Output: I don't know how to dance!
```

Instead of throwing an error, method_missing catches any unknown method and provides a custom response.

---

5. Defining Methods Dynamically with define_method

Instead of manually defining multiple methods:

```
class Animal
  def dog
    "I am a dog"
  end

  def cat
    "I am a cat"
  end
end

puts :hello.object_id  # 123456
puts :hello.object_id  # 123456 (Same ID, no duplication)

puts "hello".object_id  # 987654
puts "hello".object_id  # 987659 (Different IDs, new allocation)
```

Use define_method to dynamically create methods:

```
class Animal
  [:dog, :cat, :lion].each do |animal|
    define_method(animal) { "I am a #{animal}" }
  end
end
```

```
animal = Animal.new
puts animal.dog  # Output: I am a dog
puts animal.cat  # Output: I am a cat
puts animal.lion # Output: I am a lion
```

Here, we loop over an array and dynamically define methods for each animal.

---

6. Opening Classes (Monkey Patching)

Ruby allows modifying existing classes at runtime.

```
class String
  def shout
    self.upcase + "!!!"
  end
end

puts "hello".shout  # Output: HELLO!!!
```

We added a shout method to the String class dynamically.

---

7. Using class_eval and instance_eval

Modify a class at runtime:

```
class Car
end

Car.class_eval do
  def drive
    "Vroom!"
  end
end
```

```
car = Car.new
puts car.drive  # Output: Vroom!
```

Here, class_eval dynamically adds a method to the Car class.

When to Use Metaprogramming?

✅ When you need to write DRY (Don't Repeat Yourself) code
✅ When working with DSLs (e.g., Rails routes, RSpec matchers)
✅ When dynamically creating methods based on external data

⚠️ Avoid excessive use – It can make debugging harder and reduce readability.

To Make API calls 🤙

Post:

```
uri = URI.parse(url)
header = { 'Content-Type' => 'application/json' }
http = Net::HTTP.new(uri.host, uri.port)
http.read_timeout = 240 # seconds
http.use_ssl = true
request = Net::HTTP::Post.new(uri.request_uri, header)
request.body = request_params.to_json
response = http.request(request)
result = eval(response.body)
```

```
uri.path → /posts
uri.request_uri → /posts?userId=1
uri =
URI.parse("https://jsonplaceholder.typicode.com/posts?userId=1")

puts uri.path          # "/posts"
```

```
puts uri.request_uri  # "/posts?userId=1"
```

Get:

```
url = URI.parse("https://jsonplaceholder.typicode.com/posts/1")
response = Net::HTTP.get(url)
puts JSON.parse(response)  # Converts JSON response to Ruby Hash
```

Threading 🧬

What is Threading in Ruby?

Threading allows a Ruby program to execute multiple operations concurrently within the same process. This means multiple tasks can run "at the same time" without blocking the execution of the main program.

Why Use Threads?
- To perform tasks concurrently (e.g., downloading multiple files at once).
- To avoid blocking the main thread (e.g., handling multiple HTTP requests).
- To improve performance in I/O-bound operations (e.g., database queries, file operations, network calls).

Creating Threads in Ruby
Threads in Ruby are created using the Thread.new method.

Example: Creating and Running Threads

```
threads = []

5.times do |i|
  threads << Thread.new do
    puts "Thread #{i} is running"
    sleep 1
    puts "Thread #{i} finished"
  end
end
```

```
threads.each(&:join)  # Wait for all threads to complete
puts "All threads finished!"
```

Explanation
- We create 5 threads using Thread.new.
- Each thread prints a message, sleeps for 1 second, and prints a completion message.
- threads.each(&:join) ensures the main program waits for all threads to finish before continuing.

GIL (Global Interpreter Lock) and Threading in Ruby
Ruby (MRI) has a Global Interpreter Lock (GIL) that prevents multiple threads from executing Ruby code simultaneously. This means:
- CPU-bound tasks (heavy computations) will NOT benefit from threading.
- I/O-bound tasks (HTTP requests, database queries, file I/O) CAN run concurrently.

Example: CPU-bound Task (Threads Are Not Helpful)

```
require 'benchmark'

def count
  x = 0
  10_000_000.times { x += 1 }
end

time = Benchmark.realtime do
  threads = 2.times.map { Thread.new { count } }
  threads.each(&:join)
end

puts "Execution Time: #{time.round(2)} seconds"
```

💡 Even with multiple threads, this runs slowly because the GIL prevents parallel execution of Ruby code.

Example: I/O-bound Task (Threads Are Helpful)

```
require 'net/http'
require 'benchmark'

urls = [
  "http://example.com",
  "http://example.org",
  "http://example.net"
]

time = Benchmark.realtime do
  threads = urls.map do |url|
    Thread.new { Net::HTTP.get(URI(url)) }
  end
  threads.each(&:join)
end

puts "Execution Time: #{time.round(2)} seconds"
```

💡 Here, HTTP requests run concurrently because the GIL is released during network calls.

Thread Synchronization:

Since threads run concurrently, race conditions can occur when multiple threads try to modify shared data.

Example: Race Condition (Incorrect Behavior)

```
counter = 0

threads = 10.times.map do
  Thread.new do
    1000.times { counter += 1 }
  end
end

threads.each(&:join)
puts "Final counter value: #{counter}" # Expected: 10,000 but may
be incorrect!
```

💡 Due to race conditions, the output might be incorrect.

Fixing Race Condition with Mutex
A Mutex (Mutual Exclusion Lock) ensures that only one thread modifies a shared resource at a time.

```
mutex = Mutex.new
counter = 0

threads = 10.times.map do
  Thread.new do
    1000.times do
      mutex.synchronize { counter += 1 }
    end
  end
end

threads.each(&:join)
puts "Final counter value: #{counter}" # Always 10,000
```

💡 Using mutex.synchronize, we ensure correct final counter value.

Thread Pools (Managing Multiple Threads Efficiently):

Creating too many threads can overload the system. Instead, we use a thread pool to limit the number of active threads.

Example: Thread Pool

```
require 'concurrent-ruby'

pool = Concurrent::FixedThreadPool.new(5)

10.times do |i|
  pool.post do
    puts "Task #{i} is running"
    sleep 1
  end
end

pool.shutdown
pool.wait_for_termination
puts "All tasks completed!"
```

💡 Here, we limit execution to 5 threads at a time instead of creating 10 threads.

When to Use Threads in Rails?
- Background tasks (but better to use Active Job or Sidekiq).
- Asynchronous API calls (fetching multiple resources simultaneously).
- Parallel database queries (if supported by the database driver).

When NOT to Use Threads in Rails?
- If you are performing CPU-intensive tasks.
- If you need database writes (Rails' Active Record is not thread-safe).
- If you need precise timing (use background jobs instead).

Alternatives to Threads in Rails

1. Active Job
For background jobs, use Active Job instead of raw threads:

```
class MyJob < ApplicationJob
  queue_as :default

  def perform
    puts "Running in the background!"
  end
end

MyJob.perform_later
```

2. Sidekiq (Recommended)
For concurrent background processing, use Sidekiq (which uses Redis and real parallelism).

```
class HardJob
  include Sidekiq::Worker

  def perform
    puts "Processing in the background!"
```

```
  end
end

HardJob.perform_async
```

💡 Sidekiq is faster and better than raw threads for Rails apps.

Summary:

✅ Threads allow concurrent execution.
✅ GIL (Global Interpreter Lock) prevents true parallel execution for CPU-bound tasks.
✅ Threads work well for I/O-bound operations (HTTP, DB, File I/O).
✅ Use Mutex to prevent race conditions.
✅ Use thread pools to limit active threads.
✅ For background jobs in Rails, prefer Active Job or Sidekiq.

Global Interpreter Lock

🔹 What is GIL (Global Interpreter Lock) in Ruby?

The Global Interpreter Lock (GIL) is a mutex (lock) in Ruby MRI (Matz's Ruby Interpreter) that prevents multiple threads from running Ruby code in parallel.

🔸 Why does Ruby have a GIL?
The GIL exists because Ruby's internal memory management (Garbage Collector, Object Allocation) is not thread-safe.
Without the GIL, multiple threads could corrupt shared data, leading to segmentation faults and crashes.

🔹 How GIL Affects Threading?
- CPU-bound tasks (e.g., calculations, sorting, encryption) are blocked by the GIL, making threads useless.
- I/O-bound tasks (e.g., network requests, file operations, database queries) release the GIL, allowing true concurrency.

🔸 Example: GIL Blocking CPU-bound Tasks
Even with multiple threads, the execution time remains almost the same for CPU-heavy tasks.

```
require 'benchmark'

def cpu_task
  sum = 0
  (1..100_000_000).each { |i| sum += i }
end

puts "Running CPU-bound task with threads..."

time = Benchmark.measure do
  t1 = Thread.new { cpu_task }
  t2 = Thread.new { cpu_task }
  t1.join
  t2.join
end

puts "Time taken: #{time.real} sec"
```

Expected Output

Time taken: 5.0 sec

💡 Even though we used two threads, the execution time doesn't improve because the GIL allows only one thread to run at a time.

🔸 Example: GIL is Released for I/O-Bound Tasks
When performing I/O operations, the GIL is released, allowing other threads to execute.

```
require 'net/http'
require 'benchmark'

URL = 'https://www.example.com'

def fetch_data
  Net::HTTP.get(URI(URL))  # HTTP request
end
```

```
puts "Running I/O-bound task with threads..."

time = Benchmark.measure do
  t1 = Thread.new { fetch_data }
  t2 = Thread.new { fetch_data }
  t1.join
  t2.join
end

puts "Time taken: #{time.real} sec"
```

Expected Output

Time taken: 1.2 sec

💡 Here, threading improves performance because the GIL is released while waiting for the HTTP response.

🔹 Can We Bypass the GIL?
Yes! Since Ruby MRI is single-threaded for CPU tasks, we can bypass the GIL using:
1. Multiple Processes (fork) - Run tasks in separate OS processes.
2. JRuby / TruffleRuby - These Ruby implementations don't have a GIL.
3. Parallel Gem - Uses fork internally for true parallelism.

Example: Using fork for True Parallel Execution

```
2.times do
  fork do
    puts "Process ID: #{Process.pid}"
    (1..100_000_000).each { |i| i * i }
  end
end

Process.waitall  # Wait for both processes to finish
```

➡️ Now, both processes run in parallel on multiple CPU cores! 🚀

Action Cable 🧵

1️⃣ Define Action Cable Clearly
Action Cable is a real-time communication framework in Rails that uses WebSockets to enable features like live chat, real-time notifications, and collaborative applications.

2️⃣ Explain Why Action Cable is Useful

Unlike traditional HTTP, which follows a request-response cycle, Action Cable maintains a persistent connection between the client and server. This makes it much faster for real-time applications.

💡 Example:
If you're building a chat app, you don't want users to refresh the page to see new messages. With Action Cable, new messages appear instantly because WebSockets keep an open connection between the server and all clients.

3️⃣ How Does Action Cable Work? (Step-by-Step)
Action Cable works by setting up a WebSocket connection and defining channels for communication. The main components are:

1️⃣ Connection → Authenticates users before they can use WebSockets.
2️⃣ Channels → Act like "rooms" where users subscribe and receive updates.
3️⃣ Broadcasting → Sends messages to all users subscribed to a specific channel.

4️⃣ Example Code
For example, let's say we are building a chat application. We create a ChatChannel where users send and receive messages.

📄 app/channels/chat_channel.rb (Server-side WebSocket logic)

```
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:room]}"
  end

  def receive(data)
    ActionCable.server.broadcast("chat_#{params[:room]}", data)
  end
```

```
end
```

🎯 Key Takeaway:
- Users subscribe to a room.
- When a message is sent, it is broadcasted to all users in that room.

📄 Frontend (JavaScript WebSocket Connection)

```
import consumer from "./consumer";

consumer.subscriptions.create({ channel: "ChatChannel", room: "1"
}, {
  received(data) {
    console.log("New message:", data);
    document.getElementById("messages").innerHTML +=
<p>${data.user}: ${data.message}</p>;
  }
});
```

🎯 Key Takeaway:
- The client listens for new messages and updates the chat window.

5️⃣ How Does Action Cable Scale?
Action Cable uses Redis as a message broker to distribute messages across multiple servers. This ensures scalability and avoids bottlenecks.

🔹 Example Scenario:
If our chat app has 100,000 users, we don't want a single server handling all WebSocket connections. Instead, Redis helps distribute the load among multiple servers.

6️⃣ When Would You Use Action Cable?
Action Cable is ideal for real-time applications like:
✅ Live chat applications (WhatsApp, Slack)
✅ Live notifications (Instagram DMs)
✅ Collaborative tools (Google Docs-style editing)
✅ Live dashboards (Stock market updates)

7️⃣ How to Deploy Action Cable?
For production, we can deploy Action Cable using Redis and a multi-threaded web server like Puma.

🎯 Key Takeaway:
Polling requires repeatedly asking the server for updates, while WebSockets allow real-time communication without extra requests.

Action Cable is Rails' built-in WebSocket framework that enables real-time communication. It allows users to send and receive messages instantly without refreshing the page. It works by establishing a persistent WebSocket connection and using channels to handle different types of communication. Action Cable scales well with Redis, making it suitable for live chat, notifications, and collaborative applications.

- How does Action Cable scale in production?
  Use Redis, multiple Puma workers, and load balancers.
- Can Action Cable work with React or Vue?
  Yes, we can connect using WebSockets on the frontend.
- How do you secure Action Cable connections?
  Authenticate users in connection.rb, restrict channel subscriptions.

Rack Middleware

1️⃣ What is Rack Middleware?
Rack Middleware is a layer that sits between the web server (like Puma/Unicorn) and the Rails application. It allows modifying requests before they reach the application and modifying responses before they are sent back to the client.

🔹 Think of it like a security guard at a building entrance:
- It processes incoming requests (e.g., logging, authentication).
- It modifies outgoing responses (e.g., compression, caching).

2️⃣ Why Do We Use Rack Middleware?
Rack Middleware helps handle tasks that apply to all requests globally, such as:
✅ Logging - Track requests & responses
✅ Authentication - Check user credentials
✅ Rate Limiting - Prevent abuse
✅ Compression - Reduce response size
✅ Session Management - Handle cookies/sessions
✅ CORS Handling - Manage cross-origin requests

3️⃣ How Does Rack Middleware Work?

Rack Middleware follows a chain-of-responsibility pattern.

🔹 When a request comes in:
1️⃣ It passes through multiple middleware layers before reaching Rails.
2️⃣ Rails processes the request and sends a response.
3️⃣ The response goes back through the middleware layers before reaching the client.

📌 Example Middleware Stack

Client → Rack Middleware (Logger) → Rack Middleware (Auth) → Rails App → Rack Middleware (Compression) → Client

4️⃣ Rack Middleware Code Example

📄 Custom Middleware Example (LoggerMiddleware)

```
class LoggerMiddleware
  def initialize(app)
    @app = app  # Stores the next middleware (or Rails app)
  end

  def call(env)
    puts "📌 Request received: #{env['REQUEST_METHOD']}
#{env['PATH_INFO']}"

    status, headers, response = @app.call(env)  # Call next
middleware/Rails app

    puts "✅ Response Status: #{status}"
    [status, headers, response]  # Return modified response
  end
end
```

🎯 Key Takeaways:
- The call(env) method is required in every middleware.
- @app.call(env) passes control to the next middleware.
- It modifies requests before passing them down and modifies responses before sending them back.

5️⃣ Adding Custom Middleware to Rails
To register middleware in a Rails app, add it in config/application.rb:

```
config.middleware.use LoggerMiddleware
```

Now, every request will be logged!

6️⃣ List of Default Middleware in Rails
Run this command to see all middleware:

```
rails middleware
```

7️⃣ How to Remove or Modify Middleware?

```
# Remove Middleware:

config.middleware.delete Rack::Sendfile

# Insert Middleware Before Another Middleware:

config.middleware.insert_before Rack::Runtime, LoggerMiddleware

# Insert Middleware After Another Middleware:

config.middleware.insert_after Rack::Runtime, LoggerMiddleware
```

8️⃣ When Would You Use Rack Middleware?
🔹 Scenario 1: Logging All Requests
- You need to track all incoming HTTP requests for debugging.
🔹 Scenario 2: Rate Limiting API Calls
- You want to prevent abuse by limiting requests per minute.
🔹 Scenario 3: Handling CORS
- You allow JavaScript apps from different domains to interact with your API.

Rack Middleware is a mechanism in Rails that processes HTTP requests before they reach the application and modifies responses before they go back to the client. It is commonly used for logging, authentication, caching, and compression. Middleware is registered in config.middleware, and custom middleware can be created by defining a call(env) method that processes the request.

Rack middleware follows a stack-based execution model:

1. Request enters middleware from top to bottom.
2. Rails app is called.
3. Response goes back up the stack in reverse order.

📌 Middleware Call Flow
Every Rack middleware class has a call(env) method.
This method is responsible for passing the request forward (down the stack) and handling the response on the way back.

```
class MiddlewareA
  def initialize(app)
    @app = app
  end

  def call(env)
    puts "MiddlewareA: Before Rails"
    status, headers, response = @app.call(env)   # Forward to next
middleware/Rails
    puts "MiddlewareA: After Rails"
    [status, headers, response]
  end
end

class MiddlewareB
  def initialize(app)
    @app = app
  end

  def call(env)
    puts "MiddlewareB: Before Rails"
    status, headers, response = @app.call(env)   # Forward to next
middleware/Rails
    puts "MiddlewareB: After Rails"
    [status, headers, response]
  end
end
```

🛠️ Middleware Stack Execution Order
Let's assume MiddlewareA → MiddlewareB → Rails App is the order in which middleware is added.

Step	Middleware Execution
1	MiddlewareA.call(env) (Before Rails)
2	MiddlewareB.call(env) (Before Rails)
3	Rails App runs (Processes request)
4	MiddlewareB.call(env) resumes (After Rails)
5	MiddlewareA.call(env) resumes (After Rails)
6	Response is returned to the client

> How do we know Rails runs first before Response Middleware?

When a request enters the system:
1. Middleware forwards the request down to Rails by calling @app.call(env).
2. Rails executes and processes the request.
3. After Rails responds, the middleware resumes execution in reverse order.

📌 Understanding @app.call(env)
The @app.call(env) is key here:
- Each middleware passes control to the next component (Rails or another middleware).
- Once Rails completes processing the request, middleware resumes in reverse order.

> Rack middleware follows a stack-based model (LIFO - Last In, First Out).
> Request flows down → Rails executes → Response flows back up.
> This ensures ResponseMiddleware modifies responses AFTER Rails has executed.

Rack-Based App

🚀 What is a Rack-Based App?

A Rack-based app is any Ruby web application that follows the Rack specification. It interacts with a web server (like Puma, Unicorn) using a common interface.

Rails is built on Rack and follows its interface!

📌 1. What is Rack?
✅ Rack is a lightweight interface that connects web servers (like Puma, WEBrick, Unicorn) to Ruby web applications (like Rails, Sinatra, etc.).

✅ It provides a simple API (call(env)) that every Rack-based app must implement.
✅ It ensures compatibility between web servers and Ruby frameworks.

📌 2. How a Rack-Based App Works
A Rack-based app follows a simple structure:
1. A web server (Puma/Unicorn) receives an HTTP request.
2. Rack converts this request into a Ruby Hash (env).
3. The Rack app processes the request and returns a response ([status, headers, body]).
4. The web server sends the response back to the client.

📌 3. Example: A Pure Rack App (No Rails, No Sinatra)
We can create a minimal web app using only Rack:

🔹 Step 1: Create config.ru

config.ru (Rack app entry point)

```
require 'rack'

class MyRackApp
  def call(env)
    [200, { "Content-Type" => "text/html" }, ["Hello, Rack!"]]
  end
end

run MyRackApp.new
```

🔹 Step 2: Run the Rack app

rackup

✔ Now, visit http://localhost:9292
✔ You'll see "Hello, Rack!" in the browser.

📌 4. Rack in Rails
✅ Rails is a Rack-based app.
✅ Rails uses Rack to handle HTTP requests via middleware.
✅ When you start a Rails app, config.ru bootstraps it:

Rails config.ru
require_relative 'config/environment'
run Rails.application

✔ Here, Rails.application is just another Rack-compliant app.

📌 5. Rack-Based Frameworks
✅ Since Rack standardizes web apps, multiple frameworks use it:
- Rails (built on Rack)
- Sinatra (lightweight alternative to Rails)
- Hanami (modular Ruby web framework)

> A Rack-based app is any web application that implements the Rack interface.
> It processes HTTP requests using call(env) and returns [status, headers, body].
> Rails, Sinatra, and Hanami are all Rack-based frameworks.

Rake Task in Rails

🚀 Rake Task in Rails
A Rake Task is a script that automates tasks in a Rails application, such as:
- Database migrations
- Cleaning logs
- Seeding data
- Running background jobs

Rails provides a built-in task runner using the rake (Ruby Make) tool.

📌 1. How to Create a Rake Task?
✅ Step 1: Create a .rake file inside lib/tasks/

mkdir -p lib/tasks
touch lib/tasks/custom_task.rake

✅ Step 2: Define the Rake Task
Inside lib/tasks/custom_task.rake:

```
namespace :greet do
  desc "Say Hello to the User"
```

```
  task :hello, [:name] => :environment do |task, args|
    puts "Hello, {args[:name] || 'Guest'}! Welcome to Rails."
  end
end
```

🔹 Breakdown:
- namespace :greet → Groups related tasks under greet.
- desc → Describes what the task does.
- task :hello, [:name] => :environment → A task named hello that accepts a name argument and loads the Rails environment.

📌 2. How to Run a Rake Task?
✅ Run Without Arguments

```
rake greet:hello
```

✔ Output: Hello, Guest! Welcome to Rails.

✅ Run With Arguments

```
rake greet:hello["John"]
```

✔ Output: Hello, John! Welcome to Rails.

> 💡 Tip: If using ZSH, wrap arguments in single quotes:

```
rake 'greet:hello[John]'
```

📌 3. Where to Find Built-in Rake Tasks?
✅ List All Available Rake Tasks

```
rake -T
```

✔ Example output:

```
rake db:migrate     Run database migrations
rake db:seed        Seed the database
rake log:clear      Clear log files
rake greet:hello    Say Hello to the User
```

You can use a normal Rails method instead of a Rake Task, but Rake Tasks are preferred for automating background tasks that are not part of the main web request-response cycle.

🚀 Why Not Just Use a Normal Rails Method?

A normal Rails method inside a controller or model works, but it has limitations:
1. ❌ You must trigger it manually (via UI, API call, or Rails console).
2. ❌ Long-running tasks will slow down your web app (e.g., deleting thousands of records).
3. ❌ It won't work outside the Rails web request cycle (e.g., scheduled tasks at midnight).
4. ❌ Hard to run in production without a request hitting the server.

💡 Example: Using a Normal Rails Method in a Controller

```
class UsersController < ApplicationController
  def delete_inactive_users
    cutoff_date = 6.months.ago
    deleted_count = User.where("last_login_at < ?",
cutoff_date).delete_all
    render json: { message: "#{deleted_count} users deleted" }
  end
end
```

🚨 Problems with This Approach:
- You need to call this via an API or UI action manually.
- A web request will hang if thousands of records are deleted (bad UX).
- No way to automate execution at night without human action.

✅ Why Use a Rake Task Instead?
- Runs outside web requests (doesn't slow down your app).
- Can be scheduled using cron jobs or whenever gem.
- Handles large data processing better.

- Works in production without API calls or UI interaction.

💡 Rake Task Version

```
namespace :cleanup do
  desc "Delete inactive users"
  task :inactive_users => :environment do
    cutoff_date = 6.months.ago
    deleted_count = User.where("last_login_at < ?",
cutoff_date).delete_all
    puts "#{deleted_count} inactive users deleted."
  end
end
```

📌 Now, just run it in the terminal:

rake cleanup:inactive_users

✔ Works without needing an API call!
✔ Can be scheduled to run automatically.

✨ Best Practice: Combine Rake Task + Rails Method
Sometimes, we define the logic inside a model and call it from both a Rake Task and a Controller.

💡 Step 1: Move Logic to the Model

```
class User < ApplicationRecord
  def self.delete_inactive_users
    cutoff_date = 6.months.ago
    where("last_login_at < ?", cutoff_date).delete_all
  end
end
```

💡 Step 2: Call It from a Rake Task

```
namespace :cleanup do
  desc "Delete inactive users"
  task :inactive_users => :environment do
```

```
    User.delete_inactive_users
    puts "Inactive users deleted!"
  end
end
```

💡 Step 3: Call It from a Controller Too

```
class UsersController < ApplicationController
  def delete_inactive_users
    User.delete_inactive_users
    render json: { message: "Inactive users deleted" }
  end
end
```

✔ Now you can run the logic in two ways:
1️⃣ Via the API:

curl -X DELETE http://localhost:3000/users/delete_inactive_users

2️⃣ Via a scheduled task:

```
   rake cleanup:inactive_users
```

> A normal Rails method runs within the request-response cycle and requires manual triggers, making it inefficient for background tasks. Rake Tasks are better for automation, scheduled jobs, and batch processing, as they run outside the web server, don't block users, and can handle large operations seamlessly. The best practice is to keep the core logic in a model and call it from both Rake and Controllers when needed.

You don't need to start the Rails server or Rails console to run a Rake task.

No, your Rails server does not need to be up for a Rake task that makes class method calls or database queries.

✅ Why Doesn't Rails Server Need to Be Up?

Rake loads the Rails environment (:environment) before execution.
This means ActiveRecord models and database connections are available.

The server (rails s) is only needed for handling HTTP requests, not for executing tasks directly.

load vs require

🔍 load vs require in Ruby

1️⃣require
The require method is used to load external libraries, gems, or Ruby files only once in a program. If the same file is required multiple times, Ruby ignores duplicate calls for efficiency.

✅ Key Features of require
✔ Loads only once (prevents redundant execution).
✔ Searches for files in default load paths ($LOAD_PATH).
✔ Commonly used for loading libraries and gems.

🚀 Example of require
1️⃣Using require to Load a Library

```
require 'json'  Loads Ruby's built-in JSON library

data = { name: "Alice", age: 25 }
puts JSON.generate(data)    => {"name":"Alice","age":25}
```

- Finds the json library inside Ruby's standard load paths.
- Loads it once, so it won't be reloaded again.

2️⃣Using require for a Custom Ruby File

```
require './my_script'   Loads my_script.rb (only once)
```

- require assumes the file has a .rb extension.
- If my_script.rb is already required, it won't be loaded again.

2️⃣load
The load method is used to execute a file every time it is called, even if it was loaded before.

✅ Key Features of load
✔ Loads every time (no caching).
✔ Requires the full file path (including .rb extension).
✔ Useful for reloading modified files dynamically.

🚀 Example of load
1️⃣ Using load to Execute a File Multiple Times

```
load './my_script.rb'   Executes my_script.rb
load './my_script.rb'   Executes again (not cached)
```

- Every load call re-executes the file.
- Requires the full file path (.rb must be included).

🚀 Example: When load is Useful
1️⃣ Imagine a Config File That May Change
You have a config.rb file:

config.rb

```
$CONFIG = { theme: "dark", language: "en" }
puts "Config loaded!"
```

Now, you use load in your main script:

```
load './config.rb'   Runs every time
load './config.rb'   Runs again, reloading the latest changes
```

- This is useful if config.rb is frequently updated, and you want changes to take effect immediately.
- require would only load it once and ignore future changes.

> "require loads a file only once and caches it, while load reloads the file every time it is called. require is used for loading libraries and gems, whereas load is useful for dynamically reloading files like config settings."

Why Can You Sometimes Use Classes/Methods Without require or load?

If you're able to use a class or method from another file without explicitly using require or load, there are a few possible explanations:

1️⃣ Rails Automatically Loads Certain Files (autoloading)

If you're working inside a Rails application, Rails automatically loads files inside the app/ directory.

Example: Using a Model Without require
Imagine you have a User model inside app/models/user.rb:

```
class User < ApplicationRecord
  def self.greet
    "Hello from User!"
  end
end
```

Now, inside rails console, you can just call:

puts User.greet

Output: "Hello from User!"

🔹 Why?
- Rails uses autoloading (via Zeitwerk in Rails 6+).
- It automatically loads classes from app/models/, app/controllers/, etc.
- So, you don't need to require models manually.

📌 Where This Won't Work?
If you try to use User in a pure Ruby script (script.rb outside Rails), it won't work unless you require Rails.

2️⃣ Rails Loads Initializers and lib/ via config/application.rb
- If your file is inside lib/, Rails does not autoload it by default.
- But if you configure it inside config/application.rb:

```
config.autoload_paths << Rails.root.join('lib')
```

Then, Rails will autoload files in lib/ too.

3️⃣ require_dependency in Development Mode
Rails uses require_dependency to reload files automatically in development.

Example:

Some controller
require_dependency 'lib/custom_service'

This allows changes in lib/custom_service.rb to be reloaded without restarting the server.

4️⃣ Already Required by Another File
Sometimes, a file you didn't write explicitly has already required the file you need.

Example
Let's say:
- post.rb requires user.rb
- You're working in post.rb
- Now, you can access User because it's already loaded.

5️⃣ IRB or Rails Console Loads the Entire Environment
If you're running:

rails console

It loads the entire Rails environment, meaning:
- All models, controllers, and initializers are already available.

But if you run a plain Ruby script:

ruby my_script.rb

Then, you need to explicitly require the files.

> In Rails, I can use classes without require because Rails autoloads files inside app/. It also preloads the entire environment in rails console, making models and helpers available automatically.

require_dependency in Rails
require_dependency is a Rails-specific method that ensures a file is reloaded in development mode, unlike require, which only loads it once.

🔹 Why Use require_dependency Instead of require?
1. require only loads once → Once a file is loaded, it won't be reloaded even if it changes.

2. require_dependency ensures reloading in development mode, so any changes to the file are applied automatically.

✅ Example 1: Using require (No Auto-Reload in Development)
Imagine you have a custom service file:

🔹 lib/custom_service.rb

```
class CustomService
  def self.greet
    "Hello from CustomService!"
  end
end
```

🔹 app/controllers/home_controller.rb

require 'lib/custom_service' Loads the file ONCE

```
class HomeController < ApplicationController
  def index
    render plain: CustomService.greet
  end
end
```

Issue with require:
- If you modify CustomService.greet and refresh the page, Rails won't reflect the changes.
- You must restart the server to see updates.

✅ Example 2: Using require_dependency (Auto-Reload Enabled)
🔹 Modify home_controller.rb:

```
require_dependency 'lib/custom_service'   Enables auto-reloading in development

class HomeController < ApplicationController
```

```
  def index
    render plain: CustomService.greet
  end
end
```

Why require_dependency Helps?
- If you modify lib/custom_service.rb, Rails detects changes and reloads the file.
- No need to restart the server.

🚀 When to Use require_dependency?
Use it when loading:
✅ Custom service classes (e.g., lib/)
✅ Non-auto loaded files (not inside app/)
✅ Manually required files that may change during development

> require_dependency is a Rails method that ensures a file is reloaded in development mode. Unlike require, which loads a file only once, require_dependency allows Rails to detect changes in custom files (like lib/) without restarting the server.

yield in Rails

1️⃣ yield in Ruby (Blocks)
yield is used inside a method to call a block that is passed to the method.

✅ Example: Yielding to a Block

```
def greet
  puts "Hello!"
  yield  # Calls the block given to the method
  puts "Goodbye!"
end

greet { puts "How are you?" }
```

🔹 Output:

Hello!
How are you?
Goodbye!

🔹 If no block is passed, calling yield will raise an error unless you check with block_given?.

2️⃣ yield in ERB (Rails View Rendering)
In Rails layouts (application.html.erb), yield is used to insert view content into the layout.

✅ Example: Basic yield in Layout
Layout (application.html.erb)

```
<!DOCTYPE html>
<html>
<head><title>MyApp</title></head>
<body>
  <h1>Header</h1>

  <%= yield %>  <!-- This will be replaced by the view's content
-->

  <h2>Footer</h2>
</body>
</html>
```

View (home/index.html.erb)

```
<p>Welcome to my site!</p>
```

🔹 Final Rendered Output:

```
<!DOCTYPE html>
<html>
<head><title>MyApp</title></head>
<body>
```

```
  <h1>Header</h1>

  <p>Welcome to my site!</p>

  <h2>Footer</h2>
</body>
</html>
```

3️⃣ yield with Named Blocks (content_for)
Named yield placeholders allow inserting specific content.

✅ Example: Using content_for
Layout (application.html.erb)

```
<head>
  <title>MyApp</title>
  <%= yield :head %>  <!-- Named yield -->
</head>
<body>
  <%= yield %>  <!-- Main content -->
</body>
```

View (home/index.html.erb)

```
<% content_for :head do %>
  <meta name="description" content="Welcome page">
<% end %>

<p>Welcome to my website!</p>
```

- Final Rendered Output:

```
<head>
  <title>MyApp</title>
  <meta name="description" content="Welcome page">
</head>
<body>
  <p>Welcome to my website!</p>
</body>
```

yield in Ruby is used to call a block inside a method, while yield in ERB is used in layouts to insert view content dynamically. In Rails, content_for allows us to define named yield placeholders for more flexibility."

🔥 How yield Works in ERB Rendering (Step-by-Step)

When a Rails request is processed, Rails renders the view and inserts it into the layout at the yield position.

Let's break it down step by step.

1️⃣ How Rails Decides What to Render
Whenever you call render inside a controller (or rely on Rails' automatic rendering), Rails does the following:

1. Finds the correct layout (default: application.html.erb).
2. Finds the corresponding view file (e.g., home/index.html.erb).
3. Renders the view and inserts it into the layout at yield.

2️⃣ Step-by-Step Rendering Process

✅ Controller: home_controller.rb

```
class HomeController < ApplicationController
  def index
    # Rails automatically renders home/index.html.erb
  end
end
```

✅ Layout: application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>MyApp</title>
</head>
<body>
  <h1>Header Section</h1>
```

```
  <%= yield %>  <!-- This is where the view content will be
inserted -->

  <h2>Footer Section</h2>
</body>
</html>
```

✅ View: home/index.html.erb

```
<p>Welcome to my website!</p>
```

🛠️ What Happens Internally?

1. User visits /home/index → HomeController#index is called.
2. Rails looks for the corresponding view → home/index.html.erb.
3. Rails renders the layout (application.html.erb).
4. Rails inserts the view's content (home/index.html.erb) at <%= yield %>.
5. Final HTML is sent to the browser.

3️⃣Final Rendered HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>MyApp</title>
</head>
<body>
  <h1>Header Section</h1>

  <p>Welcome to my website!</p> <!-- View content is inserted here
-->

  <h2>Footer Section</h2>
</body>
</html>
```

4️⃣ What If There's No Layout?
If there's no layout file, Rails only renders the view:

```
<p>Welcome to my website!</p>
```

✔️ No yield, no extra HTML from application.html.erb.

5️⃣ Multiple yield Blocks with content_for
If your layout has multiple yield placeholders, Rails fills them with content_for blocks from views.

✅ Example

```
 Layout: application.html.erb

<!DOCTYPE html>
<html>
<head>
  <title>MyApp</title>
  <%= yield :head %> <!-- Named yield -->
</head>
<body>
  <h1>Header Section</h1>

  <%= yield %>  <!-- Main content here -->

  <h2>Footer Section</h2>

  <%= yield :scripts %> <!-- Named yield for scripts -->
</body>
</html>

 View: home/index.html.erb

<% content_for :head do %>
  <meta name="description" content="Welcome page">
<% end %>

<p>Welcome to my website!</p>

<% content_for :scripts do %>
  <script>alert("Hello!");</script>
<% end %>
```

```
 Final Rendered HTML
<!DOCTYPE html>
<html>
<head>
  <title>MyApp</title>
  <meta name="description" content="Welcome page"> <!-- Inserted
via content_for -->
</head>
<body>
  <h1>Header Section</h1>

  <p>Welcome to my website!</p>

  <h2>Footer Section</h2>

  <script>alert("Hello!");</script> <!-- Inserted via content_for
-->
</body>
</html>
```

> yield in Rails ERB files is used in layout templates to define where content from views should be inserted. During rendering, Rails finds the correct layout, renders the view, and replaces yield with the view's content. We can also use content_for with named yield blocks to insert content into specific places like <head> or <footer>.

Layouts and Rendering in Rails

🚀 Layouts and Rendering in Rails
In Rails, layouts and rendering control how views are displayed, allowing for a structured UI.

1️⃣ What are Layouts in Rails?
Layouts are wrapper templates that define the overall structure of your webpage. Think of them as the "master page" where your views are inserted.

✅ Default Layout (application.html.erb)
- Rails automatically wraps views inside a layout (app/views/layouts/application.html.erb).
- The yield keyword determines where the view content will be inserted.

Example: application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>MyApp</title>
</head>
<body>
  <header>Header Section</header>

  <%= yield %>  <!-- Main view content gets inserted here -->

  <footer>Footer Section</footer>
</body>
</html>
```

Example: home/index.html.erb

```
<h1>Welcome to My App</h1>
<p>This is the home page.</p>

 Final Rendered Output:
html
<!DOCTYPE html>
<html>
<head>
  <title>MyApp</title>
</head>
<body>
  <header>Header Section</header>

  <h1>Welcome to My App</h1>
  <p>This is the home page.</p>

  <footer>Footer Section</footer>
</body>
</html>
```

2️⃣ How Does Rails Select Layouts?

- By default, Rails looks for app/views/layouts/<controller_name>.html.erb
- If no controller-specific layout exists, it uses application.html.erb

Custom Layout Example:
- If a controller explicitly defines a layout:

```
class AdminController < ApplicationController
  layout "admin"   Uses app/views/layouts/admin.html.erb
end
```

3️⃣ Rendering Views in Rails
Rendering means sending HTML (or JSON, XML, etc.) as a response.

✅ Types of Rendering
1. Automatic Rendering (Default Behavior)
  - If render isn't called, Rails automatically renders a template matching the action name.
  - Example: HomeControllerindex automatically renders app/views/home/index.html.erb.

2. Explicit Rendering (render)
  - You can manually specify which view to render.

```
class HomeController < ApplicationController
     def index
       render "custom_view"
     end
   end
```

  - Renders: app/views/home/custom_view.html.erb.

3. Rendering JSON/XML (For APIs)

  render json: { message: "Success", data: @user }

4. Rendering Partial Views
  - Partials are reusable UI components, stored as _filename.html.erb.

  <%= render "shared/navbar" %>

  - Renders app/views/shared/_navbar.html.erb.

4️⃣ Using content_for for Named Yields

Sometimes, we need multiple yield blocks for dynamic content.

Example: Defining content_for
Layout (application.html.erb)

```
<head>
  <title>MyApp</title>
  <%= yield :head %>  <!-- Named yield -->
</head>
<body>
  <%= yield %>  <!-- Main content -->
</body>
```

View (home/index.html.erb)

```
<% content_for :head do %>
  <meta name="description" content="Homepage">
<% end %>
```

<h1>Welcome to MyApp</h1>

- The <meta> tag will be inserted in <head> dynamically.

5️⃣ Redirect vs Render

Feature	render	redirect_to
Purpose	Renders a template	Redirects to a new URL
Request Type	Same request	New request (302 status)
Usage	render "home/index"	redirect_to root_path

> Layouts in Rails provide a consistent page structure using yield, while rendering determines how responses are displayed. We can override layouts, render partials for reusable UI, use content_for for named placeholders, and differentiate render (same request) vs redirect_to (new request).

render vs redirect_to in Rails

In Rails controllers, both render and redirect_to are used to control what happens after an action. However, they serve different purposes.

1️⃣ render
- Used to display a view without making a new request.
- Does not trigger a new HTTP request; it just renders a view template.
- The URL remains the same in the browser.

🔹 Example (Rendering a template from another action)

```
class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
    render :profile  # Renders app/views/users/profile.html.erb
  end
end
```

🔹 Example (Rendering JSON in API)

```
render json: { message: "Success" }, status: :ok
```

When to Use render?
✅ When you want to display a view without changing the URL.
✅ When returning JSON or other formats (in APIs).
✅ When handling errors within the same request.

2️⃣ redirect_to
- Used to send the user to a different URL.
- Triggers a new HTTP request (a 302 Found redirect by default).
- The browser's URL changes.

🔹 Example (Redirecting to another action)

```
class UsersController < ApplicationController
  def create
    @user = User.new(user_params)
    if @user.save
      redirect_to @user  # Redirects to show action
```

```
(user_path(@user))
    else
      render :new  # Stays on the same page
    end
  end
end
```

- Example (Redirecting with Flash Messages)

```
redirect_to root_path, notice: "Welcome back!"
```

When to Use redirect_to?
✅ When you need to redirect the user to a different page.
✅ After a successful form submission (e.g., create, update).
✅ When following PRG (Post-Redirect-Get) pattern.

🚀 Summary
- Use render when staying on the same page (e.g., showing errors).
- Use redirect_to when sending the user somewhere else.

Method Binding

Method binding in Ruby refers to how methods are associated with objects and how they can be called, stored, or dynamically reassigned. Let's break it down in an easy-to-understand way.

1️⃣ What is Method Binding?
Every method in Ruby is associated with an object, meaning:
- Bound Methods → Methods that are tied to an object (most methods are like this).
- Unbound Methods → Methods that are detached from an object and can be assigned to another object.

2️⃣ Normal Method Binding (Regular Methods)
By default, when you define a method inside a class, it is bound to objects of that class.

```
class Car
  def start
    "Car is starting!"
  end
end

car = Car.new
puts car.start  # => "Car is starting!"
```

Here, start is bound to the car object.

3️⃣ Unbinding Methods (Using unbind)
You can detach a method from an object (unbind it) and bind it to another object.

```
class Car
  def start
    "Car is starting!"
  end
end

car1 = Car.new
car2 = Car.new

# Get an UnboundMethod
unbound = Car.instance_method(:start)

# Bind it to another object
bound = unbound.bind(car2)

puts bound.call  # => "Car is starting!"
```

📝 Why use unbind?
- It allows sharing methods between objects dynamically.
- Helpful in metaprogramming, where methods need to be reused without copying code.

4️⃣ Storing Methods as Objects (method)

Ruby lets you store a method as an object using .method.

```
class Car
  def start
    "Car is starting!"
  end
end

car = Car.new

method_obj = car.method(:start)  # Get a Method object
puts method_obj.call  # => "Car is starting!"
```

📝 Why use .method?
- Allows you to store methods and call them later.
- Useful when passing methods as arguments in functional programming.

5️⃣ Defining Methods Dynamically (define_method)
Ruby allows defining methods at runtime using define_method.

```
class Car
  define_method(:start) do
    "Dynamically defined start method!"
  end
end

car = Car.new
puts car.start  # => "Dynamically defined start method!"
```

📝 Why use define_method?
- Useful when method names are not known in advance.
- Helps reduce repetitive code.

```
class Car
  private
  def secret_code
    "1234"
```

```
  end
end

car = Car.new

puts car.send(:secret_code)        # ✅ Works (bypasses access
control)
puts car.method(:secret_code)      # ❌ Error (can't access private
method)
```

⚠️ Is using send to access private methods a good practice?
🚨 No, it's not a good practice!
- send is powerful, but bypassing private methods violates encapsulation.
- Use it only for debugging or metaprogramming when absolutely necessary.

Summary:

1️⃣ Method binding controls how methods are associated with objects.
2️⃣ Methods are bound by default but can be unbound (unbind) and rebound.
3️⃣ You can store methods as objects using .method and call them later.
4️⃣ define_method allows dynamic method creation.
5️⃣ send can bypass access controls, but it's not recommended in production code.

What is an Abstract Class in Rails?

The line self.abstract_class = true is used in Rails models to indicate that a class is an abstract class. Let's break it down in detail:

An abstract class in Rails is a class that is meant to serve as a base class for other models but does not have a direct corresponding table in the database. In other words, it doesn't have its own database table and is not instantiated directly. Instead, other models inherit from this abstract class to reuse its functionality and share common behavior.

What Does self.abstract_class = true Do?

- self.abstract_class = true is a class-level method call that tells Rails that this model should not be mapped to a database table.
- This means Rails won't attempt to create or look for a table with the same name as the model class.

- The class can be inherited by other models that do have a corresponding database table, allowing those subclasses to inherit logic from the abstract class while having their own tables.

Use Case for Abstract Classes

Abstract classes are useful when you have common logic, behavior, or configuration that needs to be shared across multiple models, but you don't want the abstract class itself to be represented by a table in the database.

Example Scenario:

Imagine you are building a reporting system where different types of reports share a lot of the same behavior but should be stored in different tables. You might create an abstract class Report that contains the shared logic, and then subclass it to create specific report types.

Example:

```
class Report < ActiveRecord::Base
  self.abstract_class = true

  # Common logic for all reports
  def generate_report
    puts "Generating the report..."
  end
end
```

Now, any model that inherits from Report will share the common generate_report method but won't try to use the Report model itself to create a table.

```
class SalesReport < Report
  # Custom behavior for SalesReport
  def generate
    # Custom logic for generating sales report
    puts "Generating sales report..."
  end
end
```

```
class InventoryReport < Report
  # Custom behavior for InventoryReport
  def generate
    # Custom logic for generating inventory report
    puts "Generating inventory report..."
  end
end
```

- SalesReport and InventoryReport will inherit the generate_report method from Report.
- Both SalesReport and InventoryReport will not have a reports table in the database, because Report is abstract.

Behavior in Rails:
- Abstract Class: A class marked as abstract_class = true will not be mapped to a table in the database. This means no migrations or schema generation for that class.
- Subclasses: Subclasses of the abstract class can have their own tables, and they will inherit the methods and functionality of the abstract class.
- Inheritance: You can write shared methods, validations, and logic in the abstract class, and then all child models will have access to this shared functionality.

Summary:

- self.abstract_class = true marks a model as abstract, meaning it will not have its own table in the database.
- The class is used to define shared behavior or logic for other models that inherit from it.
- The subclasses of the abstract class can still have their own database tables and functionality, while inheriting common code from the abstract class.

Creating a Singleton Method in Ruby

A singleton method is a method defined for a single object, not for all instances of a class.

1. Using def obj.method_name (Basic Singleton Method)
You can create a singleton method for an object by defining a method directly on it:

```
obj = Object.new

def obj.say_hello
  puts "Hello, I am a singleton method!"
end

obj.say_hello  # ✅ Works

another_obj = Object.new
another_obj.say_hello  # ❌ Error: undefined method say_hello
```

🔹 Here, say_hello is only available for obj.

2. Using class << obj (Accessing Singleton Class)
You can define multiple singleton methods inside an object's singleton class:

```
obj = Object.new

class << obj
  def method_one
    puts "Method one in the singleton class!"
  end

  def method_two
    puts "Method two in the singleton class!"
  end
end

obj.method_one  # ✅ Works
obj.method_two  # ✅ Works
```

🔹 class << obj opens the singleton class of obj, allowing multiple singleton methods.

3. Singleton Methods on a Class (Class Methods)
Classes are objects too! Defining def self.method creates a singleton method for the class itself:

```
class Sample
  def self.class_method
    puts "I am a singleton method on the Sample class!"
  end
end

Sample.class_method  # ✅ Works
Sample.new.class_method  # ❌ Error: undefined method class_method
```

- class_method is only available on Sample, not on instances.

4. Using define_singleton_method (Dynamic Singleton Methods)
You can define a singleton method dynamically using define_singleton_method:

```
obj = Object.new

obj.define_singleton_method(:greet) do
  puts "Hello from a dynamically defined singleton method!"
end

obj.greet  # ✅ Works
```

- This is useful when dynamically adding methods at runtime.

What is a Rails Engine? 🚂

A Rails Engine is a mini Rails application that can be mounted inside a larger Rails app. It allows developers to package functionality (models, controllers, views, routes, and assets) into a reusable component.

❓Why Use a Rails Engine?
✅ Modularization – Break a large app into smaller, independent components.
✅ Code Reusability – Share business logic between multiple applications.
✅ Third-party Integrations – Many gems (like Devise, ActiveStorage) are built as engines.
✅ Encapsulation – Keep a part of the application self-contained.

2️⃣ Types of Rails Engines

🔹 Full Engine → Works like a standalone Rails app (includes everything: models, controllers, views).

🔹 Mountable Engine → Designed to be embedded into another Rails app with its own routes.

3️⃣ Creating a Rails Engine

Step 1: Generate an Engine

```
rails plugin new my_engine --mountable
```

This creates a new folder my_engine/ with the structure of a Rails engine.

Step 2: Understanding the Engine Structure

```
my_engine/
├── app/
│   ├── controllers/
│   ├── models/
│   ├── views/
├── lib/
│   ├── my_engine.rb         Entry point
│   ├── my_engine/engine.rb  Engine class
├── config/
│   ├── routes.rb            Engine-specific routes
```

- lib/my_engine/engine.rb → Defines the engine.
- config/routes.rb → Defines the engine's routes.

Step 3: Defining Routes Inside the Engine

Modify my_engine/config/routes.rb:

```
MyEngine::Engine.routes.draw do
  resources :articles
end
```

Step 4: Mount the Engine in the Main App
Inside config/routes.rb of the main app:

```
Rails.application.routes.draw do
  mount MyEngine::Engine, at: "/blog"   The engine is now
accessible at /blog
end
```

Now, the engine's routes will be available as:

GET /blog/articles

4️⃣ Example: Using an Engine for Authentication
Imagine you want to create a custom authentication system as an engine so multiple apps can use it.

Creating the Engine

```
rails plugin new auth_engine --mountable
```

Defining User Authentication Routes in auth_engine/config/routes.rb

```
AuthEngine::Engine.routes.draw do
  resources :sessions, only: [:new, :create, :destroy]
end
```

Mounting the Engine in a Main Rails App
In config/routes.rb of the main app:

```
Rails.application.routes.draw do
  mount AuthEngine::Engine, at: "/auth"
end
```

Now, authentication is available at:

GET /auth/sessions/new

POST /auth/sessions
DELETE /auth/sessions/:id

5️⃣ Rails Engines in Real-World Use
Many popular Rails gems are built as engines:
- Devise – Authentication
- ActiveStorage – File storage
- Spree – E-commerce platform

Why Do We Need Gemfile.lock If Versions Are Already in Gemfile?

Yes, the Gemfile specifies gem versions, but it does not lock the exact installed versions. Here's why Gemfile.lock is still necessary:

1️⃣ Gemfile vs. Gemfile.lock – The Key Difference

Feature	Gemfile	Gemfile.lock
Defines dependencies?	✅ Yes	❌ No
Specify exact versions?	❌ No (unless hardcoded)	✅ Yes
Updated manually?	✅ Yes	❌ No (auto-generated)
Required for consistency?	🔸 Not strictly	✅ Yes (for teams)

2️⃣ Gemfile Only Defines Version Constraints

In Gemfile, you usually define version constraints, not the exact version:

```
gem 'rails', '~> 6.1'    Allows any 6.1.x version
gem 'devise'             Installs latest available version
```

This means two developers can install different versions if Gemfile.lock is missing.

3️⃣ Gemfile.lock Ensures Everyone Uses the Same Versions

When you run bundle install, Bundler resolves dependencies and locks exact versions in Gemfile.lock:

```
GEM
  specs:
    rails (6.1.4.1)
    devise (4.8.1)

DEPENDENCIES
  devise
  rails (~> 6.1)
```

Now, anyone who runs bundle install will get the exact versions listed here.

4️⃣ Example: What Happens Without Gemfile.lock?

Scenario:
- 🔹 Developer A installs gems and gets rails 6.1.4.1.
- 🔹 Developer B (without Gemfile.lock) installs gems and gets rails 6.1.5.

If rails 6.1.5 has breaking changes, Developer B's app may break.
✅ With Gemfile.lock, everyone stays on rails 6.1.4.1.

5️⃣ When Does Gemfile.lock Change?

- bundle install → Keeps the same versions if already installed.
- bundle update gem_name → Updates only that gem and updates Gemfile.lock.
- bundle update → Updates all gems and modifies Gemfile.lock.

6️⃣ Conclusion: Why You Need Gemfile.lock

✅ Prevents version mismatches across different environments.
✅ Ensures dependency consistency between developers and production.
✅ Makes deployments predictable (your app will run with the same gem versions).

Quick Guide to RSpec

1️⃣ What is RSpec?

RSpec is a testing framework for Ruby, primarily used for testing Rails applications. It follows a Behavior-Driven Development (BDD) approach, making tests easy to read and write.

2️⃣ Installation

Add to Gemfile:

gem 'rspec-rails', group: [:development, :test]

Run:

bundle install
rails generate rspec:install

This generates:
- .rspec – Config file
- spec/ – Directory for tests
- spec/spec_helper.rb – RSpec settings
- spec/rails_helper.rb – Rails-specific settings

3️⃣ Basic Structure
RSpec tests are written inside spec/.

Example: Testing a Calculator

```
RSpec.describe "Calculator" do
  it "adds two numbers" do
    expect(2 + 2).to eq(4)
  end
end
```

Key Syntax:
Syntax	Description
describe	Groups related tests
it	Defines a test case

| expect(...).to | Defines expected behavior |
| eq | Matcher for equality |

4️⃣ Testing Models
Models should be tested for validations, associations, and methods.

Example: User Model

```
RSpec.describe User, type: :model do
  it "validates presence of name" do
    user = User.new(name: nil)
    expect(user.valid?).to be false
  end
end
```

Common Matchers
Matcher	Description
eq	Checks equality
be_truthy / be_falsey	Checks boolean values
include	Checks for inclusion
match	Checks regex patterns

5️⃣ Testing Controllers
Controller tests ensure correct request handling.

```
RSpec.describe UsersController, type: :controller do
  describe "GET #index" do
    it "returns success response" do
      get :index
      expect(response).to have_http_status(:success)
    end
  end
end
```

6️⃣ Testing Requests (API Testing)

```
RSpec.describe "Users API", type: :request do
  it "returns a list of users" do
    get "/users"
    expect(response).to have_http_status(:ok)
  end
end
```

7️⃣ Testing Jobs

```
RSpec.describe EmailJob, type: :job do
  it "queues the job" do
    expect { EmailJob.perform_later }.to have_enqueued_job
  end
end
```

8️⃣ Testing Background Workers (Sidekiq, ActiveJob)

```
RSpec.describe HardWorker, type: :worker do
  it "executes perform" do
    expect { HardWorker.perform_async }.to change(HardWorker.jobs,
:size).by(1)
  end
end
```

9️⃣ Mocking & Stubbing

```
# Using double (Loose Mocking)

user = double("User", name: "John")
expect(user.name).to eq("John")


# Using instance_double (Strict Mocking)

user = instance_double(User, name: "John")
expect(user.name).to eq("John")
```

```
# Stubbing Methods

allow(user).to receive(:admin?).and_return(true)
```

🔟 Running Tests

Run all tests:

rspec

Run a specific test file:

rspec spec/models/user_spec.rb

Run a single test:

rspec spec/models/user_spec.rb:10

Useful links 🔗:

- https://rubyonrails.org/
- https://www.tutorialspoint.com/ruby-on-rails/rails-framework.htm
- https://gorails.com/