Day 2 (8/04/24)

Remaining Chapter 1

Associating Models

1. Comment belongs to one article
2. Article can have many comments

```
Class Comment < ApplicationRecord
        belongs_to :article
end

Class Article < ApplicationRecord
        Has_many: comments

        // add validations here
end
```

Adding a Route for Comments

```
Rails.application.routes.draw do
  root "articles#index"

  resources :articles do
        resources :comments
  end
end
```

Generate Controller

bin/rails generate controller Comments

Now the show page for Comments will include the comments portion as well.

```
<h1><%= @article.title %></h1>

<p><%= @article.body %></p>

<ul>
  <li><%= link_to "Edit", edit_article_path(@article) %></li>
  <li><%= link_to "Destroy", article_path(@article), data: {
                turbo_method: :delete,
                turbo_confirm: "Are you sure?"
                } %></li>
```

```
</ul>

<h2>Add a comment:</h2>
<%= form_with model: [ @article, @article.comments.build ] do |form| %>
  <p>
        <%= form.label :commenter %><br>
        <%= form.text_field :commenter %>
  </p>
  <p>
        <%= form.label :body %><br>
        <%= form.text_area :body %>
  </p>
  <p>
        <%= form.submit %>
  </p>
<% end %>
```

The controller code will now include a new function to create comments (by finding article and then accessing it's comments and using the create method with comment params). Also redirect back to article.

```
class CommentsController < ApplicationController
  def create
        @article = Article.find(params[:article_id])
        @comment = @article.comments.create(comment_params)
        redirect_to article_path(@article)
  end

  private
        def comment_params
        params.require(:comment).permit(:commenter, :body)
        end
end
```

The frontend will also include the list now:

```
<h2>Comments</h2>
<% @article.comments.each do |comment| %>
  <p>
        <strong>Commenter:</strong>
        <%= comment.commenter %>
  </p>

  <p>
```

```
        <strong>Comment:</strong>
        <%= comment.body %>
  </p>
<% end %>
```

Refactoring:
Should make another **partial** for the form + comment details and call it inside main html/erb.

Chapter 2 Active Record Basics

Active Record Pattern
Objects -> carry persistent data and behavior operating on the data

Object Relational Mapping
Connects rich objects of an application to tables in a relational database MS
(No need to write SQL directly)

- Represent models and their data.
- Represent associations between these models.
- Represent inheritance hierarchies through related models.
- Validate models before they get persisted to the database.
- Perform database operations in an object-oriented fashion.

Convention over Configuration
No configuration needed when following conventions of Rails, only needed when we can't follow standard convention

Naming Conventions
Model -> Singular + CamelCase = BookClub
Database -> Plural, snake_case = book_clubs

Schema Conventions
Foreign keys - These fields should be named following the pattern singularized_table_name_id
Primary keys - By default, Active Record will use an integer column named id

Created_at, updated_at, lock_version, type, (association_name)*type, (tabl*ename)_count

Create Active Rec Models

```
class Product < ApplicationRecord
end

p = Product.new
```

```
p.name = "Some Book"
puts p.name
```

Overriding Naming Conventions

Have to specify table then if not using convention in the record + yml file

```
class Product < ApplicationRecord
  self.table_name = "my_products"
end

class ProductTest < ActiveSupport::TestCase
  set_fixture_class my_products: Product
  fixtures :my_products
end
```

PRIMARY KEY

```
class Product < ApplicationRecord
  self.primary_key = "product_id"
end
```

(if not using default id)

CRUD Functionalities

Active Record objects can be created from a hash, a block, or have their attributes manually set after creation. The new method will return a new object while create will return the object and save it to the database.

```
user = User.create(name: "David", occupation: "Code Artist")

(Use new to instantiate without being saved):
user = User.new
user.name = "David"
user.occupation = "Code Artist"
```

Create -> also persists the resulting object to the database

```
user = User.new do |u|
  u.name = "David"
  u.occupation = "Code Artist"
end
```

READ

Get all -> users = User.all
Get first one -> user = User.first
Finding by an attribute - > d = User.find_by(name: “David”)

Ordering in reverse chronological order

users = User.where(name: 'David', occupation: 'Code Artist').order(created_at: :desc)

UPDATE

user = User.find_by(name: 'David')
user.name = 'Dave'
user.save

OR

user = User.find_by(name: 'David')
user.update(name: 'Dave')

If we want to update without callbacks / validations:

User.update_all max_login_attempts: 3, must_change_password: true

DELETE

user = User.find_by(name: 'David')
One -> user.destroy

By condition -> User.destroy_by(name: ‘David’)

All -> User.destroy_all

VALIDATIONS

save, create, update validate a model before persisting in db.
They return false if model invalid + don’t change db
Bang counterpart -> save!, create!, update!
ActiveRecord::RecordInvalid exception if model invalid

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

Callbacks

Attaching code to events in the life cycle of models. For example, something after creating, updating, destroying a record. (Maybe like triggers)

```
class User < ApplicationRecord
  after_create :log_new_user

  private
        def log_new_user
        puts "A new user was registered"
        end
end
```

Migrations

convenient way to manage changes to a database schema
Database-agnostic

To run the migration and create the table -> bin/rails db:migrate
To roll it back and delete the table -> bin/rails db:rollback

Chapter 3: Active Record Migrations

From documentation: “You can think of each migration as being a new 'version' of the database. A schema starts off with nothing in it, and each migration modifies it to add or remove tables, columns, or entries. Active Record knows how to update your schema along this timeline, bringing it from whatever point it is in the history to the latest version. Active Record will also update your db/schema.rb file to match the up-to-date structure of your database“

```
class CreateProducts < ActiveRecord::Migration[7.1]
  def change
        create_table :products do |t|
        t.string :name
        t.text :description

        t.timestamps
        end
  end
end
```

Timestamps -> adds created_at and updated_at (automatically)

Reversible

Tell Active Record how to reverse a certain DB action by reversible up/down blocks or change

```
class ChangeProductsPrice < ActiveRecord::Migration[7.1]
  def change
        reversible do |direction|
        change_table :products do |t|
        direction.up   { t.change :price, :string }
        direction.down { t.change :price, :integer }
        end
        end
  end
end

class ChangeProductsPrice < ActiveRecord::Migration[7.1]
  def up
        change_table :products do |t|
        t.change :price, :string
        end
  end

  def down
        change_table :products do |t|
        t.change :price, :integer
        end
  end
end
```

Generating Migrations

bin/rails generate migration AddPartNumberToProducts
This creates the below migration:

```
class AddPartNumberToProducts < ActiveRecord::Migration[7.1]
  def change
  end
end
```

Adding Column
“Add___To___”
To add new column part_number of string type:

bin/rails generate migration AddPartNumberToProducts part_number:string

Can add index using:
bin/rails generate migration AddPartNumberToProducts part_number:string:index

Can also add more than one

Remove Columns
“Remove__From__”
bin/rails generate migration RemovePartNumberFromProducts part_number:string

Migration generated is the one below:

```
class RemovePartNumberFromProducts < ActiveRecord::Migration[7.1]
  def change
        remove_column :products, :part_number, :string
  end
end
```

Create New Table
Use the keyword “CreateXXX”

We can create a table using table name + attributes

bin/rails generate migration CreateProducts name:string part_number:string

Associations By References
We can use the references column (belongs_to)

bin/rails generate migration AddUserRefToProducts user:references

It creates:

```
class AddUserRefToProducts < ActiveRecord::Migration[7.1]
  def change
        add_reference :products, :user, foreign_key: true
  end
end
```

Joining
We can produce a join table using keyword “JoinTable:

bin/rails generate migration CreateJoinTableCustomerProduct customer product

Model Generators (Same thing -> generate model)

$ bin/rails generate model Product name:string description:text

Passing Modifiers

Pass on command line -> by using curly braces + field type:

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

Writing Migrations

Creating Table

Products table with name column. Id will be auto included.
can access id using :primary_key
If no id needed, pass id:false
Database specific options in “:options:

```
create_table :products do |t|
  t.string :name
end
```

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

Create Join Table

Create_Join_Table creates has and belongs to many join table e.g. products + categories

```
create_join_table :products, :categories
```
(null is false by default so values are required to save a record to the table, can be overridden like below)
```
create_join_table :products, :categories, column_options: { null: true }
```

Change join table name:
```
create_join_table :products, :categories, table_name: :categorization
```

Changing Tables
change an existing table in place

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

E.g. here we remove desc + name, create a new string col part-number

Changing Columns
Table name, col name, update

```
change_column :products, :part_number, :text
```

Can use change_column_null for null constraint and change_column_defult for default constraint
```
change_column :products, :part_number, :text

change_column_null :products, :name, false
change_column_default :products, :approved, from: true, to: false
```

NOTE: applied to future transactions, any existing records do not apply

Column Modifiers

- **comment** Adds a comment for the column.
- **collation** Specifies the collation for a string or text column.
- **default** Allows to set a default value on the column. Use nil for NULL.
- **limit** Sets the maximum number of characters for a string column and the maximum number of bytes for text/binary/integer columns.
- **null** Allows or disallows NULL values in the column.
- **precision** Specifies the precision for decimal/numeric/datetime/time columns.
- **scale** Specifies the scale for the decimal and numeric columns, representing the number of digits after the decimal point.

References
creation of column acting as the connection between one or more associations
Add
add_reference :users, :role
OR
add_belongs_to :taggings, :taggable, polymorphic: true

This creates a role_id column in the users table along with index for column.
Remove
remove_reference :products, :user, foreign_key: true, index: false

Foreign Keys
Not required but good for referential integrity.
Add
add_foreign_key :articles, :authors

If the from_table column name cannot be derived from the to_table name, use the :column option
Use the :primary_key option if the referenced primary key is not :id

add_foreign_key :articles, :authors, column: :reviewer, primary_key: :email
Remove
let Active Record figure out the column name
remove_foreign_key :accounts, :branches

remove foreign key for a specific column
remove_foreign_key :accounts, column: :owner_id

Composite Keys
(more than one key for unique identification)

composite primary key by passing the :primary_key option to create_table with an array value

```
class CreateProducts < ActiveRecord::Migration[7.1]
  def change
        create_table :products, primary_key: [:customer_id, :product_sku] do |t|
        t.integer :customer_id
        t.string :product_sku
        t.text :description
        end
  end
end
```

Executing SQL
When needed:
Product.connection.execute("UPDATE products SET price = 'free' WHERE 1=1")

Reversible
specify what to do when running a migration and what else to do when reverting it
Up _> transformation wanted for schema, Down -> revert back the transformation (unchanged)

For example, to make and revert a view:

Change:

```
class ExampleMigration < ActiveRecord::Migration[7.1]
  def change
        create_table :distributors do |t|
        t.string :zipcode
        end

        reversible do |direction|
        direction.up do
        # create a distributors view
        execute <<-SQL
        CREATE VIEW distributors_view AS
        SELECT id, zipcode
        FROM distributors;
        SQL
        end
        direction.down do
        execute <<-SQL
        DROP VIEW distributors_view;
        SQL
```

```
        end
        end

        add_column :users, :address, :string
  end
end
```

Up Down:

```
class ExampleMigration < ActiveRecord::Migration[7.1]
  def up
        create_table :distributors do |t|
        t.string :zipcode
        end

        # create a distributors view
        execute <<-SQL
        CREATE VIEW distributors_view AS
        SELECT id, zipcode
        FROM distributors;
        SQL

        add_column :users, :address, :string
  end

  def down
        remove_column :users, :address

        execute <<-SQL
        DROP VIEW distributors_view;
        SQL

        drop_table :distributors
  end
end
```

Errors to prevent Reverts:
Irreversible actions can’t be reverted, use an error to show that.

```
ActiveRecord::IrreversibleMigration
```

Reverting to Previous Migration
Use the revert keyword
revert ExampleMigration

Running Migrations

To migrate to version 20080906120000 run:
$ bin/rails db:migrate VERSION=20080906120000

Rollback to last migration:
$ bin/rails db:rollback

Can also undo several migrations using STEP parameter:
bin/rails db:rollback STEP=3

Can also redo (back up again)
bin/rails db:migrate:redo STEP=3

Setup / Prep database

bin/rails db:setup (Creates of doesn't exist, or else do required stuff i.e. load schema, run pending migrations, load seed data etc if not done already)

Resetting Database

Drop database + set it up again: bin/rails db:reset

Running Specific Migrations
Can use the db:migrate:up and db:migrate:down comments

bin/rails db:migrate:up VERSION=20080906120000

Schema Dumping

Database schema is dumped to ensure copies exist.

When :ruby is selected, then the schema is stored in db/schema.rb
When the schema format is set to :sql, the database structure will be dumped using a tool specific to the database into db/structure.sql
To resolve these conflicts run bin/rails db:migrate to regenerate the schema file

Active Record and Referential Integrity

Validates : foreign key, uniqueness: true -> use validations to enforce integrity.
The :dependent option on associations allows models to automatically destroy child objects when the parent is destroyed.

Migrations and Seed Data

```
class AddInitialProducts < ActiveRecord::Migration[7.1]
  def up
        5.times do |i|
        Product.create(name: "Product ##{i}", description: "A product.")
        end
  end

  def down
        Product.delete_all
  end
end
```

Migrations can also be used to add or modify data.
-> useful for existing database that can't be destroyed and recreated e.g production

db/seeds.rb

Old Migrations
Can delete db/migrate/ to clear state