30 Rails: Seeding Data

> “If you want to grow a giant redwood, you need to make sure the seeds are ok, nurture the sapling, and work out what might potentially stop it from growing all the way along. Anything that breaks it at any point stops that growth.”
>
> — **Elon Musk**

Seed Data

We've added database records manually through the Rails Console, but in a development environment it's helpful to have lots of data to work with. It would be monotonous to add many records manually, so we'll programmatically add fake data to Bloccit.

Git

Create a new Git feature branch for this checkpoint. See **Git Checkpoint Workflow: Before Each Checkpoint** for details.

Generating Seed Data

Open `db/seeds.rb` and remove the commented lines. `seeds.rb` is a script (a small utility program) we can run to seed the database with test data. Add the following code:

db/seeds.rb

```
+ require 'random_data'

+ # Create Posts
+ 50.times do
  # #1
+   Post.create!(
  # #2
+     title:  RandomData.random_sentence,
+     body:   RandomData.random_paragraph
+   )
+ end
+ posts = Post.all
+
+ # Create Comments
  # #3
+ 100.times do
+   Comment.create!(
  # #4
+     post: posts.sample,
+     body: RandomData.random_paragraph
+   )
+ end
+
+ puts "Seed finished"
+ puts "#{Post.count} posts created"
+ puts "#{Comment.count} comments created"
```

At #1, we use `create!` with a *bang* (`!`). Adding a `!` instructs the method to raise an error if there's a problem with the data we're seeding. Using create without a *bang* could fail without warning, causing the error to surface later.

At #2, we use methods from a class that does not exist yet, `RandomData`, that will create random strings for `title` and `body`. Writing code for classes and methods that don't exist is known as "wishful coding" and can increase productivity because it allows you to stay focused on one problem at a time.

At #3, we call `times` on an `Integer` (a number object). This will run a given block the specified number of times, which is 100 in this case. The end result of calling `times` is similar to that of a loop, but in this use-case, it is easier to read and thus more idiomatic.

At #4, we call `sample` on the array returned by `Post.all`, in order to pick a random post to associate each comment with. `sample` returns a random element from the array every time it's called.

`RandomData`

`RandomData` does not exist, so let's create it. Create a file named `random_data.rb` in the `lib` (short for "library") directory:

Terminal

```
$ touch lib/random_data.rb
```

Open `random_data.rb` and add the following code:

```
  # #5
+ module RandomData
  # #6
+   def self.random_paragraph
+     sentences = []
+     rand(4..6).times do
+       sentences << random_sentence
+     end
+
+     sentences.join(" ")
+   end
+
  # #7
+   def self.random_sentence
+     strings = []
+     rand(3..8).times do
+       strings << random_word
+     end
+
+     sentence = strings.join(" ")
+     sentence.capitalize << "."
+   end
+
  # #8
+   def self.random_word
+     letters = ('a'..'z').to_a
+     letters.shuffle!
+     letters[0,rand(3..8)].join
+   end
+ end
```

At **#5**, we define `RandomData` as a module because it is a standalone library with no

dependencies or inheritance requirements. Modules help keep common functions organized and reusable throughout our application. Unlike classes, we can't instantiate or inherit from modules. Instead we use them as **mixins** to add functions to multiple classes.

At **#6**, we define `random_paragraph`. We set `sentences` to an array. We create four to six random sentences and append them to `sentences`. We call `join` on `sentences` to combine each sentence in the array, passing a space to separate each sentence. `join` combines each sentence into a full paragraph (as a string).

At **#7**, we follow the same pattern as we did in #6 to create a sentence with a random number of words. After we generate a sentence, we call `capitalize` on it and append a period. `capitalize` returns a copy of `sentence` with the first character converted to uppercase and the remainder of the sentence converted to lowercase.

At **#8**, we define `random_word`. We set `letters` to an array of the letters `a` through `z` using `to_a`. We call `shuffle!` on `letters` *in place*. If we were to call `shuffle` without the *bang* (`!`), then `shuffle` would return an array rather than shuffle in place. We `join` the zeroth through nth item in `letters`. The nth item is the result of `rand(3..8)` which returns a random number greater than or equal to three and less than eight.

Let's make `random_data.rb` accessible to all our of specs going forward. Adding it to `application.rb` autoloads any references to the `lib` directory used by our code:

```
  ...
      # Do not swallow errors in after_commit/after_rollback callbacks.
      config.active_record.raise_in_transactional_callbacks = true
+     config.autoload_paths << File.join(config.root, "lib")
    end
  end
```

Drop the database and run `seeds.rb` by typing:

Terminal

```
$ rake db:reset
Seed finished
50 posts created
100 comments created
```

Open the Rails console to randomly check some results:

Console

```
$ rails c

> p = Post.find 3
```

We called another `ActiveRecord` class method, `find`, on `Post` and passed it a value which represents a unique post id. `find` will return the instance (row) of post data which corresponds to an id of 3. You should see an output with a funny looking `title` and `body` as that's what `RandomData` created in `seeds.rb`. Run the following methods to view how many comments the given post has:

Console

```
> p.comments.count
```

`count` is an `ActiveRecord` method that can be called on an `ActiveRecord` relation. `p.comments` returns an `ActiveRecord` relation, so `count` is a valid method to call on it.

Type `exit` to exit the console.

In the following video, we demonstrate seeding:

Git

Commit your checkpoint work in Git. See **Git Checkpoint Workflow: After Each Checkpoint** for details. Then deploy to Heroku.

Recap

Concept	Description
Modules	A module is independent code that contains everything necessary to implement only one feature.
Seeding Data	Rails' seeding feature allows the database to be populated with initial data based on the contents of `seeds.rb`.

Create a new Git feature branch for this assignment. See **Git Checkpoint Workflow: Before Each Assignment** for details.

By running `rake db:seed`, new data is added to your database. Occasionally you'll seed unique data using `seeds.rb` without erasing or duplicating existing data. This concept is called **idempotence**.

Idempotent code can be run many times or one time with identical results. For example, `i = 4` is idempotent but `i = i + 2` is not.

When developing web apps, it's important to think about what code is idempotent because code might get run more than once, like when a user refreshes a page.

1. Add a post with a unique **title** and **body** to `seeds.rb`. Before creating a unique post, check whether it already exists in the database using the **title** and **body**. Only seed the post if it doesn't already exist. Use the `find_or_create_by` method.
2. Run `rake db:seed` a couple of times, confirm that your unique post has only been seeded once using the Rails console.
3. Repeat steps one and two to create a comment with a unique **body**. Your comment should use the unique post you created as its post reference. Using the Rails console, confirm that your unique comment is also only seeded once after running `rake db:seed` several

times.

Commit your assignment in Git. See **Git Checkpoint Workflow: After Each Assignment** for details. Submit your commit to your mentor.

Solution

Do not watch this video until after you've attempted to complete the assignment. If you struggle to complete the assignment, submit your best effort to your mentor *before watching a solution video*.

Seeding Data Solution

assignment completed

?