Rails Demo 2: Building a simple Blog app

So let's create a caveman blog app. Let's start by creating a new app.

1. We use the rails command to do this.

   ```
   rails new blog
   ```

2. Now let's use the scaffold generator to create our first entity. Note we use the capital letter singular version of the entity

   ```
   rails generate scaffold Post title:string article:text likes:integer status:integ
   er
   ```

3. Review briefly what this accomplishes (e.g. creates the MVC triad for Posts) as well as the db migration. Then run rake to cause the migration to execute.

   ```
   rake db:migrate
   ```

4. Now we can run the rails server and play around with the MVC triad that was created.

   ```
   rails server
   ```

5. Let's setup our Post model so it can deal with the enumerated types properly. We need to add the following line of code to the Post class in model/post.rb:

   ```
   enum status: [:published, :unpublished]
   ```

6. But let's say that we wanted the :unpublished value (which is actually 1) to be the default in the database, not 0 (or :published). We can do this by tweaking the migration that we generated and adding a default to the status field:

```
class CreatePosts < ActiveRecord::Migration
    def change
    create_table :posts do |t|
      t.string :title
      t.text :article
      t.integer :likes
      t.integer :status, default: 1

      t.timestamps null: false
    end
  end
end
```

7. Yikes, we already ran our migration. No problem, let's back off the migration and re-run it:

```
rake db:rollback STEP=1
```

8. Cool, now that we undid the previous migration, we can do it again with our updated migration.

```
rake db:migrate
```

9. However, we still have a problem, because our input forms are treating status as an integer value. Take a look at `views/_form.html.erb` for example. We can setup our form to handle the enumerated types by making several updates. First let's add an action to the `posts_controller.rb` . At the top we put this line:

```
before_action :set_statuses
```

   and on the bottom we put

```
def set_statuses
    @statuses = Post.statuses
end
```

10. Now we need to update the form code, replacing this line of code:

```
<%= form.number_field :status, id: :post_status %>
```

    with

```
<%= form.select :status, @statuses.keys, selected: @post.status %>
```

11. Now go ahead and visit the posts form and check out the enumerated type!

Using TDD to validate models

1. Let's look at how we can add some validation to the form. Let's do this via a TDD approach. Let's start by writing down some of our constraints in the README file!

```
Testing specifications for posts:
title: string
article: text
likes: integer
status: enum - published or unpublished

- title must be present
- title must be between 5 and 80 characters
- article must be present
- article must be between 20 and 600 characters
- likes must be positive
- status must be valid
```

2. This is how we want the Post model to behave. Now let's go ahead and write some tests that can test these particular specifications. First under `test/models/` we create `post_test.rb`.

```
require 'test_helper'

class PostTest < ActiveSupport::TestCase

 def setup
    @post = Post.create(title: "a title", article: "This is the actual text of ou
r article.  It can be rather long.", likes: 0, status: 1)
 end

 test "post must be valid" do
    assert @post.valid?
 end

end
```

3. Now that we have a test, we can actually run our tests with rake.

```
rake test
```

4. Notice that we pass all tests at this point, since we have not validation rules established for our model. Let's go ahead and add test routines for each of our model specifications:

```
test "title must be present" do
end

test "title must not be too short" do
end

test "title must not be too long" do
end

test "article must be present" do
end

test "article must not be too short" do
end

test "article must not be too long" do
end

test "likes must be postive" do
end

test "status must be valid" do
end
```

5. Now let's implement each one of these with the appropriate forms of the `assert` method:

```
test "post must be valid" do
    assert @post.valid?
end

test "title must be present" do
    @post.title = ""
    assert_not @post.valid?
end

test "title must not be too short" do
    @post.title = "aa"
    assert_not @post.valid?
end

test "title must not be too long" do
```

```
    @post.title = "a" * 81
    assert_not @post.valid?
end

test "article must be present" do
    @post.article = ""
    assert_not @post.valid?
end

test "article must not be too short" do
    @post.article = "aa"
    assert_not @post.valid?
end

test "article must not be too long" do
    @post.article = "a" * 601
    assert_not @post.valid?
end

test "likes must be postive" do
    @post.likes = -1
    assert_not @post.valid?
end

test "status must be valid" do
    invalid_statuses = [-10, -1, 2, 10]
    invalid_statuses.each do |is|
        begin
            @post.status = is
            assert false, "#{is} should be invalid"
        rescue
            assert true
        end
    end
end

test "status must be published or unpublished" do
    valid_statuses = [:published, :unpublished]
    valid_statuses.each do |is|
      begin
          @post.status = is
          assert true
      rescue
          assert false, "#{is} should be invalid"
      end
    end
end
```

6. Now if we run our tests again, many (but not all of them fail) since we have not yet implemented our model validation.

   ```
   rake test
   ```

7. Ok, that makes sense, so lets proceed to implement the validations in our post.rb model.

   title

   ```
   validates :title,  presence: true, length: {minimum: 5, maximum: 80}
   ```

   article

   ```
   validates :article, presence: true, length: {minimum: 20, maximum: 600}
   ```

   likes

   ```
   validates :likes, numericality: {greater_than_or_equal_to: 0}
   ```

8. Wild, now all of our model tests pass, but some of our generated controller tests do not! That is because the autogenerated fixtures do not meet our model validation constraints! In particular the article fields aren't long enough. Let's fix this by editing `test/fixtures/posts.yml`. Then run the tests again. Eureka! We now have a a great deal of test automation in place and accomplished all this without breaking a sweat.

9. Let's go ahead and try to run the web app and see how the validation works in our form. Pretty cool heh?

10. We can play around with validations in the rails console as well.

    ```
    rails console
    p = Post.create(title: "This is a valid title", article: "This is a very long article about absolutely nothing.", likes: 0)

    p.valid?
    p.title = "a" * 1000
    p.valid?
    p.errors.messages
    ```