Rspec Testing Assignment: William Maddock

1. Overview of Rspec and Its Importance in Testing

Introduction to RSpec:

RSpec is a popular testing framework for Ruby and Ruby on Rails applications, providing developers with a clear and expressive syntax to write tests. It is designed to support **Behavior-Driven Development (BDD)**, a methodology that focuses on defining how an application should behave from the user's perspective. With RSpec, you can write tests for your application's models, controllers, views, and APIs, ensuring that each part of the application works as expected.

2. Getting Started with Rspec in Rails

Setting Up Rspec: Provide a step-by-step guide on how to add Rspec to a Rails project.

- Add the following to your Gemfile:

```
group :development, :test do
  gem 'rspec-rails'
end
```

- Then, run the following commands:

```
bundle install
rails generate rspec:install
```

File Structure:

Request Specs: Located in spec/requests/, these specs are used to test HTTP requests to your application's endpoints. They simulate requests to the controllers and ensure that the responses are what you expect. In the context of the Students feature, the request specs are testing the various routes (index, show, create, update, delete) for the Students resource.

Example:
spec/requests/students_spec.rb

Model Specs: Stored in spec/models/, these specs test the models in your application. They ensure that your ActiveRecord models behave as expected, including validations, associations, and custom methods.

Example:
spec/models/student_spec.rb

Controller Specs (Deprecated): While Rails controller specs were commonly used in older Rails applications, they are now less popular, with request specs being preferred. Controller specs test the internal workings of the controller actions but are typically less comprehensive than request specs.

Example:
spec/controllers/students_controller_spec.rb

Feature Specs: These live in spec/features/ and test the application's functionality from the user's perspective. Feature specs involve simulating user actions (like visiting a page, filling out forms, etc.) to ensure the app behaves correctly in real-world scenarios.

Factories/Fixtures: Many tests will also rely on factories (defined using FactoryBot or similar libraries) or fixtures. These files can be found in spec/factories/ and contain blueprints for creating instances of models during the tests.

Helper Specs and Support Files: Additional support files, helpers, and shared examples may be included in spec/support/ or spec/helpers/.

3. Assigned Tests and Tasks

Pagination of Search Results (Optional):

This test involves ensuring that when users search for data (e.g., students, classes), the results are properly paginated.

Key points for testing pagination:

- Ensure that a set number of results are displayed on each page (commonly 10 or 20 results per page).
- Verify that there are navigation links/buttons to access the next or previous pages.
- Check edge cases where the search might return very few results or no results at all.
- Confirm that the results displayed on a specific page are accurate and do not overlap with results from other pages.

RSpec Tests for CRUD (Create, Read, Update, Delete) Functionality:

CRUD operations form the backbone of any application dealing with data management, and testing these functions ensures that the core business logic behaves as expected.

Create:

- Test that new data can be created successfully (e.g., creating a new student or record).
- Ensure that all necessary validations are checked before creating the record, such as mandatory fields.
- Handle invalid input cases (e.g., missing required fields) to ensure proper error messages are displayed.

Read:

- Verify that existing data can be read and displayed correctly.
- Ensure that the right data is fetched based on the user's query or filter (e.g., fetching student details by ID).

Update:

- Confirm that data can be updated successfully (e.g., modifying a student's details).
- Ensure proper validations when updating data (e.g., invalid input or duplicate values should not be allowed).
- Test the behavior when the user attempts to update non-existent records.

Delete:

- Test that records can be deleted and that the deletion is reflected in the database.
- Handle cases where the user attempts to delete a non-existent or invalid record.
- Verify that the correct confirmation or success messages are shown upon successful deletion.

4. Writing Rspec Tests for CRUD Functionality

Create a New Student (POST /students):

Ensure that a new student is created when valid parameters are provided. Test cases should verify error handling:

```
it "creates a new student and redirects" do
  expect {
    post students_path, params: { student: { first_name: "John", last_name: "Doe",
school_email: "john.doe@school.com", major: "Computer Science", graduation_date: "2025-
05-15" } }
  }.to change(Student, :count).by(1)

  expect(response).to have_http_status(:found) # Expect redirect after creation
end
```

Show a Student (GET /students/:id):

Test the display of a student's information when the student exists. Write a test for handling a non-existent student:

```
it "returns a 200 OK status and shows the student's details" do
  get student_path(student)
  expect(response).to have_http_status(:ok)
  expect(response.body).to include(student.first_name)
end
```

Edit/Update Student (PATCH /students/:id)

Tests for Updating a Student:

```
it 'updates a student and redirects to the student profile page' do

 student = Student.create(name: 'John Doe', email: 'john@example.com')


 patch "/students/#{student.id}", params: { student: { name: 'Jane Doe' } }


 student.reload

 expect(student.name).to eq('Jane Doe')

 expect(response).to redirect_to(student_path(student))

end
```

Test for Invalid Update (e.g., Missing Required Fields):

```
it 'does not update the student with invalid data and re-renders the edit page' do

 student = Student.create(name: 'John Doe', email: 'john@example.com')
```

```
  patch "/students/#{student.id}", params: { student: { name: '' } } # Invalid data

  student.reload
  expect(student.name).to eq('John Doe') # Name should remain unchanged
  expect(response).to render_template(:edit)
end
```

Delete a Student (DELETE /students/:id):

```
it 'deletes a student and redirects to the students list' do
  student = Student.create(name: 'John Doe', email: 'john@example.com')

  delete "/students/#{student.id}"

  expect(Student.exists?(student.id)).to be_falsey # Student should be deleted
  expect(response).to redirect_to(students_path)
end
```

Test for Deleting a Non-existent Student:

```
it 'handles deletion of a non-existent student' do
  delete "/students/9999" # Assuming this ID does not exist
  expect(response).to redirect_to(students_path)

  expect(flash[:alert]).to be_present # Optional: Ensure that a flash message is displayed

ends
```

5. Optional Pagination Testing

Test the pagination of search results, ensuring the correct number of results are shown on each page:

```
it "paginates results and returns the correct number of students per page" do
    get students_path, params: { page: 1, per_page: 10 }
    expect(response.body).to include("Showing 10 students")
end
```

Here is the same test only more in depth. I used this test in my portfolio:

```
it "paginates results and returns the correct number of students per page" do
  # Create 25 students for pagination test
  24.times do |i|
    Student.create!(first_name: "Student #{i + 1}", last_name: "Test", school_email: "test#{i +
1}@msudenver.edu", major: "Computer Science BS", graduation_date: "2025-05-15")
  end

  # Verify that the students were created
  expect(Student.count).to eq(25) # Ensure 25 students exist
end
```

7. Helpful Resources

Rspec Documentation: https://rspec.info/documentation/

Rails Guides on Testing: https://guides.rubyonrails.org/testing.html

Arc Toolkit: https://www.tpgi.com/arc-platform/arc-toolkit/