PRÁCTICA CALIFICADA 5
Desarrollo de Software

Cesar Jesus Lara Avila

Calagua Mallqui Jairo Andre

20210279F

Facultad de Ciencias, Universidad Nacional de Ingeniería

diciembre del 2023

PREGUNTAS:

1. **En las actividades relacionadas a la Introducción de Rails los métodos actuales del controlador no son muy robustos: si el usuario introduce de manera manual un URI para ver (Show) una película que no existe (por ejemplo /movies/99999), verás un menasje de excepción horrible. Modifica el método show del controlador para que, si se pide una película que no existe, el usuario sea redirigido a la vista Index con un mensaje más amigable explicando que no existe ninguna película con ese.**

   Para manejar una redirección más amigable si una película no existe, podemos modificar el método 'show' del controlador de esta manera:

```ruby
class MoviesController < ApplicationController
  def show
    @movie = Movie.find_by(id: params[:id])
    if @movie.nil?
      flash[:notice] = "La película que buscas no existe."
      redirect_to movies_path
```

   Esto redirigirá al usuario a la vista Index de películas con un mensaje de advertencia si la película no se encuentra.

2. **En las actividades relacionados a Rails Avanzado, si tenemos el siguiente ejemplo de código que muestra cómo e integra OmniAuth en una aplicación Rails:**

```
class SessionsController < ApplicationController
        def create
            @user = User.find_or_create_from_auth_hash(auth_hash)
            self.current_user = @user
            redirect_to '/'
          end
        protected
        def auth_hash
                request.env['omniauth.auth']
      end

end
```

   El método auth_hash tiene la sencilla tarea de devolver lo que devuellva OmniAuth como resultado de intentar autenticar a un usuario. ¿Por qué piensa que se colocó esta funcionalidad en su propio método en vez de simplemente referenciar request..eny['omniauth.auth'] directamente? Muestra el uso del script.

   Pienso que la razón de colocar la funcionalidad en 'auth_hash' en lugar de referenciar 'request.env[omniauth.auth]' directamente es para modularizar y mantener el código más limpio y legible.

Procedo a mostrar el uso del script:

```ruby
class SessionsController < ApplicationController
  def create
    @user = User.find_or_create_from_auth_hash(auth_hash)
    self.current_user = @user
    redirect_to '/'
  end

  protected

  def auth_hash
    request.env['omniauth.auth']
  end
end
```

Al tener 'auth_hash' como un método separado, hace que el código sea más legible y nos permite un mantenimiento más sencillo si en el futuro se necesitan realizar cambios en la forma en que se obtiene la información de autenticación.

3. **En las actividades relacionadas a JavaScript, siguiendo la estrategia del ejemplo de jQuery utiliza JavaScript para implementar un conjunto de casillas de verificación (checkboxes) para la página que muestra la lista de películas, una por cada calificación (G, PG, etcétera), que permitan que las películas correspondientes permanezcan en la lista cuando están marcadas. Cuando se carga la página por primera vez, deben estar marcadas todas; desmarcar alguna de ellas debe esconder las películas con la clasificación a la que haga referencia la casilla desactivada.**

Para implementar casillas de verificación que filtren películas por clasificación usaremos JavaScript para mostrar u ocultar las películas según las casillas marcadas, así:

```javascript
document.addEventListener("DOMContentLoaded", function() {
    const checkboxes = document.querySelectorAll('.filter');
    const movies = document.querySelectorAll('.movie');

    checkboxes.forEach(function(checkbox) {
      checkbox.addEventListener('change', function() {
        const selectedFilters = Array.from(checkboxes)
          .filter(chk => chk.checked)
          .map(chk => chk.value);

        movies.forEach(function(movie) {
          const rating = movie.getAttribute('data-rating');
          if (selectedFilters.includes(rating)) {
            movie.style.display = 'block';
          } else {
            movie.style.display = 'none';
          }
        });
      });
    });
  });
```

Este código funciona mostrando u ocultando películas en la lista en función de las casillas de verificación seleccionadas. Cada película tiene un atributo 'data-rating' que indica su clasificación, y las casillas de verificación tienen valores correspondientes a esas clasificaciones. Al marcar una casilla, se muestran las películas que tienen esa clasificación y se ocultan las que no la tienen tal y cual dice la pregunta.

4. **De la actividad relacionada a BDD e historias de usuario crea definiciones de pasos que te permitan escribir los siguientes pasos en un escenario de RottenPotatoes:**

Given the movie "Inception" exists
And it has 5 reviews
And its average review score is 3.5

Para escribir los pasos en un escenario de RottenPotatoes, creamos definiciones de paso, así:

```
Given(/^the movie "(.*?)" exists$/) do |movie_title|
  # LO QUE SIGUE DEL CÓDIGO
end

Given(/^it has (\d+) reviews$/) do |review_count|
  # LO QUE SIGUE DEL CÓDIGO
end

Given(/^its average review score is ([0-9]\.[0-9])$/) do |average_score|
  # LO QUE SIGUE DEL CÓDIGO
end
```

Nuestro primer Given es para crear la película con el título dado, el segundo es para agregar la cantidad especificada de reseñas a la película y el tercero es para establecer el puntaje promedio de la película.

5. **De la actividad relacionada a BDD e historias de usuario, supongamos que en RottenPotatoesss, en lugar de utilizar seleccionar la calificación y la fecha de estreno, se opta por rellenar el formulario en blanco. Primero, realiza los cambios apropiados al escenario. Enumera las definiciones de pasos a partir de Cucumber invocaría al pasar las pruebas de estos nuevos pasos.**

Definiciones de pasos:

```
Given(/^I am on the Add New Movie page$/) do
    #Navegar ala página para agregar una nueva película
end

When(/^I fill in the form with movie details$/) do
    #Llenar el formulario con los detalles de la película
end

Then(/^I should see the movie details on the Movie Details page$/) do
    #Verificar que los detalles de la pelóicula estén en la página de detalles
end
```

6. De la actividad relacionada a BDD e historias de usuario indica una lista de pasos como los de la siguiente figura:

```
Given /I have added "(.*)" with rating "(.*)"/ do |title, rating|
    steps %Q{
        Given I am on the Create New Movie page
        When  I fill in "Title" with "#{title}"
        And   I select "#{rating}" from "Rating"
        And   I press "Save Changes"
    }
end

Then /I should see "(.*)" before "(.*)" on (.*)/ do |string1, string2, path|
    steps %Q{Given I am on #{path}}
    regexp = /#{string1}.*#{string2}/m #  /m means match across newlines
    expect(page.body).to match(regexp)
end
```

Para implementar el siguiente paso:

When /I delete the movie: "(.*)"/ do |title|

```ruby
When(/^I delete the movie "(.*?)"$/) do |litle|
    #Lógica para eliminar la película con el título dado
    movie = Movie.find_by(title: title)
    movie.destroy if movie.present?
end
```

Este paso busca la película con el titulo proporcionado y la elimina de la base de datos si existe.

7. **Basándose en el siguiente fichero de especificaciones (specfile), ¿a qué métodos deberían responder las instancias de F1 para pasar las pruebas?**

```
require 'f1'
describe F1 do
    describe "a new f1" do
      before :each do ; @f1 = F1.new ; end
        it "should be a pain in the butt" do
        @f1.should be_a_pain_in_the_butt
      end
   it "should be awesome" do
     @f1.should be_awesome
   end
    it "should not be nil" do
      @f1.should_not  be_nil
  end
 it "should not be the empty string" do
    @f1.should_not == ""
  end
end
end
```

```ruby
claass F1
    def be_a_pain_in_the_but
        #Lógica para verificar si es un dolor de cabeza
    end

    def be_awesome
        #LO QUE SIGUE DEL CÓDIGO
    end
end
```

Las instancias de 'F1' responden a 'be_a_pain_in_the_butt' y 'be_awesome' para pasar las pruebas.

EJEMPLO DE PRODUCCIÓN DE CUCUMBER

Siguiendo las indicaciones tenemos:

features > step_definitions > movie_steps.rb

```ruby
# Add a declarative step here for populating the DB with movies.

Given /the following movies exist/ do |movies_table|
  movies_table.hashes.each do |movie|
    Movie.create! (title: movie['title'], rating: movie['rating'], release_date: Date.parse(movie['release_date']))
  end
end
```

Esto agregará películas directamente a la base de datos utilizando los datos proporcionados en las tablas de películas '.feature'.

Después de agregar este código al archivo, m aseguré de eliminar cualquier línea que tenga 'pending' para que los pasos estén completamente implementados.

RESOLUCIÓN DE CADA UNA DE LAS PREGUNTAS:

Resuelve cada una de las preguntas:

1. **Completa el escenario restrict to movies with PG or R ratings in filter_movie_list.feature. Puedes utilizar las definiciones de pasos existentes en web_steps.rb para marcar y desmarcar las casillas correspondientes, enviar el formulario y comprobar si aparecen las películas correctas (y, lo que es igualmente importante, no aparecen las películas con clasificaciones no seleccionadas).**

   Podemos completar el escenario de la siguiente manera:

```
25  Scenario: restrict to movies with "PG" or "R" ratings
26    And I check the "PG" checkbox
27    Then complete the rest of of this scenario
28    # enter step(s) to check the "PG" checkbox
29    And I check the "R" checkbox
30    # enter step(s) to check the "R" checkbox
31    And I uncheck the following ratings: G, PG-13
32    # enter step(s) to uncheck all other checkboxes
33    And I press "ratings_submit"
34    # enter step to "submit" the search form on the homepage
35    Then I should see "The Terminator"
36    # enter step(s) to ensure that PG and R movies are visible
37    And I should see "When Harry Met Sally"
38    # enter step(s) to ensure that PG and R movies are visible
39    And I should see "The Help"
40    # enter step(s) to ensure that PG and R movies are visible
41    And I should not see "Aladdin"
42    # enter step(s) to ensure that other movies are not visible
43    And I should not see "Chocolat"
44    # enter step(s) to ensure that other movies are not visible
45    And I should not see "Amelie"
46    # enter step(s) to ensure that other movies are not visible
47    And I should not see "2001: A Space Odyssey"
48    # enter step(s) to ensure that other movies are not visible
49    And I should not see "The Incredibles"
50    # enter step(s) to ensure that other movies are not visible
51    And I should not see "Raiders of the Lost Ark"
52    # enter step(s) to ensure that other movies are not visible
53    And I should not see "Chicken Run"
54    # enter step(s) to ensure that other movies are not visible
```

2. **Dado que es tedioso repetir pasos como When I check the "PG" checkbox, And I check the "R" checkbox, etc., crea una definición de paso que coincida con un paso como, por ejemplo: Given I check the following ratings: G, PG, R**

   Esta definición de un solo paso solo debe marcar las casillas especificadas y dejar las demás casillas como estaban.

   Podemos crear un paso que marque múltiples casillas de la siguiente manera:

```ruby
When(/^I check the following ratings: (.*)$/) do |ratings_list|
  ratings = ratings_list.split(', ')
  ratings.each do |rating|
    check("ratings_#{rating}")
  end
end
```

3. **Dado que es tedioso especificar un paso para cada película individual que deberíamos ver, agrega una definición de paso para que coincida con un paso como: "Then I should see the following movies".**

   Se puede agregar una definición de pasos para verificar la presencia de múltiples películas, así:

```ruby
Then(/^I should see the following movies$/) do |movie_list|
  movies = movie_list.raw.flatten
  movies.each do |movie|
    expect(page).to have_content(movie)
  end
end
```

4. **Para el escenario all ratings selected sería tedioso utilizar And I should see para nombrar cada una de las películas. Eso restaría valor al objetivo de BDD de transmitir la intención de comportamiento de la historia del usuario. Para solucionar este problema, completa la definición de paso que coincida con los pasos del formulario: Then I should see all the movies en movie_steps.rb.**

   Considera contar el número de filas en la tabla HTML para implementar estos pasos. Si ha calculado las filas como el número de filas de la tabla, puede usar la afirmación expect(rows).to eq value para fallar la prueba en caso de que los valores no coincidan.

```ruby
Then(/^I should see all the movies$/) do
  rows = page.all('#movies tr').size - 1 # Restamos el encabezado de la tabla
  expect(rows).to eq Movie.count
end
```

5. **Utiliza tus nuevas definiciones de pasos para completar el escenario con todas las calificaciones seleccionadas. Todo funciona bien si todos los escenarios en filter_movie_list.feature pasan con todos los pasos en verde.**

   Para completar el escenario con todas las calificaciones seleccionadas, hacemos:

```
56  Scenario: all ratings selected
57    # your steps here
58    Then complete the rest of of this scenario
59    Given the following movies exist:
60    | title                  | rating | release_date |
61    # Agrega las películas con sus respectivas clasificaciones
62    And I am on the RottenPotatoes home page
63    When I check the following ratings: G, PG, PG-13, R
64    And I press "ratings_submit"
65    Then I should see all the movies
```

6. **Dado que los escenarios en sort_movie_list.feature implican clasificación, necesitarás la capacidad de tener pasos que prueben si una película aparece antes que otra en la lista de salida. Cree una definición de paso que coincida con un paso como: Then I should see "Aladdin" before "Amelie". Utiliza:**

a. **page es el método Capybara que devuelve un objeto que representa la página devuelta por el servidor de aplicaciones. Puedes usarlo en expectativas como expect(page).to have_content('Hello World'). Más importante aún, puedes buscar en la página elementos específicos que coincidan con selectores CSS o expresiones XPath. Consulta la documentación querying de Capybara: https://github.com/teamcapybara/capybara**
b. **page.body es el cuerpo HTML de la página como una cadena gigante.**
c. **Una expresión regular podría capturar si una cadena aparece antes que otra en una cadena más grande, aunque esa no es la única estrategia posible.**

   Creamos una definición de pasos que verifique si una película aparece antes que otra en la lista:

```
1  Then(/^I should see "(.*?)" before "(.*?)"$/) do |movie1, movie2|
2      expect(page.body).to have_content(/#{Regexp.escape(movie1)}.*#{Regexp.escape(movie2)
3    end
```

   Esta definición utiliza las expresiones regulares para verificar si ‘movie1’ aparece antes que ‘movie2’ en el cuerpo HTML de la página.

7. **Utiliza la definición de paso que creaste anteriormente para completar los escenarios, ordenar películas alfabéticamente y ordenar películas en orden creciente según la fecha de lanzamiento en sort_movie_list.feature. Todo funciona bien son todos los pasos de todos los escenarios en ambos archivos de funcionalidades pasan a verde.**

   Podemos utilizar los siguientes pasos:

```
25   Scenario: sort movies alphabetically
26   When I follow "Movie Title"
27   # enter step(s) to sort movies alphabetically
28   Then I should see "2001: A Space Odyssey" before "Aladdin"
29   # enter step(s) to verify specific movie order
```

```
31   Scenario: sort movies in increasing order of release date
32   When I follow "Release Date"
33   # enter step(s) to sort movies by release date
34   Then I should see "Chicken Run" before "Raiders of the Lost Ark"
35   # enter step(s) to verify specific movie order
```

PREGUNTA:

1. **Describa uno o más patrones de diseño que podrían ser aplicados al diseño del sistema.**
   -MVC (Modelo-Vista-Controlador): Este patrón es fundamental en Rails. Divide la lógica de la aplicación en tres componentes principales: el Modelo (manejo de datos), la Vista (presentación) y el Controlador (lógica de negocios y manejo de solicitudes). Se aplica ampliamente en la mayoría de las aplicaciones Rails.

   - Patrón Repositorio: Permite separar la lógica de persistencia de datos. En Rails, ActiveRecord ya tiene este patrón implementado, pero para complejidades adicionales o para mejorar el modularidad, se pueden utilizar repositorios personalizados.

   - Patrón de Fábrica (Factory): Útil para la creación de objetos complejos o inicialización con datos predeterminados. Es esencial cuando se trabaja con pruebas y la generación de datos de prueba.

   - Inyección de Dependencias (Dependency Injection): Facilita la escritura de código flexible, permitiendo la sustitución de implementaciones concretas por otras. En Ruby, esto se puede lograr de varias maneras, como a través de módulos o clases.

2. **Dado un sistema simple que responde a una historia de usuario concreta, analice y elija un paradigma de diseño adecuado.**

   - Programación Orientada a Objetos (POO): Este paradigma es el núcleo de Ruby y Rails. Se centra en la encapsulación, la herencia y el polimorfismo para estructurar y organizar el código.

   - Programación Funcional: Aunque Ruby no es puramente funcional, se pueden aplicar técnicas funcionales como funciones de orden superior, composición y reducción para mejorar la legibilidad y la mantenibilidad del código.

3. **Analice y elija una arquitectura software apropiada que se ajuste a una historia de usuario concreta de este sistema. ¿La implementación en el sistema de esa historia de usuario refleja su idea de arquitectura?**

La elección de la arquitectura depende de las necesidades específicas de la historia de usuario. Por ejemplo, si la historia de usuario implica la gestión de datos complejos y la presentación de información en diferentes formatos, una arquitectura orientada a servicios o microservicios podría ser más adecuada (se mencionó en clase). Para historias que implican interacción y colaboración entre usuarios, una arquitectura centrada en la colaboración y la interactividad puede ser más efectiva.