Project Design: Behavior-Driven Development

1. Objective

The goal of this assignment is to refine the design of your proposed team project by applying Behavior-Driven Development (BDD) principles. Each member of the team will select a feature of your application, translate its requirements into multiple detailed user stories, and create a full set of acceptance tests that specify its behavior. In support of this, some of the project software may need to be implemented.

You will use **Cucumber** and **Capybara** to create these specifications. This exercise will force you to think through user interaction and application logic *before* writing any significant implementation code for your own project.

2. Background

Behavior-Driven Development is a software development process that bridges the gap between technical implementation and business requirements. By writing specifications in a natural, human-readable language, you ensure that your project's features are well-defined and meet user expectations from the start.

- Cucumber is a tool that reads plain-text specifications (written in Gherkin) and executes them as automated tests. See this link for a reference to the syntax. And this link for a 10-minute tutorial.
- Capybara is a library that simulates how a user interacts with your web application, allowing you to test the application from an external, user-centric perspective.

For this assignment, you will use these tools to create a detailed, testable design for a key parts of your application.

3. Your Tasks

Each member of the team is required to select **one core feature** from your proposed project, develop the user stories that are associated with the feature, and create the necessary files to specify and test its behavior.

Task 1: Define Your Feature with a User Story

Before writing any tests, you must clearly define the feature.

- **Create a Design Document:** In the docs directory of your project, each team member should create a new file named `<your_feature_name>.md`. One team member should

create a single .md file which links to all of the feature story files from the team. (See the slides from Chapter 7 for a reminder about user stories.)

- **Write the User Story:** Inside this file, write a user story for your chosen feature. The story should follow the standard format: "As a [type of user], I want to [perform some task] so that I can [achieve some goal]."
- **Define Acceptance Criteria:** Below the user story, list three to five specific, testable acceptance criteria. These will define what it means for the feature to be "complete" and will form the basis of your test scenarios. Consider your "happy" and "sad" paths as candidates to include.

Task 2: Outline the MVC Components

In the same `.md` file, briefly describe the Models, Views, **and** Controllers you anticipate needing to build this feature. This helps connect your BDD specification to the underlying MVC architecture.

- **Model(s):** What data is involved? What new models or model attributes will you need? (e.g., "A `Post` model with `title:string` and `content:text` attributes.")
- **View(s):** What new pages or page components will the user see? (e.g., "A `posts/new.html.erb` view with a form.")
- **Controller(s):** What new controller actions will be required to handle user requests? (e.g., "A `PostsController` with `new` and `create` actions.")

Task 3: Create the Feature File

Translate your user story and acceptance criteria into a Gherkin feature file.

1. **Create the file:** In your Rails project, create a new file under the `features/` directory (e.g., `features/creating_a_post.feature`).
2. **Write the specification:** Write scenarios that test your acceptance criteria. You should have at least one "happy path" (success) scenario and at least one failure or edge-case scenario.

Example Gherkin Structure (for a blog post feature):

```
Feature: Create a new blog post
  As a signed-in user
  I want to publish a new post
  So that I can share my thoughts with others

  Scenario: User creates a post with valid information
    Given I am a signed-in user
    And I am on the new post page
    When I fill in "Title" with "My First Post"
    And I fill in "Content" with "This is the content of my post."
```

```
    And I press "Create Post"
    Then I should be on the post's show page
    And I should see the message "Post was successfully created."
    And I should see the title "My First Post"

  Scenario: User fails to create a post without a title
    Given I am a signed-in user
    And I am on the new post page
    When I fill in "Content" with "This post has no title."
    And I press "Create Post"
    Then I should see an error message indicating the title is missing
```

Task 4: Generate and Implement Step Definitions

Translate the Gherkin steps into executable Ruby code using Capybara.

1. **Run Cucumber:** From your terminal, **`run cucumber`** to get the pending step definition snippets.
2. **Create the Step Definition File:** Create a new file for your steps (e.g., **`features/step_definitions/post_steps.rb`**).
3. **Implement the Steps:** Copy the snippets into your new file and replace the pending comments with Capybara actions.

Example Implementation Snippet (post_steps.rb):

```
Given("I am a signed-in user") do
  # This step requires you to create a user and simulate a login.
  # For now, you can model this with a user created in the test database.
  @user = User.create!(email: 'test@example.com', password: 'password')
  visit new_user_session_path
  fill_in 'Email', with: @user.email
  fill_in 'Password', with: @user.password
  click_button 'Log in'
end

Given("I am on the new post page") do
  visit new_post_path
end

Then("I should be on the post's show page") do
  # This is tricky before the model exists!
  # A good "Red" step might just check for the content.
  expect(page).to have_content("My First Post")
end

Then("I should see an error message indicating the title is missing") do
  expect(page).to have_content("Title can't be blank")
```

```
end
```

5. Deliverables

Commit and push the following files to your project repository:

1. **`<your_feature_name>.md`**: Containing your user story, acceptance criteria, and MVC component outline.
2. **`features/<your_feature_name>.feature`**: Your Gherkin feature file.
3. **`features/step_definitions/<your_steps>.rb`**: Your Ruby step definition file.
4. features/screenshots/<your_feature_name_N>.jpg: A screenshot (one or more, where N is a number) of your terminal after running Cucumber showing the failing test output for your scenarios.

6. Grading Rubric

Criteria	Does Not Meet Expectations (0-1)	Meets Expectations (2-3)	Exceeds Expectations (4-5)
User Story & Acceptance Criteria	The user story is unclear, incomplete, or does not have testable acceptance criteria.	A clear user story is provided with at least three testable acceptance criteria.	The story and criteria are exceptionally well-defined, clear, and thoughtfully cover the core of the feature.
MVC Design Outline	The MVC outline is missing or does not logically map to the feature described.	A reasonable outline of the necessary M, V, and C components is provided.	The MVC outline is detailed and demonstrates a strong understanding of how the feature will be implemented.
Feature File (Gherkin)	Scenarios are poorly written, do not match the acceptance criteria, or have syntax errors.	The .feature file is correctly formatted and its scenarios accurately reflect the defined acceptance criteria.	Scenarios are clean, expressive, and cover interesting edge cases beyond the minimum requirements.

Criteria	Does Not Meet Expectations (0-1)	Meets Expectations (2-3)	Exceeds Expectations (4-5)
Step Definitions (Ruby/Capybara)	The file is missing, contains syntax errors, or does not implement the feature steps with Capybara.	All steps are implemented, and Capybara is used correctly to describe user actions and assertions.	Step definitions are clean, concise, and demonstrate a strong command of the Capybara API.
Correctness & Completion	Deliverables are missing, or the tests do not run correctly.	All required files are submitted. Running Cucumber shows the tests failing for the correct reasons.	All deliverables are present, and the test output demonstrates a perfect setup for the development phase.

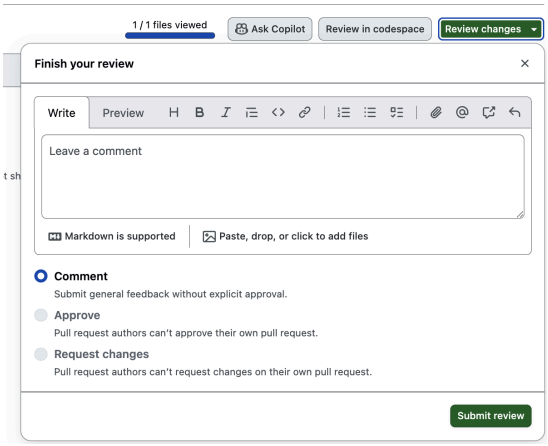
7. Grading Policy

While this assignment provides tasks for each team member, it is a "team" assignment. That is, all of the submitted files will be reviewed and one overall grade will be assigned (based on the above) rubric for all of the work. Members of the team will be expected to review each other's work and encourage completeness. Everyone contributes. If someone fails to do the work, then there will be a representative deduction for the grade across the board AND the team member(s) that do not complete the work will receive no grade.

In support of this policy, we will be looking additionally for the following:

1. Each team member should create their own files via their own branch in git.
2. Each team member should commit and push their own branch to the repo and create a pull request (PR) for code review. When creating the PR, use the "Reviewers" panel to tag one or more team members to request them to review your work.
3. Another team member should review your PR for correctness. Are the files complete? Are they in the correct directories? Are they named correctly? If not, reject the PR and ask the submitter to fix the issues, commit and push again. The PR will then require a review again.

   When reviewing another member's PR, you should review the files using the "File changed" tab and look at every file. See below.



To approve the review, use the "Review Changes" button (the green one shown above) to display the "Finish Your Review" window. Leave a comment (required) that acknowledges your review and if you approve the merge, select the correct radio button and then click "Submit your review". See image. You can also use this window to request changes to the files (in order to reject the PR's merge).

Once approved, the original code submitter can merge the PR to the main/master branch. We will be looking to see this process carried out for all team members as they add their files. This is real software development and how it works in industry. If you don't understand the process, please ask questions!

(Do you know how to update your local repo with all of the code that will be pushed by your peers? Find out! You'll want to update yo repo frequently.
To gather all changed files from the team members, and submits one commit/branch/PR it not allowed. You are learning to work collaboratively via git! Not sure how to do this? Ask!

Addendum

When you use **Cucumber** and **Capybara** in a Rails project, typically by using the **`cucumber-rails`** gem and running its generator (**`rails generate cucumber:install`**), the following key files and directories are created or configured:

Top-Level Directory

- **`features/`**: This is the main directory where your Cucumber tests reside.

Inside the `features/` Directory

- **Feature Files (`.feature`):**
  - Example: **`features/my_feature.feature`**
  - These files are written in **Gherkin**, a plain-text, structured language (using keywords like **`Feature`**, **`Scenario`**, **`Given`**, **`When`**, **`Then`**, and **`And`**). They describe the application's behavior from a user's perspective.

- **Support Files Directory (`support/`):**
  - **`features/support/env.rb`**: This is the main configuration file for Cucumber. It loads the Rails environment and typically includes:
    - The necessary **`require`** statements for Cucumber, Rails, and **Capybara**.
    - Configuration for Capybara (e.g., setting the default driver, like **`rack_test`** or a browser-based driver like Selenium or Cuprite, and the app host).
    - Setup for **Database Cleaner** (often included with **`cucumber-rails`**) to ensure a clean database state between scenarios.
  - **`features/support/paths.rb`**: This file contains helper methods to map user-friendly path names (used in feature files, like "the home page") to actual Rails routes (e.g., **`root_path`**).
  - Other files may be created here for **`World`** extensions, custom matchers, or specific setup for drivers.

- **Step Definitions Directory (`step_definitions/`):**
  - Example: **`features/step_definitions/web_steps.rb`**
  - These files contain the Ruby code that implements the steps defined in your feature files (the **`Given`**, **`When`**, **`Then`** lines). The code here uses **Capybara's DSL (Domain Specific Language)** methods (like **`visit`**, **`click_button`**, **`fill_in`**, and **`page.has_content?`**) to interact with the web application and make assertions.

Configuration Files

- **Gemfile:** The **`cucumber-rails`** and **`capybara`** gems (along with others like **`database_cleaner`**) are added, usually within the **`:test`** group, which is essential for installation.

- **Rake Task:** A new Rake task, typically **`rake cucumber`** or **`rake features`**, is configured to allow you to run the Cucumber suite easily from the command line.

In short, Cucumber establishes the **behavior (what to test)** in **`.feature`** files, and **Capybara** provides the tools used in the **step definitions (how to test)** to simulate user interaction with

the Rails application.