Basic Structure

> feature

`feature` accepts a string and is used to give an overview of the overall behavior.

```
feature 'Signing in' do
end
```

> scenario

`scenario` accepts a string and outlines a particular scenario of the feature.

```
feature 'Signing in' do
  scenario 'signs the user in successfully with a valid email and password' do
  end

  scenario 'notifies the user if his email or password is invalid' do
  end
end
```

Example Scenarios

```
feature 'Signing in' do
  scenario 'signs the user in successfully with a valid email and password' do
    create :user, email: 'person@example.com', password: 'password'
    visit sign_in_path
    fill_in 'Email', with: 'person@example.com'
    fill_in 'Password', with: 'password'
    click_button 'Sign In'
    expect(page).to have_css 'nav .greeting', text: 'Welcome, person@example.com'
  end

  scenario 'notifies the user if his email or password is invalid' do
    create :user, email: 'person@example.com', password: 'password'
    visit sign_in_path
    fill_in 'Email', with: 'person@example.com'
    fill_in 'Password', with: 'wrong password'
    click_button 'Sign In'
    expect(page).to have_css '.flash.notice', text: 'Invalid email or password'
  end
end
```

Refactoring Scenarios

Identify common code and extract to methods (or use `background`). By moving as much Capybara interaction to well-named methods, intention becomes clear and the test becomes more about behavior than syntax.

```
feature 'Signing in' do
  background do
    create :user, email: 'person@example.com', password: 'password'
  end

  scenario 'signs the user in successfully with a valid email and password' do
    sign_in_with 'person@example.com', 'password'
    user_sees_welcome_message 'Welcome, person@example.com'
  end

  scenario 'notifies the user if his email or password is invalid' do
    sign_in_with user.email, 'wrong password'
    user_sees_notice 'Invalid email or password'
  end

  def sign_in_with(email, password)
    visit sign_in_path
    fill_in 'Email', with: email
    fill_in 'Password', with: password
    click_button 'Sign In'
  end

  def user_sees_notice(text)
    expect(page).to have_css '.flash.notice', text: text
  end

  def user_sees_welcome_message(text)
    expect(page).to have_css 'nav .greeting', text: text
  end
end
```

Refactoring to Modules

Because RSpec acceptance tests are Ruby, modules can be included within RSpec to add methods. Extract methods which will be helpful in more than one feature.

```
# spec/features/sign_in_spec.rb
feature 'Sign in' do
  scenario 'signs the user in successfully with a valid email and password' do
    create :user, email: 'person@example.com', password: 'password'

    sign_in_with 'person@example.com', 'password'
    user_sees_welcome_message 'Welcome, person@example.com'
  end

  scenario 'notifies the user if his email or password is invalid' do
    create :user, email: 'person@example.com', password: 'password'

    sign_in_with 'person@example.com', 'wrong password'
    user_sees_notice 'Invalid email or password'
  end

  def user_sees_welcome_message(text)
    expect(page).to have_css 'nav .greeting', text: text
  end
end
```

```
# spec/support/features/sign_in_helpers.rb
module Features
  def sign_in_with(email, password)
    visit sign_in_path
    fill_in 'Email', with: email
    fill_in 'Password', with: password
    click_button 'Sign In'
  end
end
```

```
# spec/support/features/notice_helpers.rb
module Features
  def user_sees_notice(text)
    expect(page).to have_css '.flash.notice', text: text
  end
end
```

```
# spec/spec_helper.rb
RSpec.configure do |config|
  config.include Features, type: :feature
end
```

JavaScript-Enabled Tests

JavaScript can be enabled at the feature or scenario level.

```
feature 'Viewing comments', js: true do
end
# or
feature 'Viewing comments' do
  scenario 'displays pertinent information', js: true do
  end
end
```