**5** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

**5.1** Цели и задачи тестирования

Тестирование программного обеспечения – это процесс исследования программного продукта с целью выявления и последующее устранения несоответствий функционирования программного обеспечения

На протяжении всего этапа разработки программного продукта писалось большое количество тестов для максимального покрытия кода. Внедрению каждой новой функции предшествовало написание тестов, покрывающих возможные проблемы и проверяющих правильность работы этой функции.

Тестирование веб-приложения проводилось в два этапа:

* модульное тестирование;
* интеграционное тестирование.

Оба этапа являются достаточно важными, ни один из них невозможно исключить. Например, без модульного тестирования, при анализе работы программы в целом, будет происходить достаточное количество сбоев, выявить и локализовать которые может оказаться достаточно сложным заданием, в то время, как при анализе работы одного модуля, неисправность оказывается достаточно очевидной. И обратный случай, работоспособность каждого компонента в отдельности не гарантирует корректное поведение всей программы в целом.

Испытания web-приложения проводились на следующей машине:

* операционная система Ubuntu Server 14.04 LTS;
* 4-х ядерный процессор AMD;
* 8 Гб оперативной памяти;
* HDD 512GB.

**5.2** Тестирование web-приложения

Тестирование веб-приложения проводилось автоматически. Для данной задачи был выбрано расширение RSpec для фреймворка Ruby on Rails. Данное расширение позволяет проводить модульное и интеграционное тестирование по сценарию, который заранее определен пользователем.

**5.2.1** Модульное тестирование web-приложения

Модульное тестирование, или юнит-тестирование – процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок. Цель модульного тестирования – изолировать отдельные части программы и показать, что по отдельности эти части работоспособны. При разработке с использованием Ruby on Rail юнит-тесты обычно пишут для методов моделей, также юнит-тестами иногда покрывают специальные вспомогательные классы-хелперы, которые используются сразу несколькими моделями или контроллерами.

Модульное тестирование предполагает собой независимое тестирование каждой модели, которая входит в проект. Для этого использовались сценарии, описанный на языке Ruby. Модель User тестировалась следующим образом:

сlass ValidatableTest < ActiveSupport::TestCase

Проверяется будет ли объект модели валидным, если не заполнено поле email.

test 'should require email to be set' do

Создается пользователь с пустым email.

user = new\_user(email: nil)

assert user.invalid?

Модель должна быть не валидной, т.к поле email является обязательным.

assert user.errors[:email]

assert\_equal 'can\'t be blank', user.errors[:email].join

end

Проверяется функционал, который следит за корректным форматом электронного адрес.

test 'should require correct email format if email has changed, allowing blank' do

Создается пользователь с пустым адресом электронной почты.

user = new\_user(email: '')

assert user.invalid?

В результате не должна выдаться ошибка о том, что email некорректен, а то что он пуст.

assert\_not\_equal 'is invalid', user.errors[:email].join

Создаем пользователей с некорректным адресом электронной почты .

%w{invalid\_email\_format 123 $$$ () ☃}.each do |email|

user.email = email

assert user.invalid?, 'should be invalid with email ' << email

В результате должно выдаваться сообщение о некорректности адреса электронной почты.

assert\_equal 'is invalid', user.errors[:email].join

end

user.save(validate: false)

assert user.valid?

end

В ходе данного теста мы проверяем пользователей с корректными адресами эектроннай почты, в результате мы узнаем корректно система распознает формат или нет.

test 'should accept valid emails' do

%w(a.b.c@example.com test\_mail@gmail.com any@any.net email@test.br 123@mail.test 1☃3@mail.test).each do |email|

Создаем пользователя с указанным электронным адресом, и проверяем его на валидность.

user = new\_user(email: email)

assert user.valid?, 'should be valid with email ' << email

assert\_blank user.errors[:email]

end

end

Проверяется то, что пароль является обязательным параметром и без него объект модели валидным быть не может.

test 'should require password to be set when creating a new record' do

Имитируем создание объекта с пустым полем пароля.

user = new\_user(password: '', password\_confirmation: '')

assert user.invalid?

Ожидается, что система должна заметить ошибку и выдать соответствующее сообщение.

assert\_equal 'can\'t be blank', user.errors[:password].join

end

При регистрации необходимо ввести пароль и подтверждение пароля. В случае их несовпадения должна появиться ошибка.

test 'should require confirmation to be set when creating a new record' do

user = new\_user(password: 'new\_password', password\_confirmation: 'blabla')

assert user.invalid?

assert\_equal 'doesn\'t match Password', user.errors[:password\_confirmation].join

end

end

**5.2.2** Интеграционное тестирование web-приложения

Интеграционное тестирование – одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию. Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования. Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц – объединения, множества или группы модулей –выполняется через их интерфейс, с использованием тестирования «чёрного ящика» [16].

В итоге, полное покрытие тестами проекта заняло примерно треть от общего времени разработки всего программного средства, однако несмотря на такое большое количество потраченного времени, была достигнута большая стабильность и надёжность программного обеспечения.

В случае интеграционного тестирование проверяется работа системы в целом. Для примера рассмотрим тестирование аутенфикации.

class AuthenticationSanityTest < Devise::IntegrationTest

Проверяем, что домашняя страница должна быть доступна без аутенфикации в системе.

test 'home should be accessible without sign in' do

Заходим на домашнюю страницу.

visit '/'

Проверяем, что запрос выполнен успешно.

assert\_response :success

Проверяем, что нам вернулась та страница, которую мы ожидали.

assert\_template 'home/index'

end

Проверяем, что обычный пользователь не должен быть аутенфицирован как администратор.

test 'sign in as user should not authenticate admin scope' do

Регистрируем пользователя в системе.

sign\_in\_as\_user

Проверяем как пользатель аутенфицирован в системе.

assert warden.authenticated?(:user)

refute warden.authenticated?(:admin)

end

В данном случае тестируется обратная ситуация, когда администраттор не должен быть зарегистрирован как обычный пользователь.

test 'sign in as admin should not authenticate user scope' do

Регистрируем пользователя как администратора.

sign\_in\_as\_admin

Проверяем какую роль он получил после аутенфикации.

assert warden.authenticated?(:admin)

refute warden.authenticated?(:user)

end