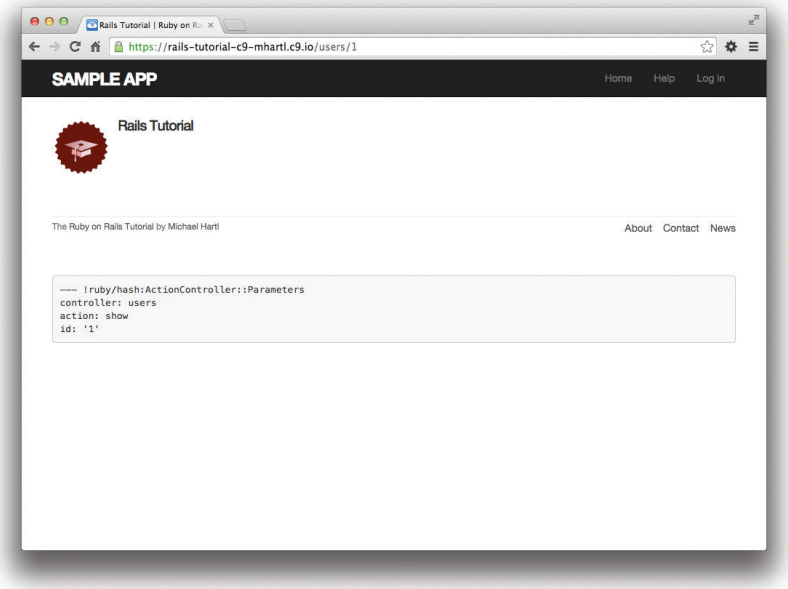

Рис. 7.23 ❖ Профиль пользователя с исчезнувшим кратковременным сообщением

7.4.4. Тесты для успешной отправки формы

Прежде чем продолжить, напишем тесты для сценария успешной отправки формы, чтобы проверить поведение приложения и предотвратить регрессии. По аналогии с тестами для сценария неудачной регистрации в разделе 7.3.4, основная цель – проверить содержимое базы данных. Сейчас мы проверяем ситуацию отправки допустимой информации, и нам нужно убедиться, что пользователь был создан. По аналогии с тестом в листинге 7.21, где использовалась инстукция:

```
assert_no_difference 'User.count' do
  post signup_path, ...
end
```

мы будем использовать здесь похожий метод `assert_difference`:

```
assert_difference 'User.count', 1 do
  post_via_redirect users_path, ...
end
```

Так же как `assert_no_difference`, метод `assert_difference` принимает в первом аргументе строку `'User.count'` и сравнивает значение `User.count` до и после выполнения блока. Второй (необязательный) аргумент определяет величину разности (в данном случае 1).

Заменив `assert_no_difference` на `assert_difference` в листинге 7.21, мы получим тест в листинге 7.26. Обратите внимание на использование варианта `post_via_`

redirect для передачи запроса post по маршруту users_path. Так мы организовали переадресацию к отображению шаблона 'users/show' после отправки формы. (Неплохо было бы написать тест для кратковременного сообщения, но я оставлю это вам в качестве самостоятельного упражнения (раздел 7.7).)

Листинг 7.26 ❖ Тест успешной регистрации **ЗЕЛЕНЫЙ**
(test/integration/users_signup_test.rb)

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest
.
.
.
  test "valid signup information" do
    get signup_path
    assert_difference 'User.count', 1 do
      post_via_redirect users_path, user: { name:  "Example User",
                                            email: "user@example.com",
                                            password:              "password",
                                            password_confirmation: "password" }
    end
    assert_template 'users/show'
  end
end
```

Обратите внимание, что тест в листинге 7.26 также проверяет отображение шаблона 'users/show' после успешной регистрации. Для прохождения этого теста необходима корректная работа маршрутов Users (листинг 7.3), метода show (листинг 7.5) и представления show.html.erb (листинг 7.8). Как результат строка

```
assert_template 'users/show'
```

надежно проверяет практически все, что имеет отношение к странице профиля пользователя. Подобный вид сквозного покрытия важных функций приложения показывает, почему интеграционные тесты так полезны.

7.5. Профессиональное развертывание

Теперь, когда у нас имеется действующая страница регистрации, самое время развернуть приложение на действующем сервере. Первые попытки развернуть приложение мы предприняли еще в главе 3, но сейчас впервые оно *действительно* умеет что-то делать, поэтому воспользуемся этой возможностью и развернем его как настоящие профессионалы. В частности, добавим в приложение важную функцию и сделаем процедуру регистрации защищенной, а также заменим первоначальный веб-сервер на тот, который подходит для реального использования.

В качестве подготовки к развертыванию объединим изменения с основной ветвью:

```
$ git add -A
$ git commit -m "Finish user signup"
$ git checkout master
$ git merge sign-up
```

7.5.1. Поддержка SSL

При отправке формы регистрации, разработанной в этой главе, имя, адрес электронной почты и пароль пересылаются по сети и, следовательно, могут быть перехвачены. Это серьезная брешь в системе безопасности приложения, а устранить ее можно при помощи поддержки защищенных сокетов (Secure Sockets Layer, SSL)[1] для шифрования всей уязвимой информации до того, как она покинет локальный браузер. Хотя мы можем использовать поддержку SSL только для страницы регистрации, проще будет задействовать ее по всему сайту, получив заодно дополнительные преимущества в виде защищенного входа пользователя на сайт (глава 8) и иммунитета к критичной уязвимости *session hijacking* (перехват сеанса), которую мы рассмотрим в разделе 8.4.

Чтобы включить поддержку SSL, достаточно раскомментировать одну строку в `production.rb`, конфигурационном файле для приложений, развертываемых в эксплуатационном окружении. Как показано в листинге 7.27, нам требуется лишь настроить переменную `config`.

Листинг 7.27 ❖ Включение поддержки SSL в приложении (config/environments/production.rb)

```
Rails.application.configure do
  .
  .
  .
  # Включить доступ к приложению только через SSL, использовать защищенный
  # транспортный уровень и защищенные сокеты.
  config.force_ssl = true
  .
  .
  .
end
```

Теперь необходимо настроить поддержку SSL на удаленном сервере. Включение поддержки SSL на сайте предполагает покупку и настройку *SSL-сертификата* для вашего домена. Это весьма большой объем работы, но, к счастью, здесь он нам не понадобится: приложения, работающие в домене Heroku (такие как учебное приложение), могут «упасть на хвост» компании Heroku и использовать ее SSL-сертификат. Поэтому, когда мы развернем приложение в разделе 7.5.2, поддержка SSL автоматически будет доступна. (Если вы захотите включить поддержку SSL в собственном домене, например `www.example.com`, зайдите на страницу: https://www.nic.ru/dns/service/ssl/get_certificate.html.)

---

[1] Технически правильнее называть SSL как TLS (Transport Layer Security – безопасность транспортного уровня), но все, кого я знаю, по-прежнему говорят «SSL».

7.5.2. Действующий веб-сервер

После включения поддержки SSL необходимо настроить приложение для использования действующего веб-сервера. По умолчанию Heroku пользуется сервером WEBrick, написанном на языке Ruby, который легко настраивается и запускается, но он плохо справляется с большими нагрузками (https://devcenter.heroku.com/articles/ruby-default-web-server). Поэтому WEBrick не подходит для целей эксплуатации, и мы заменим его на Puma – HTTP-сервер, способный обрабатывать большое количество входящих запросов (https://devcenter.heroku.com/articles/deploying-rails-applications-with-the-puma-web-server).

Чтобы добавить новый веб-сервер, просто последуем инструкциям в документации Heroku. Сначала добавим в `Gemfile` гем `puma` (листинг 7.28). Так как он нам не нужен на локальном компьютере, поместим его в группу `:production`.

Листинг 7.28 ❖ Добавление Puma в Gemfile

```
source 'https://rubygems.org'
.
.
.
group :production do
  gem 'pg',             '0.17.1'
  gem 'rails_12factor', '0.0.2'
  gem 'puma',           '2.11.1'
end
```

Мы настроили компоновщик так, что он не будет устанавливать гемы для эксплуатационного окружения (раздел 3.1), поэтому код в листинге 7.28 не добавит никаких гемов в окружение разработки, но нам нужно запустить его, чтобы обновить `Gemfile.lock`:

```
$ bundle install
```

Далее создадим файл `config/puma.rb` и заполним его содержимым из листинга 7.29. Этот код взят прямо из документации Heroku[1], и совершенно нет нужды его понимать.

Листинг 7.29 ❖ Файл настройки веб-сервера для эксплуатационного окружения (config/puma.rb)

```
workers Integer(ENV['WEB_CONCURRENCY'] || 2)
threads_count = Integer(ENV['MAX_THREADS'] || 5)
threads threads_count, threads_count

preload_app!

rackup      DefaultRackup
port        ENV['PORT'] || 3000
environment ENV['RACK_ENV'] || 'development'
```

[1] В листинге 7.29 немного изменено форматирование, чтобы уместить код по ширине страницы.

```
on_worker_boot do
  # Настройки, специфические для Rails 4.1+ см.: https://devcenter.heroku.com/articles/
  # deploying-rails-applications-with-the-puma-web-server#on-worker-boot
  ActiveRecord::Base.establish_connection
end
```

Наконец, создадим так называемый файл процессов `Procfile`, чтобы Heroku запускала процесс Puma в эксплуатационном окружении (листинг 7.30). Файл процессов `Procfile` должен быть создан в корневом каталоге приложения (то есть там же, где и `Gemfile`).

Листинг 7.30 ❖ Procfile для запуска веб-сервера Puma (./Procfile)

```
web: bundle exec puma -C config/puma.rb
```

После настройки веб-сервера можно отправлять изменения в репозиторий и приступать к развертыванию приложения[1]:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Use SSL and the Puma webserver in production"
$ git push
$ git push heroku
$ heroku run rake db:migrate
```

Теперь мы имеем форму регистрации, действующую в эксплуатационном окружении, и результат успешной регистрации можно увидеть на рис. 7.24. Обратите внимание на тип протокола `https://` и значок замка в адресной строке, все это указывает на действующую поддержку SSL.

7.5.3. Номер версии Ruby

В процессе развертывания приложения в среде Heroku вы могли увидеть предупреждение:

```
###### WARNING:
       You have not declared a Ruby version in your Gemfile.
       To set your Ruby version add this line to your Gemfile:
       ruby '2.1.5'
(
###### ВНИМАНИЕ:
       Вы не объявили версию Ruby в своем Gemfile.
       Чтобы объявить номер версии Ruby,
       добавьте следующую строку в Gemfile:
       ruby '2.1.5'
)
```

---

[1] Мы не изменяли модели данных в этой главе, поэтому нет необходимости запускать миграцию в Heroku, но только если вы четко следовали указаниям в разделе 6.4. Так как некоторые читатели жаловались на появление проблем, я на всякий случай добавил `heroku run rake db:migrate` в качестве заключительного шага.