Web Engineering:
Task: Validation

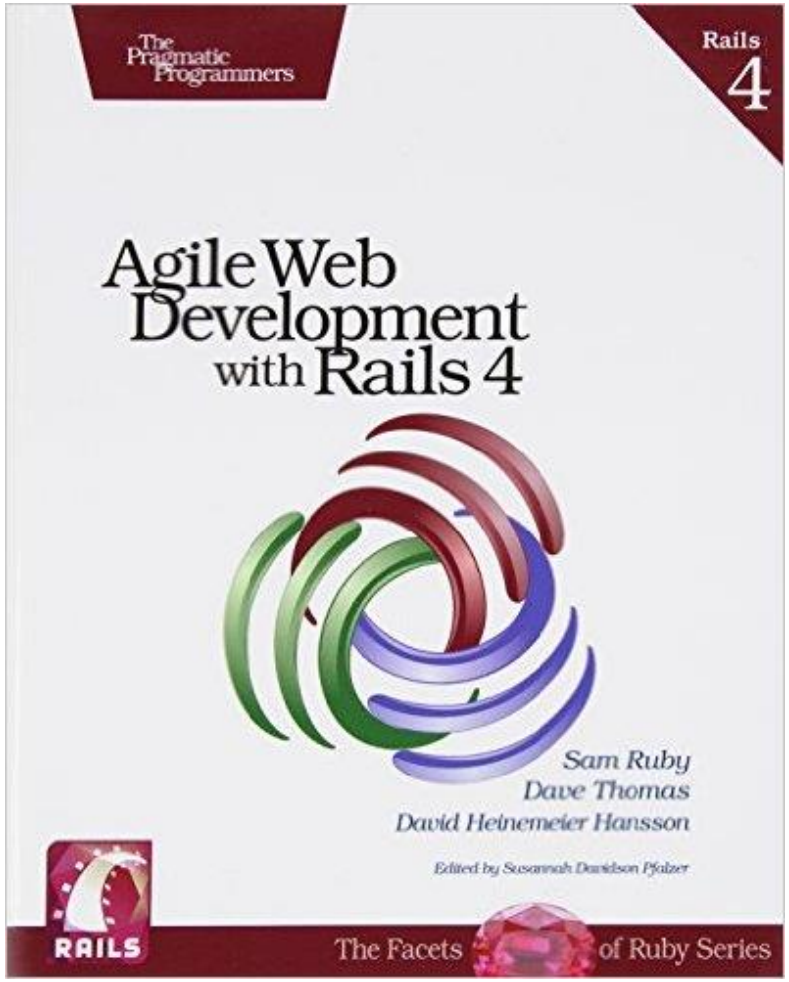
The University of Aizu
Quarter 2, AY 2018

Outline

- Validation
- Unit testing of models

Literature

- Agile Web Development with Rails 4 (1st edition) by Sam Ruby, Dave Thomas and Devid Hansson, The Pragmatic Bookshelf, 2013.
  - Chapters 7.



Validation

- If the seller enters an invalid price or puts nothing into the product description field, the application accepts the form and adds the record to the database. :(
- There is no checking data in the application.
- Where to put validation?
  - The model layer is the gatekeeper between the code and the database. The model should check the data before writing to the database.
  - The database will be protected from bad data.

Product class: Initial version

- Source code of the model class
  - app/models/product.rb

```
class Product < ActiveRecord::Base
end
```

- Adding the validation is easy!

Product class: Version with validation

- Updated *Product* model looks as follows:

Download rails40/depot_b/app/models/product.rb

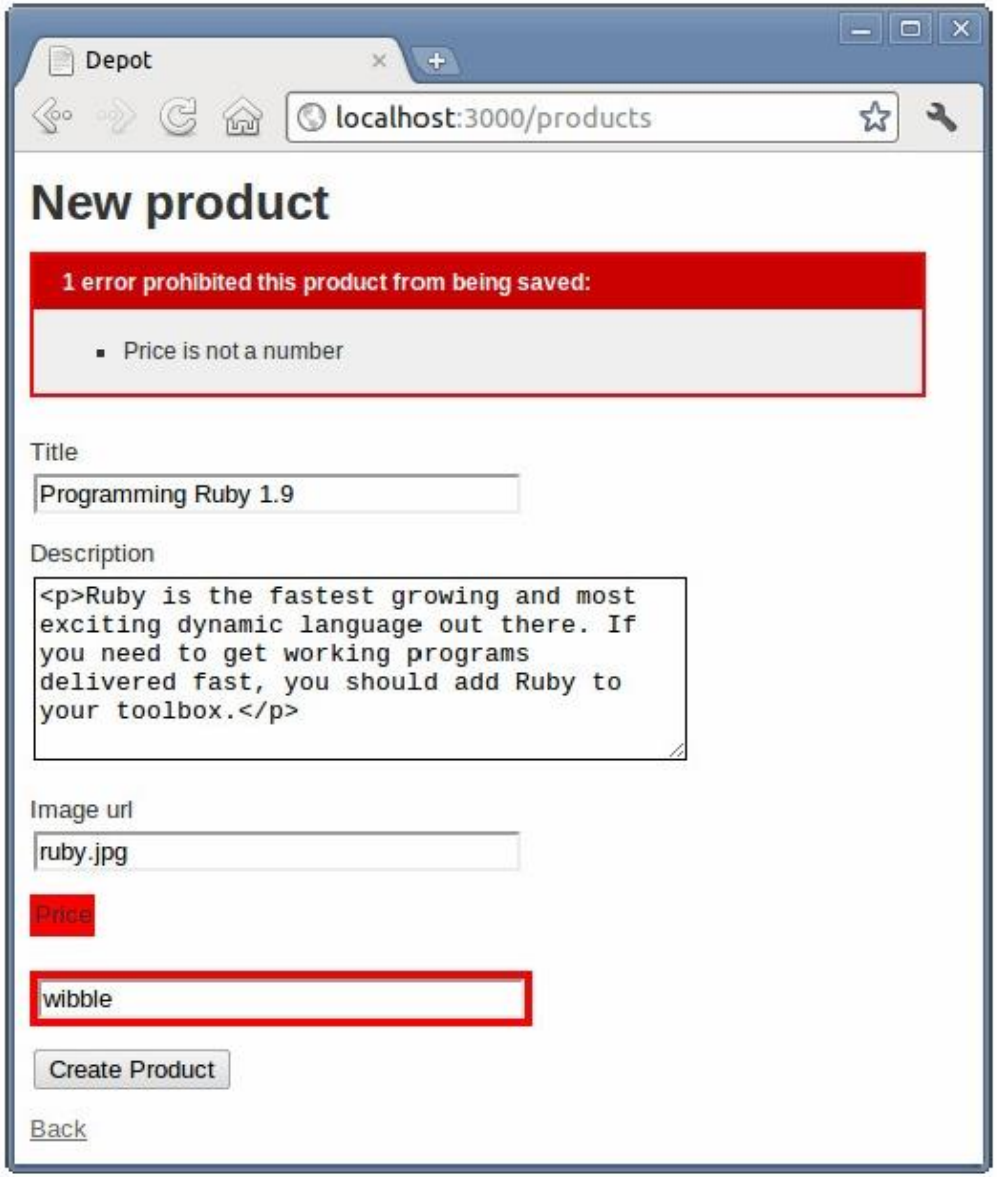
```
class Product < ActiveRecord::Base
  validates :title, :description, :image_url, presence: true
  validates :price, numericality: {greater_than_or_equal_to: 0.01}
  validates :title, uniqueness: true
  validates :image_url, allow_blank: true, format: {
    with:    %r{\.(gif|jpg|png)\Z}i,
    message: 'must be a URL for GIF, JPG or PNG image.'
  }
end
```

- Line 2 says that the fields title etc. have to have values;
- Line 3 says that the value of price field is valid number and it is >= 0.01

Product class: Version with validation

- See previous slide:
  - Line 4 says that *title* field has to be unique among all products.
  - Line 5 says that the *image_url* field has a special format. The regular expression in Line 6 describes it.
  - Line 7 is a comment.

Incorrect price entered



This message appears if we try to add new product with invalid price.

Running tests

- Before going ahead, we run:
  - rake tests
- We will see two failures in:
  - test_should_create_product and
  - test_should_update_product
- To solve the problem, we need to provide valid test data in
  - test/controllers/products_controller_test.rb

products_controller_test.rb (1)

Download rails40/depot_b/test/controllers/products_controller_test.rb

```
require 'test_helper'
class ProductsControllerTest < ActionController::TestCase
  setup do
    @product = products(:one)
➤   @update = {
➤     title:       'Lorem Ipsum',
➤     description: 'Wibbles are fun!',
➤     image_url:   'lorem.jpg',
➤     price:       19.95
➤   }
  end

  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:products)
  end

  test "should get new" do
    get :new
    assert_response :success
  end
```

products_controller_test.rb (2)

```
  test "should create product" do
    assert_difference('Product.count') do
➤     post :create, product: @update
    end

    assert_redirected_to product_path(assigns(:product))
  end

  # ...
  test "should update product" do
➤   patch :update, id: @product, product: @update
    assert_redirected_to product_path(assigns(:product))
  end

  # ...
end
```

Comments on Slides 10 and 11

- Mark ▶ points new lines in the file.
- Slide 10 illustrates how to give and describe the test data.
- Slide 11 shows where to incorporate code lines to use the test data.

Unit testing of models

- From the moment, we have created a new application using *rails* command, Rails starts generating a test infrastructure for us!
- Look inside the *unit* subdirectory:

```
depot> ls test/models
product_test.rb
```

  - *product_test.rb* file is created by Rails to hold the unit tests for our model:

Download rails40/depot_a/test/models/product_test.rb

```
require 'test_helper'

class ProductTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

Unit testing of models

- *ProductTest* is a subclass of *ActiveSupport::TestCase*. The last one is a subclass of the Mini*Test::Unit::TestCase* class.
- Rails generates tests based on Mini*Test* framework.
- Inside this test case (see the previous slide), Rails generated a single commented-out test called "*the truth*".
- The assert line is an actual test.

Real unit test

- If we create a product with no attributes set, we will expect it to be invalid and errors should be associated with each field. This is *what* to test.
- We need to know *how* to tell the test framework whether our code passes or fails.
  - An *assertion* is a method call that tells the framework what we expect to be true.
  - The simplest variant is *assert*: expects that its arguments are true.

Real unit test

- In our case, we expect that an empty *Product* model will not pass validation
- We express this expectation by asserting that it is not valid.
- We replace the *truth* test with the following code in

Download rails40/depot_b/test/models/product_test.rb

```
test "product attributes must not be empty" do
  product = Product.new
  assert product.invalid?
  assert product.errors[:title].any?
  assert product.errors[:description].any?
  assert product.errors[:price].any?
  assert product.errors[:image_url].any?
end
```

Real unit test

- Now, we may run the tests:

```
depot> rake test:models
.
Finished tests in 0.257961s, 3.8766 tests/s, 19.3828 assertions/s.
1 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

- All assertions are passed!
- Now, we can analyze individual validations.
- We will study three of the many possible tests.

Validation of the price

Download rails40/depot_c/test/models/product_test.rb

```
test "product price must be positive" do
  product = Product.new(title:       "My Book Title",
                        description: "yyy",
                        image_url:   "zzz.jpg")
  product.price = -1
  assert product.invalid?
  assert_equal ["must be greater than or equal to 0.01"],
    product.errors[:price]

  product.price = 0
  assert product.invalid?
  assert_equal ["must be greater than or equal to 0.01"],
    product.errors[:price]

  product.price = 1
  assert product.valid?
end
```

Comments on the previous slide

- In this code, we create a new prodcut and then try setting its priceto -1, 0, and +1.
- If our model is working the first and the second variants are invalid and we should receive the error message.
- These three tests may be implemented by three separate test methods.
  - Such solution is acceptable as well.

Image URL validation

Download rails40/depot_c/test/models/product_test.rb

```
def new_product(image_url)
  Product.new(title:       "My Book Title",
              description: "yyy",
              price:       1,
              image_url:   image_url)
end
test "image url" do
  ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
           http://a.b.c/x/y/z/fred.gif }
  bad = %w{ fred.doc fred.gif/more fred.gif.more }
  ok.each do |name|
    assert new_product(name).valid?, "#{name} should be valid"

  end
  bad.each do |name|
    assert new_product(name).invalid?, "#{name} shouldn't be valid"
  end
end
```

Comments on the previous slide

- We are testing that the image URL ends with one of .gif, .jpg, or .png
- We have used TWO loops:
  - Loop 1 checks the cases we expect to pass validation
  - Loop 2 pays attention to the cases we expect to fail.
- We have added an extra parameter to our assert method. It contains a string.
  - It will be written along with the error message if the assertion fails.
- Our model contains a validation that checks that all the product title in database are unique.
  - Rails *Fixtures* are to implement this validation.

Test Fixtures

- A *Fixture* is an environment in which you can run a test.
- In Rails, a test fixture is a specification of the initial contents of a model (or models) under test.
  - If we would like to ensure that our *Products* table starts off with known data at the start of every unit test, we can specify those contents in a fixture, and Rails will take care of the rest.
- You specify fixture data in files in *test/fixtures* directory.

Automatically created fixture file

Download rails40/depot_b/test/fixtures/products.yml

```
# Read about fixtures at
# http://api.rubyonrails.org/classes/ActiveRecord/Fixtures.html
one:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
two:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```

The file name must be same as the model name: *products.yml* is to show that the lines in the test data are in YAML format.

Comments on the previous slide

- The fixture file contains an entry for each row that we want insert into the database.
  - Their names are *one* and *two*
- The name gives us convenient way to reference test data inside our test code.
- <u>Important Note</u>
  - All lines for a row must have the same indentation
  - Use SPACE not TAB for indentation
  - When making changes, make sure that name of columns are correct in each entry
  - A mismatch with the database colimn names may cause hard-to-track-down exception.

Modified fixture file

Download rails40/depot_c/test/fixtures/products.yml

```
ruby:
  title:       Programming Ruby 1.9
  description:
    Ruby is the fastest growing and most exciting dynamic
    language out there.  If you need to get working programs
    delivered fast, you should add Ruby to your toolbox.
  price:       49.50
  image_url:   ruby.png
```

- Now, we want Rails to load test data into the *products* table when we run the unit test.
  - We may control which fixtures to load by specifying the following line in *test/models/product_test.rb*

```
class ProductTest < ActiveSupport::TestCase
  fixtures :products
  #...
end
```

Modified fixture file

- The *fixture* directive loads data corresponding to the given model name into corresponding database table before each test method in the test case is run.
- The name of the fixture file determines the table that is loaded so using *:products* will cause the *products.yml* fixture file to be used.

Modified fixture file

- Rails needs to use a test database.
- If you look in the *database.yml* file in the *config* directory, you will notice that Rails created a configuration for three separate databases:
  - *db/development.sqlite3* is our development database.
  - *db/test.sqlite3* is a test database
  - *db/production.sqlite3* is the production database.
    - Our application will use this one when we put in on-line.
- Each test method gets a freshly initialized table in the test database, loaded from the fixtures we provide.
- This is automatically done by command:
  - *rake test*

Using fixture data

- We know how to get fixture data into database.
- We need to find way of using it in our tests.
- In the case of our product data, calling products(:ruby) returns a Product model containing the data we defined in the fixture
- We will use that to test the validation of unique product titles.

Using fixture data

Download rails40/depot_c/test/models/product_test.rb

```
test "product is not valid without a unique title" do
  product = Product.new(title:       products(:ruby).title,
                        description: "yyy",
                        price:       1,
                        image_url:   "fred.gif")

  assert product.invalid?
  assert_equal ["has already been taken"], product.errors[:title]
end
```

- The test assumes that the database includes a row for the Ruby book.
- It gets the title of that existing row using this:

  products(:ruby).title
- See the next slide.

Using fixture data

- Our test from the previous slide creates a new Product model, setting its title to that existing title.
- It asserts that attempting to save this model fails and that the title attribute has correct error associated with it.

Using fixture data

- If you want to avoid using hard-coded string for the Active Record error, you can compare the response against its buil-in error message table:

Download rails40/depot_c/test/models/product_test.rb

```
test "product is not valid without a unique title - i18n" do
  product = Product.new(title:       products(:ruby).title,
                        description: "yyy",
                        price:       1,
                        image_url:   "fred.gif")

  assert product.invalid?
  assert_equal [I18n.translate('errors.messages.taken')],
               product.errors[:title]
end
```

Using fixture data

- We can feel confident:
  - Our validation code not only works but will continue to work.
  - Our product now has a model, a set of views, a controller and a set of unit tests.

Playtime

- If you are using Git, it is a good time to commit our work.
- First, you should see the files we changed:

```
depot> git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   app/models/product.rb
# modified:   test/fixtures/products.yml
# modified:   test/controllers/products_controller_test.rb
# modified:   test/models/product_test.rb
# no changes added to commit (use "git add" and/or "git commit -a")
```

  - Since we did not add any new files (only change existing), we may commit changes:
    - depot>git commit –a –m 'Validation!'

What we just did

- A dozen of code lines gave us the following:
  - We ensured that required fields were presented
  - We ensured that price fields were numeric and at least one cent
  - We ensured that the titles were unique
  - We ensured that images matched a given format
  - We updated the unit tests that Rails provided, both to conform to the model constrains and to verify the new code that we added.