

# **Lesson 15:**

## Debugging & testing

# Objectives

---

- Logging
- Using the Ruby debugger
- Understand unit, functional, and integration testing
- Write and use a unit test

## What is debugging?

---

Debugging is the process of using tools to find and remove code errors.

Various tools may be used during debugging:

- Logging
- Exceptions
- Variable output commands and expressions
- Parameter filters
- Interactive debugger

More discussion of Rails debugging can be found here

[http://guides.rubyonrails.org/debugging\\_rails\\_applications.html](http://guides.rubyonrails.org/debugging_rails_applications.html)

## What is a log file?

---

Log files provide insight into runtime application performance.

Rails normally writes to up to three log files to /log folder:

- development.log
- test.log
- production.log

## How do I write specific values to a log file?

---

You can write custom messages in both code and template files.

Write to log a model, controller, or other code file

```
logger.info 'This will appear in the log'
```

Write to log in a view or other ERB template

```
<% logger.info 'Appears in the log' %>
```

Both display as output and write to log in a view or other ERB template

```
<%= logger.info 'Appears in the log and the page' %>
```

## How do I prevent passwords, etc from being written to log files?

---

Logging of sensitive information being sent from the browser can be suppressed.

Add filter parameter method to app/controllers/application.rb to block logging specified parameter(s)

```
filter_parameter_logging :password, :credit_card_number
```

**Exercise:**  
Writing values  
to the log file

## What is the Rails console and how do I use it?

---

The Rails console is an IRB instance which runs in context of an application and its objects.

Launch Rails console with persistent changes to the application

```
rails console
```

Launch Rails console with changes rolled back when the console session ends

```
rails console --sandbox
```



## How do you display variable values from a controller or view?

---

Specific variable values can be displayed as output.

Display variable from a controller by raising an exception and converting output to YAML

This approach raises an error message which displays the specified variable values

```
raise @variable_name.to_yaml
```

Display variable from a view template

```
<%= debug @variable_name %>
```

**Exercise:**

Displaying variables  
as debug output

## **What is the Ruby debugger and how do I install it?**

---

A debugger enables line by line execution of code beginning at a pre-defined breakpoint.

The Ruby debugger is installed as a Gem, and should be used only on development systems.

```
gem install ruby-debug19
```

Once installed, uncomment or add this to the application Gemfile

```
gem 'ruby-debug19', require: 'ruby-debug'
```

## How do I configure code for the debugger?

---

Add the debugger command in source code where line by line execution should begin:

```
def reviews
  @movie = Movie.find(params[:id])
  @reviews = @movie.reviews

  debugger

  respond_to do |format|
    format.html { render 'reviews/index' }
    format.json { render json: @movies }
  end
end
```

## How do I use the debugger?

---

Launch the Rails server in debugging mode

```
rails server -debugger
```

Interact with the application, and use these commands from the (rdb:<line\_number>) command line:

**list**: display where code execution has halted

**next** (or **step**): execute the next line of code

**cont**: leave the debugger and continue normal code execution

**quit**: leave the debugger and shut down Rails server

## **Exercise:**

Install, configure & use  
the Ruby debugger

## **What is a test database fixture?**

---

A test database fixture is a pre-configured data set for the database to ensure a consistent test environment.

Fixtures are automatically created in test/fixtures when models are generated and are written in YAML and may be customized to provide more realistic data.

They need to be customized to support associations and need to be manually verified against validation rules before using.

## What is a test database fixture?

---

Test databases are rebuilt to a default state using fixtures between tests to provide consistency.

Once manually customized, test fixtures by running:

```
rake test
```



## What environments are available in Rails?

---

Rails supports three runtime modes:

***Development*** = default mode, no caching so code changes are run immediately

***Test*** = caching plus fixtures to provide consistent test environment

***Production*** = caching and abbreviated logging to maximize speed

```
# or -e production or -e development  
rails server -e test
```

# What are some approaches to software testing?

---

## *Unit testing*

Automated approach to checking fine-grained behavior (e.g., validation works correctly with sample data?)

- unit testing focuses on model validations and associations
- generating a model generates “stub test code” in folder test/unit

# What are some approaches to software testing?

---

## *Functional testing*

Automated approach to checking action (method) behavior

- In Rails, functional testing focus on controller
- Generating a controller generates code in folder test/functional

# What are some approaches to software testing?

---

## *Integration testing*

Automated approach to checking all interactions in a full request cycle

- integration testing tests full stack: models, controllers, routes
- Rails generates stub test code in folder test/integration

## **What tools and approaches are commonly used in software testing?**

---

Software testing is a complex topic largely beyond the scope of an introductory course:

<http://guides.rubyonrails.org/testing.html>

# What tools and approaches are commonly used in software testing?

---

## Test Driven Development (TDD)

*Tests designed and built before code is written*

Commonly used Rails testing tools include:

RSpec	<a href="http://rspec.info">http://rspec.info</a>
Minitest	<a href="https://github.com/seattlerb/minitest"><u>https://github.com/seattlerb/minitest</u></a>
Cucumber	<a href="http://cukes.info">http://cukes.info</a>
Capybara	<a href="https://github.com/jnicklas/capybara">https://github.com/jnicklas/capybara</a>
Factory Girl	<a href="https://github.com/thoughtbot/factory_girl">https://github.com/thoughtbot/factory_girl</a>

## How do you build a simple unit test?

---

Rails generates a test class derived from ActiveSupport::TestCase which requires test\_helper

Unit testing relies on the assert statement

An assert statement

- expects its argument to be true, and fails the test if it is false
- may be followed by a string describing what has failed if the assertion is false

Whether a test is checking for success or failure determine if the assertion is true or false

## How do you build a simple unit test?

---

```
require 'test_helper'
```

```
class ReviewTest < ActiveSupport::TestCase
```

```
  def test_score_too_high
```

```
    # test if a too-high score is not valid
```

```
    review = Review.new
```

```
    review.score = 15
```

```
    assert !review.valid?, "score too high"
```

```
  end
```

```
end
```



## What assertions are supported by Rails?

---

Rails supports assertion types beyond `assert(expression, message)` including

```
assert_equal(object1, object2, message)
assert_not_equal(object1, object2, message)
assert_kind_of(class, object, message)
assert_respond_to(object, :method, message)
```

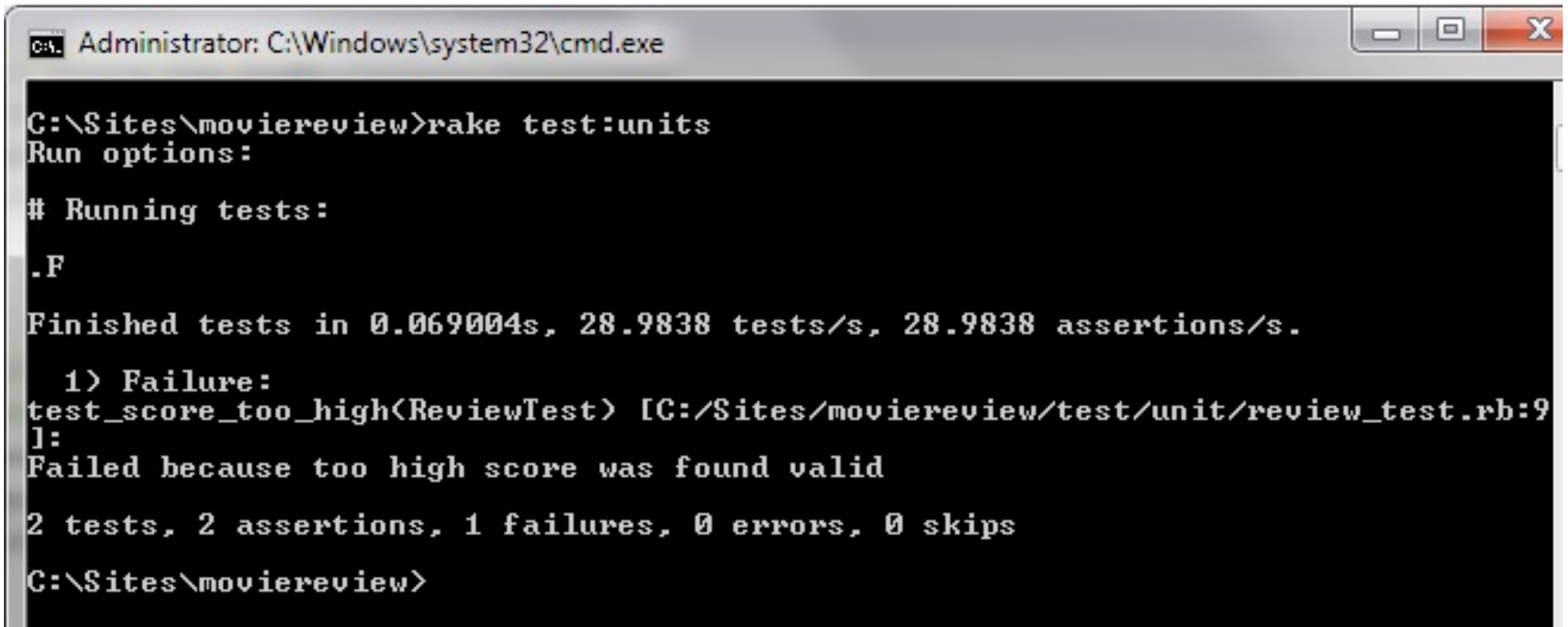
Plus many, many more:

<http://guides.rubyonrails.org/testing.html#assertions-available>

## How do you run the tests you've created?

---

Defined unit tests are executed using the rake test:units command



```
Administrator: C:\Windows\system32\cmd.exe

C:\Sites\moviereview>rake test:units
Run options:
# Running tests:
.F
Finished tests in 0.069004s, 28.9838 tests/s, 28.9838 assertions/s.

  1) Failure:
test_score_too_high<ReviewTest> [C:/Sites/moviereview/test/unit/review_test.rb:9
]:
Failed because too high score was found valid

2 tests, 2 assertions, 1 failures, 0 errors, 0 skips
C:\Sites\moviereview>
```

**Exercise:**

Building & running  
a simple unit test

# Heroku

---

How to get your site online using a service called Heroku:

- Sign up at <http://www.heroku.com/>
- Install the toolbelt at <https://toolbelt.heroku.com/>
- Do `heroku login` on the command line
- Change in your Gemfile `gem 'sqlite'` to `gem 'pg'`
- Update your bundle by doing `bundle install`
- Go to your app and type in `heroku create <your-appname>`
- Make sure you've set up a git repository in that folder and added and committed your files (`git init`, `git add`, etc)
- Then do `git push heroku master`
- Once finished do `heroku open`

## Heroku issues

---

Heroku doesn't let you write to its system, so you can't do things like file uploads. To get around this you can upload to a cloud management tool like Amazon S3.

<https://devcenter.heroku.com/articles/paperclip-s3>

## Heroku logs and console

---

View the logs on Heroku to see where things are going wrong by using:

`heroku logs`

Debug using the console can be done using

`heroku console`

## Heroku Add-ons

---

One thing that's always worth checking out is the extra add-ons services that Heroku and 3rd parties provide:

<https://addons.heroku.com/>

## Heroku Gems

---

One gem I use quite a lot is Heroku\_san

[https://github.com/fastestforward/heroku\\_san](https://github.com/fastestforward/heroku_san)

Helps configure things across different Heroku sites. For example, I have a development version of a site, a staging version and a production (or live) site.