Test driven Web development with Ruby on Rails

**2008**

Michael Grosser

TFH-Berlin

20.06.2008



# Erklärung der Basistechnologien

## Ruby

### Jung

Seit 1993 kann man in Ruby programmieren, doch lange fristete es ein ehr unauffälliges Dasein. Erst 2003 wurde Ruby durch das Web-Framework „Ruby On Rails“ (nachfolgend als „Rails“) ins Licht der Öffentlichkeit gerückt. (Wikipedia: Ruby, 2008)

Dank des großen und anhaltenden Erfolgs von Rails, steigt der Bedarf an Ruby Entwicklern ständig (200% Steigerung pro Jahr, in 2005-2008) (Indeed: Ruby, 2008)).

### Objektorientier

Viele Sprachen behaupten Objektorientiert zu sein, doch meist gibt es primitive Konstrukte die keine Objekte sind (z.B. int/double in Java). In Ruby ist alles ein Objekt, jede Nummer, jeder String, selbst die Klassen sind Objekte.

### Dynamisch

Im Gegensatz zu statischen Sprachen wie Java, ist es bei Ruby nicht nötig Variablen oder Rückgabewerten eine Klasse zu geben (a = 1). Methoden können zur Laufzeit definiert werden (define\_method :add do …). Man kann ohne Klassen ein funktionierendes Programm schreiben(print “Hello World“). Code kann evaluiert werden (eval(“print 1+3“)).

### Offen

Jedes Objekt und jede Klasse kann jederzeit verändert werden. Man kann ein neue Methode auf Basis-Klassen wie String definieren, oder existierende Methoden überschreiben. (def String.hello; self+“hello“;end) . Frameworks wie Rails nutzen dies, um Domain spezifische Probleme einfache zu lösen. (String.blank?, String.titlecase …)

### Lesbar

Ruby und Ruby Entwickler legen viel Wert auf Lesbarkeit. Code wird dem Namen „Code“ nicht mehr gerecht, denn er ist selbst für nicht-Programmierer verständlich.

Java:

for(int i=0;i<6;i++){System.out.println(“Hello“);}

Ruby:

print 5 \* ”Hallo”

## Ruby on Rails

### Web MVC Framework

Rails ist ein Model-View-Controller Framework um Webseiten zu erstellen. Es ist ausgerichtet auf HTML + CSS + Javascript(nachfolgend „JS“), kann aber auch z.B. als Backend für Flash Seiten benutzt werden.

Model:   
Datenbank Abstraktion (transparent für SQLite, MYSQL, Oracle…) die jede Spalte einer Tabelle als Objekt repräsentiert.

Controller:   
Aktionen (actions) werden auf Anfrage des Nutzers (Aufruf der entsprechenden URL im Browser) ausgeführt. Diese sind entweder anzeigende Aktionen (Daten eines Autos darstellen) oder ändernde (Neuer Name des Autos 1 = ‚Mein Auto‘)

View:  
Es wird das vom Controller geladene Model in der gewünschten Anzeigesprache repräsentiert(HTML,XML,JSON,…) oder Formulare zur Eingabe von Daten angezeigt.

### Auf Testbarkeit optimiert

Für alle Bereiche von Rails gibt es die passenden Test-Strukturen und Test-Helfer. Es wird großer Wert darauf gelegt, das auch sonst schwer testbare Aktionen (z.B. verschicken einer Mail order Aufruf einer URL) einfach zu testen sind.

### Convention over configuration

So lange man sich an die Konventionen von Rails hält (z.B. Model heißt wie die Tabelle, Controller liegen in apps/controller .. ) kann man sehr schnell sehr viel Funktionalität bauen, ohne sich langwierig in Dokumentation einzulesen. Dies wird erleichtert, durch die große Zahl an Generatoren, die z.B. einen kompletten Controller samt Views und Modell erstellen.

### Plugin Architektur

Plugin sind einfach zu erstellen und brauchen (meist) keinerlei Konfiguration. Man erstellt einen neuen Ordner in vendor/plugins mit einer init.rb. Diese wird beim Start von Rails aufgerufen. Durch die Offenheit von Ruby lässt sich so in einem Plugin das gesamt Verhalten der Anwendung ändern, ohne das die Erbauer es vorher einplanen mussten.

# Introduction

Testing web applications is complicated, fragile and time consuming. Rails offers a basic set of tools to make testing more effective and usable. But these are not sufficient to thoroughly test a whole application. With additional methods, tools and plugins a complete test from model through controller and views up to Javascript is possible.

# The Lack of Testing in Web development

## Why testing a web application is hard

Web development, especially HTML/Ajax based, has a long tradition of untested code. Unlike a normal programming environment the programmer here has to handle 2 separate applications.

* Model/HTML generation (calculations and rendering) through MVC
* Browser behavior (page change/forms/redirects…)

One could only test that calculations inside the model were correct (normal Unit testing) and that the rendered HTML was valid. Testing that the correct action is called upon entering a URL is mostly unpractical since Apache rewrite rules were used for this purpose. For the same reason it is hard fake a request-response cycle when you first have to set up all kind of global variables (php: $\_GET, $\_REQUEST…) or build a session object. Some modern web frameworks provide ways of testing (Symfony/CakePHP..) , but for most small framework this stays a half-hearted approach. Therefore Controller or view testing is often achieved through a remote controlled browser.

Rails changed this story, by allowing the programmer to test between controller-actions and view-rendering. Now he can make requests, see the controller change and verify that a view works with edge-case inputs. With the help of plugins it is even possible to submit forms or click links. All these basic actions allow to thoroughly test code without opening the browser.

If we still were in web 1.0 lands with 98% users on IE5, no CSS or JS, the story would end here and the web developers would live happily ever after…

## Browser differences

Even if we could test JS from inside Rails, we still could not be sure that it works in all browsers. Every browser interprets JS. Something as simple as {‘a’:1,} will work somewhere and will crash somewhere else. To write reliable tests we have to test all major browsers, which again is not possible within Rails, since it just knows plain HTML-String output and nothing about parsing JS or what proper JS syntax should look like.

Another part that is very hard to test is CSS. It changes the graphical output and has no testable Text-representation. Therefore verifying that your page looks the same on all browsers can hardly be automated.

The most common method is watching your site in all browsers, which is a very dull task. It is possible to test some basic parts of CSS. Positions of elements can be checked with Javascript, by testing if the offsetWidth/offsetHeight/… of an element is correct. Getting the expected values is relatively easy with the Webdeveloper Toolbar[[1]](#footnote-2). Since testing everything is not possible (overflow:hidden works or not?), this is only a way to assure some basic alignment or pixel exact positioning.

To visually test your layout in many browsers many professionals use BrowserShots.org (or a similar service), where your page is rendered in up to different browsers. Most are free as long as you are willing to wait for ca. 2 hours until your screenshots arrive. For 10€/month you can get priority processing, which is worth the time saved to setup all browsers and systems needed for such a complete test.

## Javascript

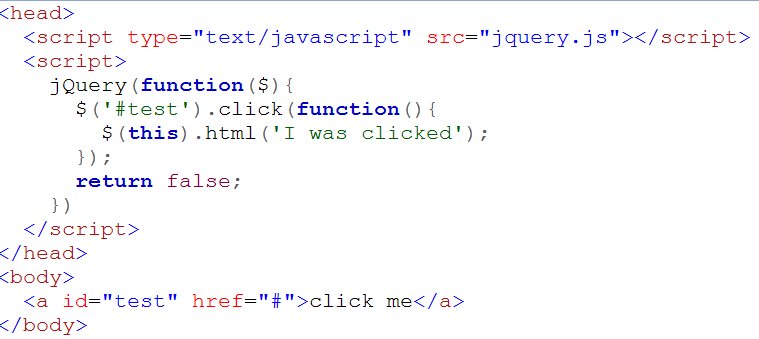
### Unobstrusive JS

There are 2 ways of building a application with JS, obtrusive or unobtrusive. Here is a small example to show the difference.

<div onclick=”hello()”>

This is obtrusive, since it changes the structure of the HTML document (the Document Object Model, further referred to as DOM) by inserting attributes (onclick,onmouseover,…) to the DOM that do not describe its structure.

Unobtrusive by example: ( examples/unobtrusive/unobstrusive.html )



2

3

1

1. load the library
2. “when the DOM is loaded, start this function” and “pass the jQuery object as parameter $ to it”
3. when test is clicked, change its ‘innerHtml’ to ‘I was clicked’

Using this setup (step 1 + 2) can be overhead for simple cases like this. But for larger projects, not using it leads to ever-increasing complexity.

In general there should be 3 Layers in your Frontend: HTLM(Structure), CSS(Style), JS(Behavior). Unobtrusive in this context means: “separate JS from HTML like CSS”.

A short list of obtrusive disadvantages:

* using onclick=”” limits your code to have one event per element
* JS messes up the HTML
* JS is repeated several times 🡪 ~~DRY~~, debugging gets harder
* you cannot turn off JS
* hard to move behavior from one component to another, or reuse it

### Problems with unobtrusive JS

Everything is cleanly separated from HTML, but how to test that the JS works or even if it is there after all? It is not possible to write something like “test that HTML changes when link #test is clicked” since Rails only knows the pure HTML output that came from the controller, and has no idea of any attached JS behavior.

Simple structures written in obtrusive JS are testable by “assert there is a onclick attribute that looks like /$(‘#test’).click(/”. But this kind of testing does not help, since we cannot be sure the script will work, we just can get sure that it looks like it could work. What nevertheless some frameworks do.

script/plugin install http://thar.be/svn/projects/plugins/arts/

### Conclusion

While most other languages have one default compiler or platform, JS has many. While most other platforms can simulate almost any user interaction, even the most basic fail with Javascript. The main problem is not that there is no Javascript compiler (you could run one from the command line e.g. rhino), but that passing tests in them does not mean your code will run in the real, browser-world.

Hence, any good test has to run through a browser, which means opening a browser and going through all test pages. This burden seems too hard for many developers, and consequently most low-to-medium sized projects lack even basic JS tests.

# Testing Basics

## Getting started

### Before wrting tests, test

This topic may sound strange, but it is essential to not just start writing tests and hope you can refactor or fix any error that occurs. You should be sure that what you are building will work. Before that it is highly recommended to understand the problem domain by RTFM(Reading the manual) or looking at example code. When you have a basic idea, start with a prototype (a real throw-away-and-never-look-at-again kind of prototype). Alternatively irb or script/console (remember to always use “script/console test” or you mess up your development environment) can be of great help.

Even after starting to test, some problems may occur, that cannot be fixed with testing alone. If you have no idea what is right or why something fails, reading and playing with the console is faster than testing.

script/console test

Loading test environment (Rails 2.0.2)

**>> r = Rating.new**

=> #<Rating id: nil, r1: 0, r2: 0, r3: 0, r4: 0, r5: 0, rating: nil, movie\_id: nil, created\_at: nil, updated\_at: nil>

**>> r.save!**

ActiveRecord::StatementInvalid: Mysql::Error: #23000Column 'movie\_id' cannot be null: INSERT INTO …

**>>r.movie = Movie.new**

=> #<Movie id: nil, country: nil, year: nil, duration: nil, title: nil, director: nil,

**>> r.save!**

ActiveRecord::StatementInvalid: Mysql::Error: #23000Column 'movie\_id' cannot be null

#what is wrong now, we assigned a movie and there is no id ? let us dig deeper…

**>> r.movie.save!**

ActiveRecord::RecordInvalid: Validation failed: Title can't be blank

#problem found!

**>> r.movie.title = 'test'**

=> "test"

**>> r.movie.save!**

=> true

#success, now let us see if all other values are fine too

**>> y r** #y is a shorthand for outputting the yaml version of an object

--- &id003 !ruby/object:Rating

attributes:

rating: 0

updated\_at: 2008-03-20 15:21:47.116913 Z

movie\_id: 633756704

id: 953125646

r1: 0

…

#looks fine, now it is time to build the tests

**>> exit**

SAMPLE CONSOLE SESSION

Sample console session, before writing tests.

### Integration and Unit testing

There are 2 types of testing mentalities, integration testing (testing deep) and unit testing (testing small). Integration testing means: writing a test case that touches a lot of code and verify that all works in unison. This can work good to verify that something is working as expected, since any failure would be visible in the final result. This kind of testing is typical for test last development, since testing here only serves as verification that the code runs as expected. But this approach has many drawbacks.

* No safety that everything is tested
* A lot of different failures when changing one aspect
* Very unclear failures, that require more “digging”
* Not usable for test-driven development, since everything has to be build at once

In consequence we will rely on unit testing, which means that every test should cover only one aspect of the code. Some bad examples:

* Testing 2 paths of execution (success and failure)
* Testing something, without verifying an underlying assumption first

**def** test\_should\_calculate\_correctly

assert\_equal "2.83", **@rating**.rate!(**2**).rating

**end**

* Assumes that rate! returns itself, test this first!

**def** test\_should\_return\_self\_when\_rating

assert\_equal **@rating**, **@rating**.rate!(**2**)

**end**

This test is a bit extreme, there certainly is a border where testing something to trivial will only slow down your work, but in general it is better to err on the side of making to small test, since they are easier to understand and maintain.

Another example refactoring:

**def** test\_save

get **:edit**, **:id** => Rating.first.id

submit\_form **:rating** => {**:r1** => **3**}

assert\_response :redirect

assert\_equal **3**, Rating.first.r1

assert\_match(/success/, flash[**:notice**])

assert\_redirected\_to **:controller** => 'rating', **:action** => 'index'

**end**

What would you change here, is everything as small as possible/necessary?

My suggestions:

* Split tests
* Move request to helper
* Remove duplicate redirect test
* Move Rating.first to setup (@rating = Rating.first) or use fixtures
* Rename test to express what they are testing

**def** save(params = {})

get **:edit**, **:id** => **@rating**.id

submit\_form **:rating** => params

**end**

**def** test\_should\_update\_rating\_on\_successful\_save

save **:r1**=>**3**

**@rating**.reload.r1.should == **3** #reload from db

**end**

**def** test\_should\_redirect\_to\_index\_on\_successful\_save

save

assert\_redirected\_to **:controller** => 'rating', **:action** => 'index'

assert !flash[**:notice**].blank?

**end**

* What does come to your mind when comparing before and after?
* Is it lighter, easier or more verbose?
* Would you have refactored differently?

In my opinion, it is clearer and easier to read and it now seems obvious, that a failing save was never tested, but before it was easy to overlook.

### Make just enough to pass

Before you write code, write a test case that and watch it fail, to ensure the functionality is not there and that the test-case is really proving that a feature is missing.

When writing the code to satisfy your tests, just write enough to pass, not more. Writing more means writing something that is not tested, but looks as if it is being tested. Finish your first idea, see it pass and then extend it with a new test case. Finally refactor, for code maintainability or performance.

### naming conventions

In general tests should be named for what they verify. If you have problem finding a good name, it often means you are testing more then you should, break it down in 2 and see if you can find names for them. A good scheme for naming is the expression for your assumption. “update should redirect me to index when successful”, and since we are typing lazy, this will be shortened to “update should redirect to index on success”, which results in test\_update\_should\_redirect\_to\_index\_on\_success.

These names seem rather long, but it is essential to state what you are testing. If not, test will end up being named test\_save, which suggests “save is covered, no more testing required” even it is just verifies half of the behavior. It also helps when seeing the failure output; you can instantly see where and why (“because it did not redirect to index”) something is broken. Last but not least other developers can see what you intended to test without reading the code.

### Focus on the risky parts

Sometimes testing homogenous can be a waste of time, especially when some parts are only seen by administrators or are ever-changing (e.g. HTML Structure) without effect on the behaviour. Try for a maximum amount of security per written test case. Imagine a list of risk factors where

risk = (amount of money lost)\*(chance this will happen)

and start working down this list.

## develop without opening your browser

Try to verify something with tests and the console first and save the browser as last method. Clicking around and filling forms will slow you down and give you nothing that can be automated. While writing tests helps you to become better at it and leaves an automated suite for all other members to use. Testing first and then seeing it break in the browser, because there was no redirect/input/link/… , raises your awareness for what needs testing and therefore results in better test cases and higher coverage.

## Decoupling

### Loose coupled tests

In general, the idea is to have a tight coupling between behavior and test, but on the other hand we want to keep our test decoupled from spelling and often changing environments as possible, so that our test can keep valid as long as the behavior does not change.

### Text output

Try to not repeat you views or flash messages, since every typo means you have to go back into your tests.

assert\_equals 'Your transaction was sucessful!', message, 'bad'

assert\_match / success/, message, 'good'

### Flash

Separate your flash into success(:notice) and failure(:error). On the one hand, your user will faster recognize if something succeeded or not when used with different styles, on the other hand your test can use assert flash[:notice] to test if a success notification was sent.

### Duck typing

„*If it walks like a duck and quacks like a duck, I would call it a duck.*“ - James Whitcomb Riley

Rely on duck typing (something behaves in a desired way <-> it is an instance of some type) for testing, kind\_of? and responds\_to are far better friends then instance\_of when changing the type of a parameter or refactoring.

### Push don’t pull

Always try to push information and configuration into your objects, this way they will be easier to reuse and easy to fill with mock servers/service providers/databases/… . It should be a warning sign when some Objects need to load a configuration file or pull information from the environment using something like CFG[RAILS\_ENV][@option]. Try to make all these connections with the environment as loose as possible, by pushing them in using the constructor or explicit setters (“dependency injection”).

## fixtures

#### Valid fixtures

Only use valid fixtures. Invalid fixtures, usable for testing edge-case behavior, will increase the number of fixtures you have to maintain and make maintaining fixtures harder, since it is hard to distinguish invalid fixtures from broken fixtures.

Only keeping valid fixtures, and change them inside the ‘failure behavior’ test cases. This way you need fewer fixtures, and you keep your concerns (failure cause and failure handling) close by. Furthermore when all your fixtures should be valid, it is easy to run a task to check that all fixtures are indeed valid. This saves time when introducing new validations or fields, to the model.

**def** test\_user\_should\_not\_be\_valid\_without\_email

assert\_invalid users(**:no\_email**)

**end**

* Without invalid fixtures, code gets longer, but we save 1 fixture

**def** test\_user\_should\_not\_be\_valid\_without\_email

u = users(**:one**)

u.email = ''

assert\_invalid u

**end**

A rake task to validate all models (and all fixtures)

$ rake db:validate\_fixtures

-- records - model --

1x FtpAccount

22x Movie

Movie: id=5

["Title can't be blank"]

8x Order

24x Rating

1x User

Sample output: 1 movie does not have the required title -> correct this fixture and re-test

#### Many Fixtures

When you need an enormous amount of fixtures, there 2 solutions:

* Create them inside the test 🡪def create\_100\_fixtures…
* Use fixture\_scenarios, which allows to have fixture-scenarios in subfolders, which than get combined with your normal fixtures. The project is not very well maintained, look through the PATCHes in the issues section to get help when trouble strikes.

## Testing views

Try not to.

Why? Because it is cumbersome to test views with assert\_tag or assert\_select(verifying that some tags exist), and result in your tests breaking when you change your view layout or text.

The question is: What are the things that break? For me these are always mistyped functions 🡪 errors the I see when rendering the view or CSS problems(check the view by browser) or syntax errors (found via html\_test). For me, most of the time things that break undetected are errors inside view-logic.

To test the logic in the views, remove it from the views and place it in your helpers, where it naturally belongs. Testing your helpers will be easier, since most of the time they only return a line of text or a link. From my experience, the only things worth testing are:

* Form has all fields required for an update/create
* Valid HTML
* Page contain only valid links
* Valid layout 🡪 see for yourself in the browser

The first 3 can be tested with form\_test\_helper (5.2.5) and html\_test (5.2.7). When there is still logic left, either ignore it, or only test for the most relevant result of this logic(i.e. a link to go to the admin page is shown or not:   
assert\_equal 1,tags("[@href=#{admin\_url}]").size

Simple demonstration using RSpec and 2 Users:

it "sould have one row for each user" **do**

render '/users/index'

response.should have\_tag('table tr', :count => 2)

**end**

Is wrong and unnecessary:

* because the table has 3 rows (2 Users + Heading)
* tr is always inside a table, or it would raise a syntax error (html\_test (5.2.7)).

Is hard to maintain, since I do not assert that 2 users where rendered, but that 2 rows exists, which could contain anything and nothing.

* When I add another heading, I have to change the test
* When I add a table to describe the contents, I have to find a new selector
* When I refactor using divs, everything changes
* Wrote <% user.email %> instead of <%=…%>(nothing <-> email) but everything passes

## It’s not a bug, it’s a Test case

When something goes wrong or you get an email saying “when I do this, it shows me that”, it is the perfect opportunity to build a test case with this and see it fail. And in the case that it does not fail, you get a new test case for free and have further narrowed down the problem.

Bug to Testcase (5.2.3)

# Testing Tools

## installation Tips

Document all plugins and gems you install, to be sure how it is made and help coworkers.

### Plugins

There are 3 ways to install a plugin, 2 of them are not recommended! So let us sort them from bad to good.

#### Using script/plugin –x or svn:external

It sounds good, to always have the latest version of your plugin, but the path to hell is paved with good intentions. When the author decides that a ‘bit of testing’ is enough for his new feature or the API changes 1 day before you install your system to the server, you will have no clue why suddenly all your code breaks.

(Also changes in the plugin cannot be saved or committed. So every new checked out version will have to be changed again, ore on this subject can be found at )

#### Using plain script/install

This is by far the most often used method, but it has drawbacks. In case you hacked or patched the plugin, an upgrade to the newer version with a new ‘script/plugin install’ will remove all svn information, meaning you have to svn rm;svn commit;svn add. In addition changes will be deleted, and have to be made into a patchfile and then reapplied.

#### Piston

A combination of all the advantages can be found in piston. You check the source code into your SVN, and keep up to date with the latest plugin revision. When you run ‘piston update’ it will work like a normal SVN update, and warn you about conflicts and merges inside the plugin.

gem install --include-dependencies piston

svn update #before each import, to stop piston from complaining

piston import svn://errtheblog.com/svn/plugins/will\_paginate \

vendor/plugins/will\_paginate

piston update vendor/plugins/will\_paginate #get new version

* ‘piston status’ to see which repositorys have changed and which plugins you modified
* ‘piston convert /folder/’ convert an existing svn:externals folder into a piston-managed folder
* ‘piston lock /folder/’ lock some of your piston-managed folders, so that all others can be updated normal, but the one plugin whose new version does not work, stays the same

It will also work with any other SVN hosted library.

### Gems

Add config.gem "GemName" to your Rails::Initializer and to install missing gems using   
rake gems:install or rake gems:unpack. If the name of the gem does not equal the required file, you have to add the library too: config.gem "SQS", :version => "0.1.7",:lib => 'sqs'.

## Tools

### cache\_test

To know if a page is cached or that the cache was expired is helpful when using complicated sweeper (cache expiry) logic. (Copy the extracted archive into plugins)

Now we can make sure that e.g. the index and the show actions are expired when a model gets deleted.

**def** test\_expiring

assert\_expire\_pages("ratings ", "ratings/1") **do** |\*urls|

delete "ratings/1"

**end**

**end**

### Red Green

sudo gem install RedGreen

RedGreen is a very simple plugin, it changes your test results from boring white/black to red/green.

rg test/unit/rating\_test.rb

Loaded suite test/unit/rating\_test

Started

....

Finished in 2.358123 seconds.

4 tests, 13 assertions, 0 failures, 0 error

To make it work in all your rake tests, insert “require ‘redgreen’” into your test\_helper.rb.

### Bug to testcase (laziness)

A very simple way to convert most bugs to a test cases.

script/plugin install http://svn.extendviget.com/lab/laziness/trunk

Now every failing request will print a small test case to repeat this request, it is not much, but it is a starting point and can be helpful if you have a lot of parameters being passed.

Sample output:

**def** test\_get\_rating\_edit\_should\_not\_raise\_activerecord\_recordnotfound\_exception

assert\_nothing\_raised(ActiveRecord::RecordNotFound) **do**

get **:edit**, {"id"=>"1"}, {**:user\_id**=>**nil**, **:return\_to**=>"/orders/1"}, {}, {"\_session\_id"=>["…"]}

**end**

**end**

### ZenTest

sudo gem install ZenTest

#### Autotest

When working with continuous integration there is nothing better than continuous testing. With Autotest this is easy. Autotest will run the matching Testsuite for every file you save. Save user.rb, it runs units/user\_test.rb, the only thing you have to do is watch your console from time to time to see if it is green.

Finished in 55.106864 seconds.

84 tests, 219 assertions, 0 failures, 0 errors

================================================================================

Editing movie\_controller.rb

/usr/bin/ruby1.8 -I.:lib:test -rtest/unit -e "%w[test/functional/movie\_controller\_test.rb].each {

|f| require f }" | unit\_diff -u

Loaded suite –e

#GROES BILD

But it does not end here. Autotest will re-run only the failed tests, so you can narrow down your problem while having fast feedback. It also strips most of the useless error output (framework-trace) and leaves the worst for you: your own code and callstack. After the last failure has been removed, autotest runs the complete suite again, to see if any new errors have been introduced.

Or you can even stop looking at the output and make Autotest play a sound if it fails or raise a desktop notification .

#### Test::Rails

Test::Rails introduces new testing possibilities, helpers, assertions and Test classes. We will have a look on the most potent parts of this Framework. More in-depth knowledge can be found at

Test::Rails supports the idea of separating functional tests into controller and view tests, which follows the basic principles of TDD, to test as small as possible. When we modify the views, the controller tests should not fail and vice-versa when the controllers change, the view-tests should not fail. To test the interaction of views and controllers (e.g. submitting forms) is left for the integration tests, where they naturally belong.

##### Test::Rails::ControllerTestCase

The ControllerTestCase is only responsibe for assuring that all values get assigned (assert\_assigned), that the right model actions were called (save/create…) and flash/session are set correctly. It is a great place to use mocking (see Mocha 5.2.8) since you do not need to know all attributes of an object(since they are never displayed), and can separate your controllers from all validation logic (test failing actions by mocking @your\_model.valid? to false).

Notice:

* Remove @response, @request, @controller lines from your setup and call super
* When your controller does not follow the naming of your ControllerTest add this into setup:

**return if self**.class.to\_s.tabelize == File.basename(**\_\_FILE\_\_**,".rb")

**@controller\_class\_name** = File.basename(**\_\_FILE\_\_**,"\_test.rb").classify

More info on this topic can be found at

##### Test::Rails::ViewTestCase

This type of TestCase sits between a controller and its view. You provide the parameters for your view, and test only what the view does with the given input.

* Test edge-case behavior of your views without building all the support-code that would normally be needed.
* Test partials, which are used by many views, with varying input.
* Test just your layout, by rendering it with an empty content.  
  render :text => '', :layout => 'application'

ViewTestCase comes with many specialized assertions like assert\_links\_to, assert\_post\_form, assert\_input …

**class** MovieViewTest < Test::Rails::ViewTestCase

**def** test\_delete

assigns[**:loggedin\_user**] = users(**:herbert**)

assigns[**:movie**] = movies(**:two**)

render #render the view

assert\_submit form\_url, 'Delete!'

**end**

**end**

#IMG

This approach is time-consuming when you only want to test a simple action and its resulting view. But the more complex your controllers get, and the more edge cases your views have to handle, the more appealing this approach becomes. You can always use a mixture of functional and pure controller/view tests, but once you started, it is wise to switch all functional to controller/view tests, since then you can use rails\_test\_audit.

##### rails\_test\_audit

When used in combination ViewTestCase, ControllerTestCase supplys a new feature: running rails\_test\_audit will show you which variables have been tested for in the controller (by assert\_assigned), that have not been supplied in the ViewTestCase (by assign) and vice versa.

### form\_test\_helper

Form\_test\_helper helps you to submit a form that was created in a previous request and through this see if your form works as it is supposed to be. It also reduces the work to test a post request, by using the values already filled in the form.

script/plugin install http://form-test-helper.googlecode.com/svn/form\_test\_helper/

#IMG

Example: filling a form and submitting it. (more at)

submit\_form **do** |form|

form.movie.title = 'Test movie'

form.movie.plublic.uncheck #checkbox handling

**end**

#OR

movie = {‘title‘=>‘Test movie‘,‘public‘ =>**false**}

submit\_form {|form| form.movie.update(movie)}

assert\_response :success

With the help of some syntactic sugar it is even possible to click a link.

#test/test\_helper.rb

**def** click(text)

select\_link(text).click

**end**

This will validate that the link exists and make the correct request.

Notice: if you are on revision 69 you need to apply this patch for click to work with post/put/delete links.

#vendor/plugins/form\_test\_helper/lib/form\_test\_helper.rb

-if self["onclick"] && self["onclick"] =~ /'\_method'.\*'value', '(\w+)'/

- $1.to\_sym

+if self["onclick"] && self["onclick"] =~ /\.method = '(.\*)'/

+ $1.downcase.to\_sym

### RailsTidy

#installation for Ubuntu 8.04, from your Rails directory

sudo apt-get install tidy

sudo gem install tidy

wget http://www.cosinux.org/~dam/projects/rails-tidy/rails\_tidy-0.2/tidy.patch

sudo patch /var/lib/gems/1.8/gems/tidy-1.1.2/lib/tidy/tidybuf.rb < tidy.patch

cd vendor/plugins/

wget http://www.cosinux.org/~dam/projects/rails-tidy/rails\_tidy-0.2.tar.bz2

tar -xf rails\_tidy-0.2.tar.bz2

rm rails\_tidy-0.2.tar.bz2

More installation-details: http://www.cosinux.org/~dam/projects/rails-tidy/doc/

RailsTidy can be used to validate existing rhtml templates like this:

rake test:templates

/home/data/projekte/short/app/views/movie/list.rhtml ERRORS

/home/data/projekte/short/app/views/movie/edit.rhtml OK

To get a more detailed look, we append FILE=path/to/file

rake test:templates FILE=app/views/movie/list.rhtml

app/views/movie/list.rhtml ERRORS

line 6 column 1 - Warning: <br> element not empty or not closed

line 10 column 1 - Warning: <table> lacks "summary" attribute

Info: Doctype given is "-//W3C//DTD XHTML 1.0 Transitional//EN"

Info: Document content looks like XHTML 1.0 Transitional

2 warnings, 0 errors were found!

Some of the errors result from using html helpers like form\_tag, so this output can be a good pointer, but is not to be trusted in general.

Inside your tests you can use assert\_tidy, which can see the whole HTML and is more accurate.

Tidy detected html errors in response body:

------------------HTML of whole page----------------------

line 65 column 33 - Warning: inserting implicit <p>

line 65 column 33 - Warning: trimming empty <p>

#IMG

I you do not want to be harassed with warnings like “lacks "summary" attribute” you can easily configure RailsTidy to only show warnings you accept, by modifying config/tidy.rc. A list of possible configuration options can be found at .

The best way to ensure continuouse nagging is inserting assert\_tidy into your teardown method.

#test\_helper.rb

**class Test::Unit::TestCase**

**def** teardown

assert\_tidy **if @response**

**end**

**end**

### Html\_TEST

* use up to 3 validators
* validate every test - request
* suppress useless warnings
* check all URLs (no 404)

script/plugin install http://htmltest.googlecode.com/svn/trunk/html\_test

To use the HTML validation, you need to first install RailsTidy (or any other of the 3 supported validators).

#insert into test/test\_helper.rb

#validate every request

ApplicationController.validate\_all = **true**

ApplicationController.validators = [:tidy]

#ignore common warnings

Html::Test::Validator.tidy\_ignore\_list = [

/<table> lacks "summary" attribute/,

/trimming empty <fieldset>/,#erros\_on missing -> empty fieldset

/line 1.\*Warning: inserting missing 'title' /,#redirect html has no title....

/Warning: replacing invalid character code 130/, #€ not supported…

]

#check urls

ApplicationController.check\_urls = **true**

ApplicationController.check\_redirects = **true**

A simple setup with RailsTidy: Suppressing basic warnings and checking every test request.

It is also possible to check URLs on a production server by letting html\_test check all links. The problem with this approach is that you cannot specify which warnings to ignore, resulting in a lot of useless warnings.

vendor/plugins/html\_test/script/validate http://my.blog.com --validators tidy

### Mocha

Mocha is a Mocking / Stubbing framework, which helps to separate controller and view tests from database and validations. With Mocha you can separate all model specific aspects (validations and environment requirements) from unit tests. Therefore the controller and integration tests can validate pure request/response logic, without worrying that e.g. an EmailAccount needs a User to be saved. All complex or time-consuming model and database operations can be skipped.

script/plugin install svn://rubyforge.org/var/svn/mocha/trunk

Basic test to verify that the correct record is assigned.

**def** test\_edit\_should\_assign\_rating

rating = Rating.new

Rating.expects(**:find**).with('1').returns(rating)

get **:edit**, **:id** =>'1'

assert\_equals rating, assigns[**:rating**]

**end**

If all your test cases use Mocha, stubbing can be placed inside the setup.

**def** setup

**@rating** = Rating.new

Rating.stubs(**:find**).returns(**@rating**)

**end**

**def** test\_edit\_should\_assign\_rating

get **:edit**, **:id** => **1**

assert\_equals **@rating**, assigns(**:rating**)

**end**

The difference is: The first test will fail if find is called with 2, or find is not called or find is called twice. Whereas the second test only verifies that the rating is assigned, not caring from where it came.

Notice:   
If you stub one find ((:find).with(:all)), you have to stub all other finds (find :first, find 13…) since any other call to find, that does not match one of your stubs, will make your tests fail.

### RCov

RCov is a code coverage analyzer for ruby, with a rails plugin, that can be used to test your code, without learning all RCov command line options. The result is a html file for every file tested, with detailed execution coverage, and a index.html where you can see the overall coverage for all tested files.

wget http://rubyforge.org/frs/download.php/28270/rcov-0.8.1.2.tar.gz

tar -xf rcov-0.8.1.2.tar.gz

sudo ruby rcov-0.8.1.2/setup.rb

rm -rf rcov-0.8.1.2

script/plguin install http://svn.codahale.com/rails\_rcov

#IMG

Now you can get started with “rake test:units:rcov”

Parameters:

* Show only selected parts: SHOW\_ONLY=m,l,c,v (model,lib,controller,view)
* Run a single tests: rcov FILE --rails (the --rails option will ignore config and environment)
* Add RCov parameters: RCOV\_PARAMS=""
* Sort by coverage: COV\_PARAMS="--sort=coverage"
* Hide fully covered: RCOV\_PARAMS=" --only-uncovered"

If you see something like (eval):580:in `attribute’: wrong number of arguments (2 for 0) (ArgumentError) it means one of your tests is uncoverable, this can happen while using SQS gem and is a pain to debug.

Ignoring all standard libraries helps keeping the output to a manageable size. When using the command line, add RCOV\_PARAMS="-exclude ‘/var/|/usr/’ ” , /var/ and /usr/ being the folders where your libraries lie. When using rake, this can be done the same way, but is a bit repetitive. For this purpose it can be simpler to modify the plugin.

#vendor/plugins/rails\_rcov/tasks/rails\_rcov.rake @ line ~60

**if** show\_only.any?

reg\_exp = ['/var/','/usr/'] #CHANGED

**for** show\_type **in** show\_only

#IMG

## RSpec

### RSpec on Rails

script/plugin install git://github.com/dchelimsky/rspec.git

script/plugin install git://github.com/dchelimsky/rspec-rails.git

script/generate rspec

RSpec focuses on BDD (Behavior Driven Design). When you are doing TDD right, you are doing BDD. The problem RSpec is trying to solve lies mostly in syntax and task splitting. Normal test verify that something is the way it should be, whereas the idea behind of TDD is that you state what should work and then make it work. So RSpec uses a syntax that focuses on ‘should’ and not assert.

Another main part is the story framework, which makes it possible to let your customers write and review the tests, by using normal English sentences as steps. Once a step is programmed, it can be combined freely in any number of stories.

RSpec uses different words for test related things: test=spec, test case=example, integration test=story (we will only use stories for integration testing, but they also could be used for other test areas)

#### Syntax

Read like English:

assert\_equal ‘new’, rating.status #Test::Unit

@rating.status.should == ‘new’ #RSpec

#### Stories

The most simple and stunning part of RSpec, a story.

Story: transfer from savings to checking account

As a savings account holder

I want to transfer money from my savings account to my checking account

So that I can get cash easily from an ATM

Scenario: savings account has sufficient funds

Given my savings account balance is $100

And my checking account balance is $10

When I transfer $20 from savings to checking

Then my savings account balance should be $80

And my checking account balance should be $30

To transform this story into a test case, we need to define what ‘Given my savings account balance is $100’means in program terms.

steps\_for(**:accounts**) **do**

Given("my $account\_type account balance is $amount") **do** \

|account\_type, amount|

create\_account(account\_type, amount)

**end**

When("I transfer $amount from $source\_account to $target\_account") **do** \

|amount, source\_account, target\_account|

get\_account(source\_account).transfer(amount).to(get\_account(target\_account))

**end**

Then("my $account\_type account balance should be $amount") **do** \

|account\_type, amount|

get\_account(account\_type).should have\_a\_balance\_of(amount)

**end**

**end**

#IMG

When we have a scaffold for all sentences used, we can run any number of test cases with them. This way an executive or designer can write stories, and later the programmer can fill them with life.

If you just like the story feature, but want to stay with the Test::Unit syntax, using ‘asserts’ in your specs is possible, but not recommended.

#### Models, Controllers, Views, Helpers

But stories are not everything. They are just the top part, the former integration tests. For all other ‘small’ parts we user normal RSpec test cases. These are split into 4 different categories: Model, Controller, View, Helpers. Controllers and views are the former parts of a functional test, like Test::Rails, controller tests only verify that i.e. an object was assigned and a view was rendered, whereas a view test gets predefined assigns and verifies that it renders correctly. Only when used with integrate\_views, controller tests behave like functional (Test::Unit) tests.

RSpec test work like normal unit/functional tests and are a good point of starting to learn RSpec syntax. A helpful learning recourse is the RSpec cheat sheet and the RSpec converting table .

A basic controller spec looks like this, not so different from Test::Unit.

describe RatingController **do**

integrate\_views #render views

fixtures **:ratings**

before(**:each**) **do**

**@rating** = ratings(**:one)**

**end**

it 'should pass' **do**

**true**.should be\_true

**end**

**end**

(Requiring the controller or catching and re-raising errors is no longer necessary and the 3 old @controller, @request, @response statements can be left out too.)

#### Stub and mock

RSpec contains a mocking framework.

The syntax is not as clean as Mochas, but it offers comparable abilities.

**@rating** = mock("rating")

**@rating**.stub!(**:new\_record?**).and\_return(**false**)

Rating.stub!(**:new**).and\_return(**@rating**)

Rating.new.should\_not be\_new\_record

If you want to stay with Mocha mocking, put this into your spec/spec\_helper.rb.

Spec::Runner.configure **do** |config|

config.mock\_with **:mocha**

**end**

#### Running RSpec

There are 2 ways of running RSpec. Specs can be run direct via rake “rake spec:models”, where you can use models/controllers/views/helpers/plugins and app/all. “all” = “app” + “plugins”. Test execution can be slow, since the whole Rails framework has to be loaded before tests can be run. For faster test execution, you can host a spec server.

script/spec\_server –d #run server as deamon (kill:‘ps –A’,‘kill ID’)

echo ‘--drb’ >> spec/spec.opts #add –drb to spec options

rake spec #run spec as normal

#INSTALL

When the spec server is running, testing no longer requires starting up the Rails framework, and should therefore be faster.

#### Running Stories

Getting started with stories is rather hard, since there is no generator to help, and the default helper.rb and all.rb are not very helpful.

helper.rb – when placing all stories in stories/stories/{name}.txt and all steps in stories/steps/{name}.rb

ENV["RAILS\_ENV"] = "test"

require File.expand\_path(File.dirname(**\_\_FILE\_\_**) + "/../config/environment")

require 'spec/rails/story\_adapter'

#load all steps

Dir[File.join(File.dirname(**\_\_FILE\_\_**), "steps”, ”\*.rb")].each **do** |file|

require file

**end**

#run a story

**def** run\_story(file\_name)

run File.join(File.dirname(**\_\_FILE\_\_**),"stories”,”#{file\_name}.txt"), **:type** => RailsStory

**end**

The actual runner lies in stories/{name}.rb and is executed with ‘ruby stories/{name}.rb‘ Example:

require File.join(File.dirname(**\_\_FILE\_\_**),"helper")

with\_steps\_for **:rating**, **:general do**

run\_story 'rating'

**end**

### RSpec converter

Converting old Test::Unit test cases by hand can be very helpful in learning RSpec syntax, but once you understood how it works, it is a painful task.

script/plugin install \ http://svn.davidjrice.co.uk/svn/projects/plugins/test\_unit\_to\_rspec\_converter

rake convert\_to\_rspec

Notice: it will overwrite all specs, that happen to be where the new specs should be created.

With the converter, all your old tests will be mostly converted to RSpec. Not all old asserts need to be replaced, since RSpec can work with Test::Unit assertions. But before everything runs you need to do a bit of hand work:

1. Copy old code from test/test\_helper.rb to spec/spec\_helper.rb (leave include‘s outside of Spec::Runner.configure do)
2. Change spec/spec\_helper.rg ‘config.fixture\_path =‘ to test/fixtures OR copy all fixturesfrom test to spec (svn cp test/fixtures spec/fixtures)
3. Correct syntax errors in converted files

When all you tests run successfully, remove the old fixtures and see if they still run. When everything works, commit and start refactoring.

### test\_spec

For those that like RSpec syntax, but can or will not make the switch, there are 2 alternatives:

#### test\_spec

Test\_spec is an RSpec like syntax replacement for Test::Unit. It allows mixing of RSpec coding style and normal tests.

sudo gem install test-spec

script/plugins install \

http://svn.techno-weenie.net/projects/plugins/test\_spec\_on\_rails/

#test/test\_helper.rb

require 'test/spec/rails'

Syntax cheat sheets can be found at http://cheat.errtheblog.com/s/test\_spec/. Syntax example:

**@rating**.should.validate #assert @rating.valid?

response.should.be.redirected #assert\_response :redirect

#### Using RSpec with Test::Unit

When normal Test::Unit files are run as a spec, they behave exactly like before, but can use should / should\_not etc. for validation. So drop all your functional tests into spec/controllers, copy the fixtures, replace all test/xxx with spec/xxx and your tests should pass as if nothing has happened.

## Javascript

### Javascript\_test

With javascript\_test it is possible to run JsUnit tests automatically in up to 4 different browsers. To use this feature you have to install all browsers you need (currently supported: IE and Firefox for Windows, Firefox and Safari for Mac, Firefox and Konqueror for Linux). On Windows all browsers must be installed in their default location and tests in Internet Explorer have to be closed by hand. It is hard to get javascript\_test running properly, but if it works you can save a lot of repetition.

script/plugin install http://dev.rubyonrails.org/svn/rails/plugins/javascript\_test/

The plugin also allows you to generate a javascript test scaffold, with:

script/generate javascript\_test my\_app

create test/javascript/my\_app\_test.html ...

This gives you a good starting point for test setup and organization, even if you run your tests by hand (loading them in the browser) and do not automate them.

### UnitTest.js

A test framework build using prototype.js, it is easy to get started and has graphic output.

Test setup:

...

<script src="path/to/prototype.js" type="text/javascript"></script>

<script src="path/to/unittest.js" type="text/javascript"></script>

...

<!-- Log output -->

<div id="testlog"> </div>

<!—Elements to test on-->

<div>Test text</div>

<script type="text/javascript" language="javascript">

new Test.Unit.Runner({

setUp: function(){},

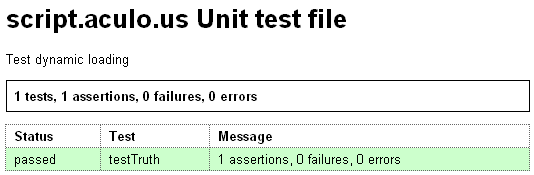
testTruth: function () {

this.assertEqual(2, 1+1);

}

})

</script>

Result:  
 

More information can be found here: http://wiki.script.aculo.us/scriptaculous/show/Test.Unit.Runner a live demo can be found here: http://sunnyfortuna.com/scripts/test/run\_unit\_tests.html

To get started, go to http://sunnyfortuna.com/scripts/test/unit/loading\_test.html and save the page to your Desktop, it includes every JS and CSS file you need.

### jqUnit

Jqnit is a testrunner that was originally developed by the jQuery team, and was given TestCase support as well as some basic anonymity by Colin Clar and me. Its current home is http://code.google.com/p/jqunit/ it hopefully will be integrated into jQuery core soon.

JqUnit is originally not a xUnit framework, for example assertrEquals(1,output()) is written as equals(output(),1).It still supports all standard xUnit commands. JQUnit uses the full power of javascript, by breaking out of xUnit conventions.

* The whole page will be reset after each test, so there almost never is a setup/teardown needed
* Click an test output line to unfold Test – Results (see each assertion)
* Double Click to rerun the selected Test
* test.html?xxx -> run only tests that match /xxx/

It is possible to use TestCase or to just start writing tests with test().

with(jqUnit){

module('With local interface');

test('test 1', function(){

ok(true);

});

};

//same Using TestCase

var t = new jqUnit.TestCase('TestCase',function(){

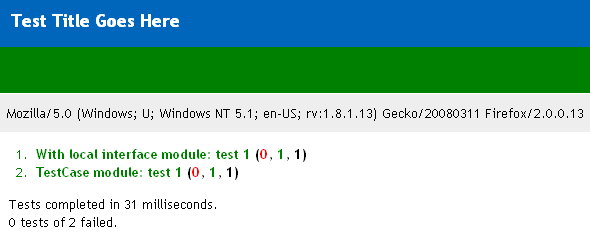
/\*setup\*/

},function(){

/\*teardown\*/

});

t.test('test 1',function(){with(jqUnit){ this.ok(true) }});

Result:  


The advantage using TestCase is that setup and teardown will run each time you run a t.test .

To get started, download the demo and source packaged from the homepage.

### newjs

Newjs is a Javascript project generator using UnitTest.js. It is build to work independent of Rails. Use it when you are building your own standalone Javascript library.

gem install newjs

Generate the whole testing infrastructure with:

#!Do not run it in RAILS\_ROOT, but e.g. in public/javascripts!

newjs mylib -a "myName" -e "myMail" -u "http://mylib.rubyforge.org"

script/generate unit\_test mylib

#IMG

It combines javascript\_test with an automated project generation, upload to rubyforge and some other extras, all without relying on Rails infrastructure.

More examples and deeper knowledge can be found at http://newjs.rubyforge.org/.

### Selenium

script/plugin install <http://svn.openqa.org/svn/selenium-on-rails/selenium-on-rails>

For detailed installation instructions and ‘getting started’ help, visit http://selenium-on-rails.openqa.org/ .

Selenium is a tool for browser-automation. It comes with a Rails plugin and an IDE to create Unittests simply by surfing your site and hitting the record button. Recorded surfing sessions can then be automated inside a normal Unittest using Seleniums ‘Remote Control’ Browser API. All that is left to the developer is inserting the right assertions for the content that is displayed and for the Data that is saved to the database.

Using Selenium is not as easy as it sounds. Often user interactivity cannot be reproduces since the Javascript does complicated interactivity that depends on e.g. mouseovers.

To test your website automatically, you will need the Selenium Remote Control Server. It runs on any operating system and will execute your test calls. This way testing of any browser on any system is possible.

When testing, the loading of the website will often ‘hang’, pressing F5 helps. Also there are a few issues that hopefully get sorted out with 1.0, e.g. not starting in Firefox, not being able to ‘press\_key’ in IE or not starting a second time.

Tips:   
Pressing ‘return’ is “ key\_press locator, ‘/13’ “  
Locators can be ids(#search = ‘search’) or CSS (input with name search=“//input[@name='search']”)

To see if Selenium is the right tool for you, start with the standalone Ruby Client that comes with the Remote Controll Server. Start the server with “java –jar selenium-server.jar” and then run your tests.

require 'test/unit'

require 'selenium'

**class** ExampleTest < Test::Unit::TestCase

include SeleniumHelper

**def** setup

ff = '\*firefox E:\-PROPGRAMME\-INTERNET\FireFox\firefox.exe'

ie = '\*iexplore C:\Programme\Internet Explorer\IEXPLORE.EXE'

url = "http://www.ebay.com/"

**@selenium** = Selenium::SeleniumDriver.new("localhost", **4444**, ie, url, **6000**);

**@selenium**.start

**end**

**def** teardown

**@selenium**.stop

**end**

**def** test\_small

open "/"

type 'satitle', 'all'

click '//input[@value=\'Search\']'

sleep **7**

assert\_equal('My Ebay', get\_text('MyEbay'))

**end**

**end**

Simple sample Selenium TestCase.

In conclusion, Selenium is a powerful tool, that’s hard to setup and maintain, but it allows simultaneous browser interaction on many remote servers. And it is the only tool so far, that delivers this ease of TestCase construction with the integrated Selenium IDE and clients in 7 programming languages.

## Helpers and Tasks

### invalid\_attributes

A helper to test validations can be found at tinyurl.com/6krjox [[2]](#footnote-3) it allows syntax like:

assert\_invalid\_attributes User, **:login** => ['',**nil**], **:email** => [**nil**,'','ohno']

And gives failure messages like:

<User.email> expected to be invalid when set to <ohno>

To test that ‘password is not the same as a username’ or ‘password does match the password\_confirmation’, define a @valid\_attributes = {'username'=>'foo'…} hash in the setup.

### agiledox

With AgileDox you can convert your test cases into documentation and insert this documentation into the corresponding models/controllers, to always have a reference when developing.

Copy into lib/tasks from http://code.google.com/p/agiledox-rake/

Then you can run rake dox or rake spec:models:dox or rake test:units:dox to get your tests displayed in this format:

A User:

- should not be valid without login

- should not be valid without email

A Users Controller's:

'new' action:

- should succeed

'edit' action:

- should succeed

A /users/edit:

- sould show errors

Changing the options :write value to true (in agiledox.rake) it will also write these comments:

#AGILEDOX !WILL BE OVERWRITTEN!

#A User:

# - should create valid user

# - should stop invalid

#AGILEDOX END

**class** User < ActiveRecord::Base

...

**end**

### Single Test

#IMG

Copy code from (Grosser: single, 2008) and make a new single.rake file in lib/tasks

One of the most useful and easiest plugins, just run one file or one test at a time.

rake test:rating:should\_rate\_

🡪 run every test that matches /should\_rate\_/ in test/units/rating.rb

rake test:ratings\_C

🡪 run test/functionals/ratings\_controller.rb

#IMG

To run a controller, you can append \_C or supply the filename HERE: rating\_C or rating\_controller.[[3]](#endnote-2)It also works nicely with RSpec, where the syntax translates to spec:\*.

## Plugin Recourses

http://wiki.rubyonrails.org/rails/pages/Testing+Plugins

http://svn.techno-weenie.net/projects/plugins/

http://dev.rubyonrails.org/svn/rails/plugins/

http://wiki.rubyonrails.com/rails/pages/plugins

http://topfunky.net/svn/plugins

http://www.agilewebdevelopment.com/plugins/index

...

# Example TDD

Now comes the time to get your hands dirty, we will start by building a small application using traditional Test::Unit with normal functional tests. Halfway through we will convert to RSpec and finish by separating our functional tests into Controller and View tests with support from Mocha.

Requirements engineering, Class model, Interaction graph and Activity analysis follow… no, I am just kidding, we will build something with users, let’s go!

## Basic User tests

To make thing fast and short, we rely mostly on scaffold code, and we will only test things we build ourselves.

$ rails tdd-example

$ script/generate scaffold user

#change config/routes.rb (map.root :controller => "users")

#delete public/index.html

Before doing anything, we build some basic unit tests, to sort out what our model needs. Install assert\_invalid\_attributes and then continue with:

**class** UserTest < ActiveSupport::TestCase

fixtures **:users**

**def** setup

**@user** = users(**:one**)

**@valid\_attributes** = {

'login'=>'newuser',

'email'=>'new@email.com',

}

**End**

it 'should create valid user' **do**

User.create!(**@valid\_attributes**)

**end**

it 'should stop invalid' **do**

assert\_invalid\_attributes User, **:login** => ['',**nil**], **:email** => [**nil**,'','ohno']

**end**

it 'should not allow duplicated fields' **do**

User.create!(**@valid\_attributes**)

assert\_invalid\_attributes User, **:login** => **@user**.login, **:email** => **@user**.email

**end**

**end**

When we try to see the tests fail, we get information that we first need to run migrations, so be it.

create\_table **:users do** |t|

t.string **:email**, **:login**, **:null** => **false**

t.timestamps

**end**

$ rake db:migrate

$ rake test:user #see 5.5.3 or use rake test:units

Test results: 6 tests, 5 assertions, 5 failures, 0 errors, looking good.

To satisfy our tests, we modify user.rb

**class** User < ActiveRecord::Base

validates\_presence\_of **:email**, **:login**

validates\_format\_of **:email**, **:with** => /@/

validates\_uniqueness\_of **:email**, **:login**

**end**

Test results: 6 tests, 5 assertions, 0 failures, 0 errors nice and green.

Now we turn to the controller, install form\_test\_helper 5.2.5 (optional: railstidy 5.2.6, html\_test 5.2.7).

Notice: if you use Rails 2.0.2, you cannot use the generated UsersControllerTest < ActionController::TestCase, since it does not support setup. Rewrite it using Test::Unit::TestCase.

Test result of then generated tests: 7 tests, 11 assertions, 1 failures, 0 errors ? Why do they fail? Some quick investigation shows that create fails, since we already changed our model validations. So first we fix that.

Now we can define some scenarios where validations fail and result in a different template being rendered.

**def** test\_create\_should\_fail\_without\_login

get **:new**

submit\_form **:user** => valid\_attributes('login')

assert !assigns(**:user**).valid?

assert\_response **:success**

assert\_template 'new'

**end**

#similar for update

Test result: 9 tests, 17 assertions, 0 failures, 2 errors, both failures are ‘Field named 'user' not found in FieldsHash’, meaning that our form has no room for login/email. Making them pass is easy:

app/views/users/edit.html.erb and new.html.erb:

<% form\_for(@user) do |f| %>

<%= render :partial => 'form', :locals =>{:f => f}%>

app/views/users/\_form.html.erb

<fieldset>

<label for="user\_login">Login</label> <%= f.text\_field 'login' %> <br/>

<label for="user\_email">Email</label> <%= f.text\_field 'email' %> <br/>

</fieldset>

Test result: 9 tests, 23 assertions, 0 failures, 0 errors

## Converting to RSpec

Now we install RSpec 5.3, RSpec Converter + patch 5.3.2.

Test Results (using rake spec): 15 examples, 0 failures, don’t worry, that there are not 23+6 examples. RSpec counts each test case, unlike Test::Unit which counts every assertion.

If your results are not as green as they should be, you are in for a little bug-hunt…

Now I will delete all old tests, to not overwrite my specs in case I happen to run convert\_to\_rspec again. Then I will have a look at the specs and refactor anything that could be enhanced.

assert\_difference('User.count', -**1**) **do**

delete **:destroy**, **:id** => users(**:one**).id

**end**

* in create and delete

lambda {

delete **:destroy**, **:id** => users(**:one**).id

}.should change(User,**:count**).by(-**1**)

assigns(**:user**).valid?.should\_not be\_true

* in many examples, makes our intent clearer

assigns(**:user**).should\_not be\_valid

## Splitting the controller

Now we will split the users\_controller\_spec into Controller, View specs and stories. For this we need Mocha 5.2.8. And then uncomment spec/spec\_helper.rb # config.mock\_with :mocha.

Even if I argued against view tests, we will briefly cover them for the sake of completeness. We do not want to decrease our test coverage and keep everything running, so we work on the view tests first, and then the stories and then strip any view/integration related testing from the controllers.

### Views

Will give us a view test for index in spec/views/users/index.html.erb\_spec.rb which we change to look like this:

require File.dirname(**\_\_FILE\_\_**) + '/../../spec\_helper'

describe "/users/index" **do**

fixtures **:users**

before(**:each**) **do**

assigns[**:users**]=[users(**:one**),users(**:one**)]

**end**

it "sould have enought a row for each item" **do**

render '/users/index'

response.should have\_tag('tr',**:count**=>**2**+**1**) #2items + heading

**end**

**end**

$ script/generate rspec\_controller users index

Result of rake spec:view: 1 example, 0 failures.

I am not using mocks or stubs here, since mocking all fields of a Record is too complicated. But for some cases like method calls on an object or special behavior, it is easy to just add e.g. user.stubs(:new\_record?).return(true) to test edge cases.

### Stories

See RSpec:Running Stories 5.3.1 for getting started help.  
Our runner lies in stories/run.rb and is executed with ‘ruby stories/run.rb‘

require File.join(File.dirname(**\_\_FILE\_\_**),"helper")

require File.join(File.dirname(**\_\_FILE\_\_**),'form')

with\_steps\_for **:users**, **:general do**

run\_story 'users'

**end**

For form interaction (using form\_test\_helper plugin 5.2.5) we need stories/form.rb

#re-selecting the form means loosing field-data

**def** form(selector=**nil**)

**@form** ||= select\_form selector

**end**

#reset the form object on each submit

**def** form\_submit

**@form**.submit

**@form** = **nil**

**end**

#enter a nested value into the form

**def** form\_enter type, valid\_attributes

valid\_attributes.each { |field,value| form.send("#{type}[#{field}]=",value)}

**end**

And finally our user story is in stories/stories/users.txt

Story: User interaction

Scenario: protect from creating a user without email

When i go to users/new

And i enter valid attributes except email

And i submit

Then i should stay at new

And the user should have an error on email

Scenario: create a new user

When i go to users/new

And i enter valid attributes

And i submit

Then i should be redirected to the user

Now that everything is set up and we can start making our story do something useful. At the moment, when we run it, we only see a nice reminding ‘(PENDING)‘ behind each line of the story, to tell us “not build yet”.

There are only 2 kinds of ‘sentences’ in this story, Then and When, all ‘And’ sentences are the same type as the one before them. Now let us write the steps, that our story needs stories/steps/users.rb.

valid\_attributes = {'login'=>'newuser', 'email'=>'new@email'}

steps\_for(**:users**) **do**

When("i enter valid attributes") **do**

form\_enter 'user', valid\_attributes

**end**

When("i enter valid attributes except $attribute") **do** |attribute|

form\_enter 'user', valid\_attributes

form.user[attribute]=''

**end**

**end**

This seems rather few, and you are right, all general steps that I want to reuse are in stories/steps/general.rb

steps\_for(**:general**) **do**

When("i go to $url") **do** |page|

get page

**end**

When("i submit") **do**

form\_submit

**end**

Then("i should stay at $url") **do** |url\_or\_action|

response.should render\_template(url\_or\_action)

**end**

Then("the $record should have an error on $field") **do** |record,field|

assigns(record).errors.on(field).should\_not be\_nil

**end**

Then("i should be redirected to the $record") **do** |record|

response.should redirect\_to(send("#{record}\_path",assigns(record)))

**end**

**end**

With only 7 steps, from which only 2 are ‘user’ centered, we covered 9 story sentences. A very good reuse quota!

### Mocking in the controller

Now the form interaction is covered inside our stories, we can remove it from our controllers. We start mocking all database actions, so we no longer have to worry about whether the controller logic failed, or the object was not saved properly: real “Controller unit tests”, without any view or database interaction.

Since we will mix mocked/unmocked and remove ‘integrate\_views’ from our tests, most of them would fail. To not get flooded with failure messages, I usually refactor and test one test method at a time:   
rake spec:user\_C:’create should succeed’ (using the Single Test rake script).

First we define some general stubs (using Mocha):

* We no longer need our @valid\_attributes, since we no longer care if something is valid or not.
* We stub find 2 times, since find can be called with a string (params[:id]) or an integer(user\_path(@user))

before(**:each**) **do**

**@user** = users(**:one**)

User.stubs(**:find**).with(users(**:one**).id).returns(**@user**)

User.stubs(**:find**).with(users(**:one**).id.to\_s).returns(**@user**)

**end**

All other invocations of calls to User will be mocked inside the tastcase that uses it, e.g. index:

User.expects(**:find**).with(**:all**).once.returns([])

For each action we test the success (and failure) route to see if the controller reacted appropriately. We do not test if a record is valid or that the User.count was increased, since we now eliminate all database concerns.

it 'destroy should succeed' **do**

**@user**.expects(**:destroy**). once.returns(**true**)

delete **:destroy**, **:id** => users(**:one**).id

response.should redirect\_to(users\_path)

**end**

it 'create should succeed' **do**

expect\_new

expect\_save **true**

#user\_path(@user) = /users/ --> /user/1

**@user**.excpects(**:new\_record?**).times(2).returns(**false**)

post **:create**, **:user** => {}

flash[**:notice**].should\_not be\_nil

response.should redirect\_to(user\_path(**@user**))

**end**

it 'create should fail' **do**

expect\_new

expect\_save **false**

post **:create**, **:user** => {}

flash[**:notice**].should be\_nil

assigns(**:user**).should\_not be\_nil

response.should render\_template('new')

**end**

The converted test names stay almost unchanged, so that comparison for you is easy.  
:new\_record is called twice, since we use it ourselves with user\_path.

The helpers excpect\_new and excpect\_save are rather simple:

**def** expect\_new

**@user** = User.new

User.expects(**:new**). once.returns(**@user**)

**end**

**def** expect\_save val

**@user**.excpects(**:save**).once.returns(val)

**end**

## Starting from Scratch

Now comes the time to build a new feature truly test-first, as opposed to the last example were we most of the time only converted, translated and refactored existing code.

User story: When users behave badly, it should be possible to deactivate them and reactivate them if necessary. This implies:

* A new user is activated by default
* A user should be deactivateable
* A deactivated user should be activateable
* Index should only show activated users

### Integration test

Since we have a nice user story, we can build a new Integration test.

Scenario: deactivating a user

When i go to users

And i change the first to activated = false

And i go to users

Then i do not see the first

I did not use the whole form submission process, since it is already covered by our other stories, so we can proceed without having “to go to edit”, “change the form” and “submit the form”.

Write the missing steps in stories/steps/users.rb

When("i change the first to $field = $value") **do** |field,value|

response.body.should have\_tag('tr a') **do** |links|

**next if** (links.to\_s !~ /\/(\d+)\/edit/)

**@first\_id** = $1

User.find(**@first\_id**).update\_attributes({field => value})

**end**

**end**

Then("i do not see the first") **do**

response.body.should have\_tag('\*') **do** |all|

all.to\_s.should\_not =~ /\/#{**@first\_id**}[^\d]\//i

**end**

**end**

N

The have\_tag statement cannot simply use (‘\*’, regex) since what we want to match is an attribute of an ‘a’ tag and will not be found, since it only searches for inner texts.

Add the new form attribute to the valid attributes:

valid\_attributes = {

… 'activated' =>"1", …

}

And see it fail 3 scenarios: 2 succeeded, 1 failed, 0 pending.

### Model tests

We build our model test cases:

it 'should be activated by default' **do**

user = User.create!(**@valid\_attributes**)

user.activated.should be\_true

**end**

it 'should only find activated' **do**

User.activated(**:all**).should\_not be\_empty

User.activated(**:all**).reject(&**:activated**).should be\_empty

**end**

See it fail: 8 examples, 2 failures, build the migration:

$ script/generate migration add\_activated\_to\_users

**class** AddActivatedToUsers < ActiveRecord::Migration

**def self**.up

add\_column "users", "activated", **:boolean**, **:default** => **true**

**end**

**def self**.down

remove\_column "users", "activated"

**end**

**end**

Migrate db:migrate RAILS\_ENV=test and test again: 8 examples, 1 failure,  
alter the model:

**def self**.activated \*args

with\_scope(**:find** => { **:conditions** => { **:activated** => **true**}}) **do**

find(\*args)

**end**

**end**

and test again: 8 examples, 0 failures, hurray!

### Controller tests

Activated will only be another field in the user form, so create and save have it already covered, but we need to make sure that activated is used, so we change the controller test for index.

User.expects(**:activated**).with(**:all**).once.returns([])

See it fail 9 examples, 1 failure and make it pass:

**def** index

**@users** = User.activated(**:all**)

…

Then we run our controller tests: 9 examples, 0 failures.

### View tests

There is no test for the \_form, but in my opinion it does not need any. All that I want to verify is that every field has a corresponding form element and we already test this with our stories.

We still have to fix the app/users/\_form.html.erb by adding:

<label for="user\_activated">Activated</label>

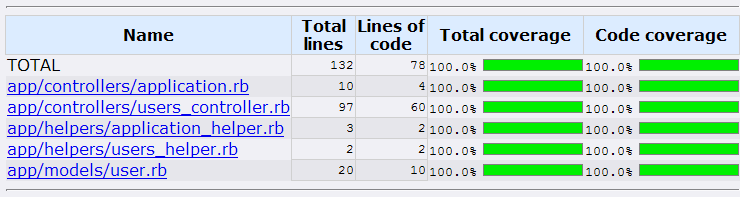
<%= f.text\_field 'activated' %> <br/>

### Integration Test Continued

Now we come to the final step and run our stories and see them pass   
3 scenarios: 3 succeeded, 0 failed, 0 pending.

### RCOV

Finally we run rake spec:rcov to see if we left anything out (modify spec/rcov.opts if there is too much garbage in the output). And see nice and green results.



# Conclusion

Rails test-friendly architecture allows to test far better, deeper and smaller than ever before. The Model and Controller Test-helpers are good and usable. But for testing views, JS and CSS more tools and plugins are required. Especially RSpec, a tool with fast rising popularity, changes the way tests are written. When all aspects of the application from model to Javascript are covered with tests, that run automatically, the developer can easily see the consequences of changes. And therefore can work fast, reliable and efficient.

Martin Fowler: "There's an impossible-to-express quality about test-first design that gives you a sense of unhurriedness. You are actually moving very quickly, but there's an unhurriedness because you are creating little micro-goals for yourself and satisfying them. At each point you know you are doing one micro-goal piece of work, and it's done when the test passes. That is a very calming thing. It reduces the scope of what you have to think about. You don't have to think about everything you have to do in the class. You just have to think about one little piece of responsibility. You make that work and then you refactor it so everything is very nicely designed."

# Literaturverzeichnis

Bscofield: Laziness. (2008, 10 12). Retrieved from http://github.com/bscofield/laziness/tree/master: http://github.com/bscofield/laziness/tree/master

continuouse. (2008). Retrieved 06 15, 2008, from http://continuous.rubyforge.org/form\_test\_helper/rdoc/: http://continuous.rubyforge.org/form\_test\_helper/rdoc/

Eric. (2006). Retrieved 06 12, 2008, from http://www.verticalexpressionsoftware.com/blog/2006/06/17/converting-to-using-testrails/: http://www.verticalexpressionsoftware.com/blog/2006/06/17/converting-to-using-testrails/

err: evil twin. (2007). Retrieved 10 12, 2008, from http://errtheblog.com/posts/67-evil-twin-plugin: http://errtheblog.com/posts/67-evil-twin-plugin

fixture-scenarios. (2007). Retrieved 10 12, 2008, from http://code.google.com/p/fixture-scenarios/: http://code.google.com/p/fixture-scenarios/

Fowler, M. (2008, 08 4). Retrieved from http://www.artima.com/intv/testdrivenP.html: http://www.artima.com/intv/testdrivenP.html

Foz works. (2007). Retrieved 10 12, 2008, from http://www.fozworks.com/2007/7/28/autotest-sound-effects: http://www.fozworks.com/2007/7/28/autotest-sound-effects

Grosser. (2008). Retrieved 06 06, 2008, from http://code.google.com/p/jqunit/: http://code.google.com/p/jqunit/

Hale, C. (2006). Retrieved 10 15, 2008, from http://blog.codahale.com/2006/05/26/rails-plugin-rails\_rcov/: http://blog.codahale.com/2006/05/26/rails-plugin-rails\_rcov/

Hanrigou, P. (2006). Retrieved 06 05, 2008, from http://ph7spot.com/articles/getting\_started\_with\_autotest: http://ph7spot.com/articles/getting\_started\_with\_autotest

Indeed: Ruby. (2008, 10 11). *indeed.com*. Retrieved from http://www.indeed.com/jobtrends?q=ruby&relative=1&relative=1

Mocha. (2008). Retrieved 10 15, 2008, from http://mocha.rubyforge.org/: http://mocha.rubyforge.org/

Neil: Arts. (2007). Retrieved 10 12, 2008, from http://www.railsforum.com/viewtopic.php?pid=55887: http://www.railsforum.com/viewtopic.php?pid=55887

page-cache-test. (2006). Retrieved 10 12, 2008, from http://blog.cosinux.org/pages/page-cache-test: http://blog.cosinux.org/pages/page-cache-test

Piston. (2008, 10 13). Retrieved from http://piston.rubyforge.org/: http://piston.rubyforge.org/

Pragmatig: valid\_attr. (2008, 10 12). Retrieved from http://pragmatig.wordpress.com/2008/10/03/dry-validation-testing-and-easy-edge-case-records/: http://pragmatig.wordpress.com/2008/10/03/dry-validation-testing-and-easy-edge-case-records/

quircksmode.org. (2006). Retrieved 06 05, 2008, from http://www.quirksmode.org/dom/getstyles.html: http://www.quirksmode.org/dom/getstyles.html

Red Green. (2007). Retrieved 10 13, 2008, from http://robsanheim.com/2007/07/24/did-redgreen-get-borked/: http://robsanheim.com/2007/07/24/did-redgreen-get-borked/

Rice, D. J. (2007). Retrieved 06 08, 2008, from http://www.davidjrice.co.uk/articles/2007/8/12/ruby-on-rails-plugin-test-unit-to-rspec-converter: http://www.davidjrice.co.uk/articles/2007/8/12/ruby-on-rails-plugin-test-unit-to-rspec-converter

RSecp Convertion. (2008). Retrieved 10 15, 2008, from http://rspec.info/documentation/test\_unit.html: http://rspec.info/documentation/test\_unit.html

RSpec. (2008). Retrieved 10 15, 2008, from http://rspec.info/: http://rspec.info/

RSpec Cheat. (2007). Retrieved 10 15, 2008, from http://blog.daveastels.com/files/QuickRef.pdf: http://blog.daveastels.com/files/QuickRef.pdf

Sanheim, R. (2006). Retrieved 06 12, 2008, from http://robsanheim.com/2006/12/29/bdd-in-rails-testspec-and-rspec/: http://robsanheim.com/2006/12/29/bdd-in-rails-testspec-and-rspec/

tidy. (2008). Retrieved 06 20, 2008, from http://tidy.sourceforge.net/docs/quickref.html: http://tidy.sourceforge.net/docs/quickref.html

Wikipedia: Ruby. (2008, 10 11). Retrieved from wikipedia.org: http://en.wikipedia.org/wiki/Ruby\_(programming\_language)

ZenTest. (2006). Retrieved 10 12, 2008, from http://zentest.rubyforge.org/ZenTest/classes/Test/Rails.html: http://zentest.rubyforge.org/ZenTest/classes/Test/Rails.html

1. https://addons.mozilla.org/en-US/firefox/addon/60 [↑](#footnote-ref-2)
2. http://pragmatig.wordpress.com/2008/04/04/testing-validation-the-dry-way/ [↑](#footnote-ref-3)
3. [↑](#endnote-ref-2)