Successful and Maintainable testing of Database Backed Applications

Jason Gessner

Performics/DoubleClick

YAPC::NA 2006 – June 27, 2006

Overview

- You are not doing enough testing
- Your app doesn’t want to be tested.
- You have 1001 reasons that it will be hard
- Testing fact vs. fiction
- Once you start you won’t stop, but getting started can be hard.
- Once you embrace it, your velocity will skyrocket.

You are not doing enough testing

- We have all been there.
- “I tested it”
- Prove it.
- Clicking around in your interface counts…sorta.
- Each person, no matter how bright, has a narrow view.
- In most cases, you don’t control your code. Other users will find other problems.

Your app doesn’t want to be tested.

- As soon as you write a line of code, a module, or a web application, you have made your life harder.
- Testing is not free. Each change you make without testing in mind pushes you further from your goal.
- The longer your code has not been tested, the steeper your climb will be.

1001 reasons why it will be hard

- SQL is spread throughout the application.
- The web app only works in IE
- Our database is too large to setup and teardown.
- I didn’t write this 5000 line module!!!
- These scripts are black boxes
- There is no documentation
- The application is time-dependent.

Where do you start?

- Start at the beginning (if it is new)
- or
- Start where you are the most comfortable (if it exists already).
- Let’s assume it exists already.

Gauging where you are at:

- Make a list of the pieces of your application:
  - Models (list each)
  - Interface Pages (plus their options or conditions).
  - Utility scripts
    - How many, what do they do, what kind of options do they take?

How well covered are you?

- Note how confident or how much coverage you have for each component.
- Rate the following areas:
  - Statement Coverage – in most cases, can be generated with Devel::Cover
  - For command line scripts, list how many tables they touch and how many modules they use.

Rate Your Risk

- What is the system that would destroy or cripple your business if it failed?
- What supports that system?
- That is a good candidate to start.

A Quick Gut Check

- How well is your code organized right now?
- Do you use an ORM (Class::DBI(x), Tangram, something homegrown?)
  - This can sometimes make things easier.
- Do you have DBI->connect() statements all over the codebase?
  - If so, pause here.
- Testing needs infrastructure. Start with your DBI calls.

Organizing DBI calls

- If you have DBI connect calls littered all over your code base, you need to get that changed first.
- Your use of DBI can be the first seam you introduce into your code. (define seams).
- If you have 100 DBI->connect calls, you have 100 hardcoded pieces of configuration in your code.

Subclassing DBI

- Subclassing DBI can give you control over an implementation of DBI, similar to an interface in other languages.
- If you can treat DBI as an interface rather than a concrete class then you immediately gain a leg up on testing.
- Testing can now be done by subclassing and changing configuration, not by changing code.

DBIFactory

- DBIFactory is a little example module that gives you a hook to configure what classes to use for your DBI work.
- Basically, you can use a class like this to check a config file or an environment variable or something to determine what class to return instead of DBI.
- Look at t/30subclass.t in the DBI distro and perldoc DBI for more info on subclassing DBI.

DBIFactory Example

```
package Example::DBIFactory;

use strict;
use warnings;

use ConfFile;
use Memoize;
memoize('get_class_name');

sub get_class_name {
    # to minimize changes, default to DBI
    my $DBI_class = 'DBI';

    my $conf_filename = get_conf(
       '/environment/dbi_factory.conf‘
    );
```

DBIFactory Example

```
 if ( !$ENV{FORCE_DBI} && -e $conf_filename ) {
        my $dbi_factory_conf = ConfFile->new(
  $conf_filename );
        if ( $dbi_factory_conf ) {
            $DBI_class = $dbi_factory_conf->{dbi_class};
            print STDERR "Setting our DBI class from
  dbi_factory.conf to [$DBI_class]\n";
        }
    }

    eval "use $DBI_class";

    die "Could not load DBI class [$DBI_class]: $@"
        if $@;

    return $DBI_class;
}
```

DBIFactory Example

```
=head2 connect()

connect() will actually call the configured class' connect
   method.  Simple.

=cut

sub connect {
    my $class = shift;

    my $DBI_class = get_class_name();

    return $DBI_class->connect(@_);;
}
# repeat for the other needed methods…
```

DBIFactory example()

- Some possible uses
  - An All Success Factory
  - An All Failure Factory (won't get too far).
  - A random failure factory.
  - An expected results factory:
    - SQL fragments and method types could be read from a config file and matches could be trigger expected results (success, failure, certain data returned).
  - A logging Factory could generate test files that generate testing code to replay a given session or produce a customized version of the trace info from DBI.
  - A rollback factory that gives you convenience around wrapping your tests in a transaction and rolling back changes.
  - A hybrid of all the above. Traits?

Factory Benefits

- Centralized seam for your database.
- You can control the ins and outs from a single location.
- More flexibility to replace the implementation for testing, debugging and production purposes.
- Doesn’t mean you can’t bring in DBD::Mock or any other DB testing methodology.

DBIFactory usage

```
use Test::More qw(no_plan);

$ENV{FORCE_DBI} = 1;
use_ok(“Example::DBIFactory", “Example::DBIFactory can be
   used!");
ok( my $dbh = Example::DBIFactory-
   >connect("dbi:mysql:host:db", “user", “pass") );
isa_ok($dbh, "DBI::db", "\$dbh isa DBI::db");
#####################################################
use Test::More qw(no_plan);

$ENV{FORCE_DBI} = 0;
# conf has dbi_class = Example::SubclassedDBI
use_ok(“Example::DBIFactory", “Example::DBIFactory can be
   used!");
ok( my $dbh = Example::DBIFactory->connect(
"dbi:mysql:host:db", “user", “pass") );
isa_ok($dbh, “Example::SubclassedDBI::db", "\$dbh isa
   Example::SubclassedDBI::db");
```

Testing the DBI Factory

```
$ prove -v factory_test*
factory_test.....
ok 1 - use Example::DBIFactory;
Setting our DBI class from dbi_factory.conf to
   [Example::SubclassedDBI]
ok 2
ok 3 - $dbh isa Example::SubclassedDBI::db isa
   Example::SubclassedDBI::db
1..3
ok
factory_test_forced....ok 1 - use Example::DBIFactory;
ok 2
ok 3 - $dbh isa DBI::db isa DBI::db
1..3
ok
All tests successful.
Files=2, Tests=6,  0 wallclock secs ( 0.14 cusr +  0.01
   csys =  0.15 CPU)
```

Using your new Factory

- Now, once you have something like this, you shouldn’t run off and change every piece of application code to use it.
- Still start with something you can get your hands around.

Mocking and its limitations

- Now, a slight diversion.
- A while ago I picked a module in our code base to try out mocking with.
- This work became the example for several mocked tests in our code base.
- The module is simple
  - Used by a polling process that looks for records that need to be sent to an external web service.
  - If the call succeeds, the record is updated.
  - If not, an error is emitted.

Where mocking was a huge help

- With a couple of hours of trial and error I eliminated 75% of my dependencies on a live database for testing.
- I ended up with a mixture of Test::MockObject and Test::MockModule.
- The app is already configured with a conf file (service URLs and methods).
- This needed to have methods added so I could modify it on the fly.

The holy grail

- After each tweak I made, I was running Devel::Cover reports.
- The numbers crept closer and closer to 100%.
- I even threw in a couple of ridiculous test cases to force it to 100%.
- 100% coverage!!!

Mocking Code Example

```
my $dbi = Test::MockObject->new();
$dbi->fake_module( 'DBI',
    new => sub  {return $dbi;}
);
my $sth = Test::MockObject->new();
$sth->fake_module( 'DBI::st',
    new => sub  {return $sth;}
);
$sth->mock( execute         => sub { return 1; } );
$sth->mock( state           => sub { return 0; } );
$sth->mock( finish          => sub { return 1; } );
$sth->mock( fetchrow_array  => sub { return qw(a b c d e f
   g);; } );
$dbi->mock( prepare         => sub { return $sth;} );
$orm->mock( handle          => sub { return $dbi; } );
```

Mocking Code Example

```
# fake our LWP objects
my $response    = Test::MockObject->new();
$response->fake_module( 'HTTP::Response',
    new => sub { return $response; }
);
$response->mock( is_success => sub { return 1; } );
$response->mock( content    => sub { return "Some content that will make this work."; } );
my $lwp = Test::MockObject->new();
$lwp->fake_module( 'LWP::UserAgent',
    new         => sub { return $lwp; }
);
$lwp->mock(
    request     => sub { return $response; }
);
```

Mocking Test

```
SampleService::set_error("");
ok( SampleService::do_thing(12345, 54321,"10099"),
  "Attempting to do_thing with all successes
  mocked." );
SampleService::set_error("");
ok( SampleService::do_other_thing(12345,
  54321,"10099"),
  "Attempting to do_other_thing with all successes
  mocked." );
```

Mocking Test

```
my $mocked_service_error = "Drats!  Foiled again!";
$response->mock( content => sub { return "Error:
  $mocked_service_error"; } );
SampleService::set_error("");
ok( !SampleService::do_thing(12345, 54321,"10099"),
  "Forcing a service error.");
is( SampleService::error(),
    "Found error: $mocked_service_error",
    "Service error message correct." );
```

Mocking saved the day!

- One day we saw some funny errors in the logs for the application that hosts the web service.
- A coworker was convinced that this app was misbehaving when this error message came up.
- “Send me the exact HTTP response body and I will add it to the test suite.”
- 5 minutes later I had a new test that watched the processing with this exact text as the response.
- The app behaved as expected.

Mocking saved the day part 2

- This example is not designed to point a finger at my coworker or to pat myself on the back.
- This test meant the difference between me saying, “No, this app is working correctly.” and “I don’t think that should do anything. No, no WAY could this app be misbehaving in that case.”
- Had I said the latter, I would have eaten my words in a few days or a week. Again.

Mocking, why hast thou forsaken me?

- A couple of months later we made some large scale schema changes to our application.
- When it came time to test this application we realized something horrible.
- Our test suite with 100% coverage of this module was worthless for verifying this case!
- We had written away our ability to add a test to this suite to verify that the app worked under the new schema.

Some tests need reality

- This application was simple enough that the world did not end because we couldn’t add this test to our suite.
- What it did do is point out what changes we need to make to let this app be testable from a disconnected/mocked setup and from a connected setup.
- This module had several package level variables that held prepared sql statements.
- These were prepared at use time for the module.

Reality Check

- Package level variables are not evil.
- But they are a pain.
- For this module, we can move the use time inits to a method, or a series of methods, that prepares each statement.
- This lets us mock out the ones we want to ignore.
- It also lets us more safely verify which statements succeed and fail.
- Our test can move from use_ok(“MyModule”) to:
  - Ok(MyModule::prepare_statement1())l
  - Is_ok(MyModule->statement1(), “dbi::st”, “Local variable is of the right type”);
  - # repeat x 10.

Initializing Databases

- Sometimes your app is small and you can reinitialize on the fly for each test suite run…
- Mysql> drop database MYAPP;
- Mysql> create database MYAPP;
- Mysql> grant privileges, etc.
- prove –v MyTestSuite/

Different Strokes for Different DBs

- Doing this with DB2 or Oracle is much, much slower.
- Doing this generally involves DBAs, not application developers.
- All of this adds up to time, overhead or red tape.
- This can mean the difference between a suite that can be run over and over, or something that needs to be scheduled.

Dealing with Big Databases

- So what can be done?
- Careful Cleanup
- Hardware Snapshots
- Virtual Machine Snapshots

Careful Cleanup

- In theory, you can wrap your application in a transaction and roll it back at the end.
- In reality, you are testing active, new code, which means that you will break stuff.
- Sometimes your db structure will make tearing down tests cumbersome or error prone.
- But it can be done.

How much do you need?

- At work, our databases are in the hundreds of gigabytes range.
- We have 8 of them, each with slightly different distributions of data.
- For the most part we need a few hundred megs from that at most.
- But sometimes we want it all.
- At least we think we do.

Hardware snapshots

- Use a filesystem that supports snapshots
- Use a network storage device that supports snapshots.

Virtual Machine Snapshots

- If you go down the virtualization route, this can be an excellent solution.
- If you have a virtual machine in 5 different states, you can fire it up at each given state before your test runs.
- The speed of this may be an issue.
- You could also cheat and keep read only images around and swap them out.

VMWare

- VMWare supports snapshots in its workstation product.
- Even without that, you can use the free product and an existing tool to set yourself up.
- Make your base image.
- Place a read only copy somewhere and copy in and make changes when needed.

Virtualization Benefits

- Firing up a paused VM takes a very short amount of time.
- The environment can be tricky to set up.
- If you are testing, though, SOMEONE is doing that setup already.
- Don’t throw that time away!
- Snapshotting the database may not be perfect, but if it save you 2 days of setup for a given release, it is worth it.

Automated new files

- Best practices for creating CPAN distros says to use Module::Builder or at least ht2xs.
- This isn’t all that helpful for a module that is part of a larger application.
- But it shouldn’t be a real barrier.
- A developer on my team created something that at least automated creating a test skeleton for every module in our app.

Test Generators

- We have a class called TestFileGenerator.
- This is used by a command line script that takes a path or a filename as an argument.
- It can generate and execute test files.
- In execute mode, it seeks out all of the Test files in a given path and executes them.
- This uses Test::Class and a more xUnit style of testing, but still uses the Test::More stuff for assertions.

Test Class

```
package BrandSpankingNew;
use warnings;
use strict;
use base qw(DateTime);
use Class::DBI;
use DBI;

sub some_awesome_function {
    print "Oh yeah!\n";
}

sub some_other_function {
    print "I am different.\n";
}

1;
```

Autogenerated Test 1

```
package BrandSpankingNew::Test;
use strict;
use warnings;

use base qw( TestBase );

use Test::More;
use Test::MockObject;
use Test::MockObject::Extends;
use Test::MockModule;
use Test::Exception;
```

Autogenerated Test 2

```
sub my_startup : Test(startup=>3) {
# code here will execute once for the entire class...
   don't forget to update the test count if you add any
   tests here
     use_ok('Class::DBI');
     use_ok('DBI');
     use_ok('BrandSpankingNew');
}
sub my_shutdown : Test(shutdown){
     # cleanup code for your startup
}
sub my_setup : Test(setup) {
     # code here will execute before every test in this
   class
}
sub my_teardown : Test(teardown) {
     # cleanup code for your setup
}
```

Autogenerated Test 3

```
sub auto_generated_tests : Tests {
    my $obj;
    if ( BrandSpankingNew->can('new') ){
        $obj = BrandSpankingNew->new() or do{  diag(
  "skipping autogenerated test for BrandSpankingNew
  :".BrandSpankingNew->error());  return; };
        isa_ok( $obj, 'BrandSpankingNew');
    } else {
        $obj = "BrandSpankingNew";
    }
    my @methods = qw(some_awesome_function
  some_other_function );
    can_ok( $obj, @methods );
}
```

Test Results

```
jgessner perlcode $ ./perltest/rt path/to/module

RUNNING:BrandSpankingNew::Test
ok 1 - use Class::DBI;
ok 2 - use DBI;
ok 3 - use BrandSpankingNew;
not ok 4 - auto_generated_tests died (Mandatory parameter
   'year' missing in call to DateTime::new
#  at
   /opt/dynamic/jgessner/perltest/search/lib/BrandSpankingN
   ew/Test.pm line 39)
#   Failed test 'auto_generated_tests died (Mandatory
   parameter 'year' missing in call to DateTime::new
#  at
   /opt/dynamic/jgessner/perltest/search/lib/BrandSpankingN
   ew/Test.pm line 39)'
#   in -e at line 1.
1..4
# Looks like you failed 1 test of 4.
```

What does this buy us?

- Gets people over the initial hump.
- Places tests in a consistent location.
- Gives the tests a consistent structure and style (at least to start). ☺

Things we could do with this.

- Have this generate your skeleton package as well as your skeleton tests.
- Add pod coverage and correctness tests for each module.
- If your modules are all based around a certain ORM, generate basic tests for each attribute and CRUD function.
  - Part of this will be redundant, but it will help catch things if people are only running individual tests instead of the whole suite (and stuff is broken). ☺
- Again, for the ORM example, add checks for the existence of the database tables and any columns that are defined.
- If you use a consistent methodology to do object validation, read your db definition and compare it to your object.

Setting up data

- Now that you have more test modules, and a safe place to test them, you need some data to test against.
- What you want are fixtures.

Fixtures in Rails

```
# Read about fixtures at
   http://ar.rubyonrails.org/classes/Fixtures.html
fred:
  id: 1
  name: fred
  hashed_password: <%=
   Digest::SHA1.hexdigest('abracadabra') %>
jason:
  id: 2
  name: jason
  hashed_password: <%= Digest::SHA1.hexdigest('changeme')
   %>
```

Rails Fixtures

- Rails fixtures are YAML files that are processed by their template engine and then converted into objects via ActiveRecord.
- Rails generates these for you when you generate a new model.
- Catalyst may as well.

How do we use fixtures?

- We have a set of fixture packages.
- These have named subroutines that create an object, save it and return it.
- Some of these are simple
- Some take arguments to configure them to be more applicable for one test case or the other.
- They aren’t fancy.
- But they are centralized and easy to read.

Fixtures don’t need ORM

- If you have applications that are file driven, then the files can become your fixtures.
- Someone on our team wrote a DSL to generate a certain kind of crucial file for us.
- The DSL includes data generation and assertions.

Test File DSL

- **TEST NAME ‘Upate Games'**

- **DEFINE FIXTURE zelda_fixture**
  - **PRINCESS**
    - **SET name TO ‘Zelda'**
  - **HERO**
    - **SET name TO ‘link'**
    - **SET weapon TO ‘master sword'**
- **END FIXTURE**

- **DEFINE FIXTURE mario_fixture**
  - **PRINCESS**
    - **SET name TO ‘Peach'**
  - **HERO**
    - **SET name TO ‘mario'**
    - **SET weapon TO ‘star'**
- **END FIXTURE**

Test File DSL

```
DEFINE HEADER TEMPLATE header1
    SET date TO “20060627"
    SET action TO ‘Update'
    SET platform TO ‘Nintendo DS'
END TEMPLATE

DEFINE BODY TEMPLATE b1
    SET weapon TO ‘rail gun'
    SET villain TO ‘bowser'
END TEMPLATE
```

Test File DSL

```
DEFINE RUN 'required field tests'
    # If you only have one header or fixture, that one
  will be assumed.
    USE FIXTURE zelda_fixture
    USE HEADER h1

    DEFINE CASE ‘wrong weapon for link'
        # order is important...use, change, check
        USE BODY b1

        CHECK FILE FIELD return_message IS ‘Wrong Weapon
  for Link'
        CHECK DB RECORD DOES NOT EXIST
    END CASE
```

Test File DSL

```
DEFINE CASE ‘wrong villain for link'
    # order is important...use, change, check
    USE BODY b1

    CHECK FILE FIELD return_message IS ‘Wrong Villain
for Link'
    CHECK DB RECORD DOES NOT EXIST
END CASE
```

Test File DSL

```
DEFINE CASE ‘good record for link'
    # order is important...use, change, check
    USE BODY b1

    CHECK FILE FIELD return_message IS 'SUCCESS'
    CHECK DB RECORD EXISTS
    CHECK DB FIELD hero IS ‘link’
    CHECK DB FIELD weapon IS ‘master sword’

END CASE
```

Soft Benefits

- Starting testing, even in a small way can help quantify the problems you may not know you have.
- Testing forces you to think about better software development, not just what you need right now.
- Testing can get your team excited.

Starting Small

- My Mocking example was not the most crucial piece of application code we have.
- It was a representative sample, though.
- It also got us over the mocking hump.
- Testing it full was achievable in a couple days.
- Each piece of your application will introduce new challenges to your testing methodology.

Testing and Team Momentum

- Once a group is bitten by the testing bug progress will be made.
- Tradeoffs are constantly happening, though.
- The more success testing shows, the more those tradeoffs will favor safety and sustainability.

Cautionary Notes

- As much as you want to sell testing to a team, you need to be careful about how you sell it outside your group.
- You may have found a way to automate the testing for a critical piece of infrastructure….
- That doesn’t mean that it has been done.
- Or that it was done right / perfectly.
- 100% statement coverage is not your goal.