Testing with PostgreSQL

Hi, I'm Shawn.

- Polyglot Programmer
- First love is perl
- UNIX Geek
- Lots of database experience
- Currently on assignment with All Around the World

- First heard the term Polyglot Programmer during a talk by John Anderson and found it described me
- Developed in Ruby, Python, Java, Groovy, JavaScipt, C#, PHP, various SQL dialects, dabbled in Go and of course perl
- I've been coding in perl since 1997
- started when I picked up AIX and needed to automate some tasks
- haven't stopped automating since then

What is it?

- Test::PostgreSQL
- PostgreSQL runner for tests
- Test::PostgreSQL automatically sets up a PostgreSQL instance in a temporary directory, and destroys it when the perl script exits.
- Currently maintained by Toby Corkindale

- described as a test runner however it doesn't execute any test code
- Test::PostgreSQL allows testing with a throw away database
- The 'test' here refers to not production or permanent
- I have been known to spin them up for debugging from within reply

How does it work

- Wrapper around the **pg_ctl** PostgreSQL controller
- Assigns a port if unused starting at **15432**

- pg_ctl is used by init scripts to initialize, start, stop, or control a PostgreSQL server
- standard PostgreSQL port is 5432
- Automatically increments if the port is in use (15433)
- can spawn multiple instances

Sqitch Basics

Testing with Postgresql // Toronto Perl Mongers // Shawn Sorichetti

Why?

- Sqitch gives us a methodology to setup and tear down test databases that match what is is in production.
- App::Sqitch
- Written by David Wheeler
- Works with Firebird, MySQL, Oracle, PostgreSQL, SQLite, Vertica

- Covered last month by Cees, can be covered in more detail in a dedicated talk
- Sqitch is a tool for managing your DDL.
- Sqitch is an application not a module so it can be used with any project, and for numerous databases
- PostgreSQL variants like Netezza and Greenplum should also work

Fast Start

- Initialize sqitch environment

```
sqitch init test_postgresql --engine pg
```

- Creating a migration

```
sqitch add user_table -n 'Add user table.'
```


- Creates `sqitch.conf`, `sqitch.plan`, `deploy/`, `revert/`, `verify/`
- Create the first migration, creating files in the deploy/, revert/, verify/ with the migration name
- A word on running with carton, I received an error message `Cannot find deploy template` the work around was to add `template_directory = local/etc/sqitch/templates` to `sqitch.conf` in the `add` section

Deploying

```
Creates file deploy/user_table.sql


-- Deploy test_postgresql:user_table to pg

BEGIN;

  CREATE TABLE "user" (
    user_id SERIAL PRIMARY KEY,
    user_name TEXT UNIQUE,
    password TEXT
  );

COMMIT;


Executed by issuing:

sqitch deploy db:pg:test_db


Testing with Postgresql // Toronto Perl Mongers // Shawn Sorichetti
```

- Simple user table
- quick note about TEXT columns, in PostgreSQL there is no performance or storage benefit from using a fixed width VARCHAR column, so just use TEXT
- PostgreSQL reserved words, `user` is a reserved word, and will throw an error unless double quoted `"user"`
- while database connectivity can be specified in the `sqitch.conf` file, personal preference is to use it on the command line.

Verifying

```
Creates file verify/user_table.sql.


-- Verify test_postgresql:user_table on pg

BEGIN;

  SELECT 1/COUNT(0)
    FROM information_schema.tables
    WHERE table_name = 'user'
  ;

ROLLBACK;


Executed by issuing:

sqitch verify db:pg:test_db


Testing with Postgresql // Toronto Perl Mongers // Shawn Sorichetti
```

- ROLLBACK verifications can change data, and at the end they will be undone by default.
- Table creation verification is simple, and if it didn't create, there would be a SQL error during deployment.
- More advanced verification for procedures and table alters

Reverting

Creates file **revert/user_table.sql.**

```
-- Revert test_postgresql:user_table from pg

BEGIN;

  DROP TABLE "user";

COMMIT;
```

Executed by issuing:

```
sqitch revert db:pg:test_db
```


- When you need to remove changes
- Reverting is destructive and the only way to go back is through restoring

Verifying during deployment

Executed by issuing:

```
sqitch deploy --verify db:pg:test_db
```

Verify each migration after deploying.

- Overtime changes aren't backward compatible and running `sqitch verify` will break.
- During these times using the `--verify` switch during deployment tells sqitch to verify what it's deploying, not what has already been deployed.

The Test::PostgreSQL Module

Testing with Postgresql // Toronto Perl Mongers // Shawn Sorichetti

Not just for testing!

- While the name suggests test, it's really a temporary PostgreSQL database

DBIC Schema Update Utility

- Helps keep sqitch deployments and DBIC schemas in sync
- Ensures that developer temporary changes don't make their way into DBIC schemas

- Avoid `dbicdump` creating `my_tmp_table_bkup`
- Force truth into sqitch

dbic_update.pl

```
use App::Sqitch;
use App::Sqitch::Command::deploy;
use App::Sqitch::Command::verify;
use Test::PostgreSQL;

my $db = Test::PostgreSQL->new;

my $sqitch = App::Sqitch->new(
  options => {
    engine => 'pg',
  },
);

my $deploy = App::Sqitch::Command::deploy->new(
  sqitch => $sqitch,
  target => $db->uri,
);
$deploy->execute();

say $db->dsn;
my $dsn = $db->dsn;

$dsn =~ s/^dbi/dbi/i;
$dsn =~ s/dbname\=/database=/i;

system "dbicdump -o dump_directory=./lib MyApp::Schema '$dsn'";
```


- Creates a new PostgreSQL database
- Deploys sqitch database migrations to it
- Runs `dbicdump` to create the DBIC schema packages
- This code is going to become familiar

Automating Sqitch for Testing

Requires App::Sqitch version 0.996

- calling sqitch from within a script/module was added in 0.996 with a patch from Chris Prather
- used to error out on a Moose error

Test Sqitch automation

- Creates a test database
- Runs deploy and verify steps
- Reverts changes

- going to create a temporary database
- deploy and verify each step in turn
- test that reverting each step succeeds
- there's no test at the end to make sure the database is back to an empty state

Setting up the test

```
use Test::PostgreSQL;
use Test::Most;

use App::Sqitch;
use App::Sqitch::Target;
use App::Sqitch::Command::deploy;
use App::Sqitch::Command::verify;
use App::Sqitch::Command::revert;
```


- There is the option of executing the command line via `system`, however I find this method cleaner, even though it uses more modules up front
- Test modules and the App::Sqitch internals

Setting up the database

```
my $db = Test::PostgreSQL->new;

my $sqitch = App::Sqitch->new(
   options => {
     engine => 'pg',
   },
);

my $target = App::Sqitch::Target->new( sqitch => $sqitch);
my $plan   = $target->plan;
```


- Calling new on `Test::PostgreSQL` starts the PostgreSQL instance

Setting up Sqitch

```
my $db = Test::PostgreSQL->new;

my $sqitch = App::Sqitch->new(
   options => {
     engine => 'pg',
   },
);

my $target = App::Sqitch::Target->new( sqitch => $sqitch);
my $plan   = $target->plan;
```


- Tell Sqitch that we want to execute using a pg engine
- Sqitch classes are a hierarchy, need to build each individual object to proceed

Getting the Sqitch plan

```
my $db = Test::PostgreSQL->new;

my $sqitch = App::Sqitch->new(
   options => {
     engine => 'pg',
   },
);
```

```
my $target = App::Sqitch::Target->new( sqitch => $sqitch);
my $plan   = $target->plan;
```


- Set the Sqitch target and read in the plan file

Looping through the plan

```
while ( my $change = $plan->next ) {
  my $deploy = App::Sqitch::Command::deploy->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $deploy->execute() } 'deploy sqitch plan to ' . $change->name;

  my $verify = App::Sqitch::Command::verify->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $verify->execute() } 'verify sqitch plan';

  push @revert, $change->name;
}
```


- plan file contents are not necessarily in order
- The `$plan` iterator is used to get the next step in the plan
- Run a deploy on each step via `deploy` command
- `to_change` will process the plan up to and including the specified change

Looping through the plan

```
while ( my $change = $plan->next ) {
  my $deploy = App::Sqitch::Command::deploy->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $deploy->execute() } 'deploy sqitch plan to ' . $change->name;

  my $verify = App::Sqitch::Command::verify->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $verify->execute() } 'verify sqitch plan';

  push @revert, $change->name;
}
```


- Run a verify on each step via `verify` command
- `to_change` will process the verify plan up to and including the specified change

Looping through the plan

```
while ( my $change = $plan->next ) {
  my $deploy = App::Sqitch::Command::deploy->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $deploy->execute() } 'deploy sqitch plan to ' . $change->name;

  my $verify = App::Sqitch::Command::verify->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $verify->execute() } 'verify sqitch plan';

  push @revert, $change->name;
}
```


- Store the change to be able to revert it in reverse order later

Looping through the plan

```
pop @revert;
while ( my $change_name = pop @revert ) {
  my $verify = App::Sqitch::Command::revert->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change_name,
    no_prompt => 1,
  );
  lives_ok { $verify->execute() } 'revert sqitch plan to ' . $change_name;
}
```


- If we try to revert the last change, sqitch will complain that there's nothing further deployed.

Looping through the plan

```
pop @revert;
while ( my $change_name = pop @revert ) {
  my $verify = App::Sqitch::Command::revert->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change_name,
    no_prompt => 1,
  );
  lives_ok { $verify->execute() } 'revert sqitch plan to ' . $change_name;
}
```


- We've built and verified the entire database structure and now we'll tear it down
- Run the revert command on each change

Test Module Skeleton

```
package MyApp::Test::PostgreSQL;

use strict;
use warnings;
use App::Sqitch;

use Moose;
use DBI;
use Test::PostgreSQL;
use DateTime;
use MyApp::Schema;
use App::Sqitch::Command::deploy;

+--  6 lines: has db => (----------------------------------------------------------------------------------

+--  6 lines: has dsn => (----------------------------------------------------------------------------------

+-- 15 lines: has dbh => (----------------------------------------------------------------------------------

+--  9 lines: has dbic => (----------------------------------------------------------------------------------

+-- 11 lines: has sqitch => (----------------------------------------------------------------------------------

+--  7 lines: has fixtures => (----------------------------------------------------------------------------------

+-- 10 lines: has perm_fixtures => (----------------------------------------------------------------------------------

+--  8 lines: sub deploy {----------------------------------------------------------------------------------

no Moose;
__PACKAGE__->meta->make_immutable;
1;
```


- highlevel of the module with methods
- The skeleton provides methods to be used during your testing: `db`, `dsn`, `dbh`, `dbic`
- Now step into each of the methods

MyApp::Test::PostgreSQL::db

```
has db => (
   is       => 'ro',
   isa      => 'Test::PostgreSQL',
   lazy     => 1,
   default => sub { Test::PostgreSQL->new },
);
```


- access to the Test::PostgreSQL object
- not really needed during testing directly

MyApp::Test::PostgreSQL::dsn

```
has dsn => (
   is => 'ro',
   isa => 'Str',
   lazy => 1,
   default => sub { return $_[0]->db->dsn },
);
```


- returns the dsn created by Test::PostgreSQL
- simple string that can be used to create additional connections
- Create new DBI connections, hand over to applications

MyApp::Test::PostgreSQL::dbh

```
has dbh => (
  is      => 'ro',
  lazy    => 1,
  default => sub {
    DBI->connect(
      $_[0]->dsn,
      undef, undef,
      {
        pg_enable_utf8 => 1,
        RaiseError     => 1,
        AutoCommit     => 1,
      }
    )
  }
);
```


- returns a DBI database handle connected to the Test::PostgreSQL instance.
- this is a shared connection, transactions might cause issues, if in doubt start a new connection

MyApp::Test::PostgreSQL::dbic

```
has dbic => (
  is       => 'ro',
  lazy     => 1,
  default => sub {
    my ($self) = @_;
    MyApp::Schema->connect({
      dbh_maker    => sub { $self->dbh },
      quote_names => 1,
    });
  }
);
```


- returns a DBIC Schema object
- uses the Schema object from your application

Fixtures

- DBIx::Class::EasyFixture

- Data to add to the database for testing
- Written by Curtis "Ovid" Poe
- While stating that it's Alpha software, we use it a lot

The Fixture Package

```
package MyApp::Test::Fixtures;
use Moose;
extends 'DBIx::Class::EasyFixture';
use Crypt::Bcrypt::Easy ();

my %definition_for = (
  basic_user => {
    new => 'User',
    using => {
      user_name => 'tester',
      password => Crypt::Bcrypt::Easy->crypt( text => 'tester', cost => 10 )
    },
    next => [qw(user_entry)],
  },
  user_entry => {
    new => 'Entry',
    using => {
      user_id => \'basic_user',
      entry => 'This is my entry.',
    },
  },
);

sub get_definition {
    my ( $self, $name ) = @_;
    return $definition_for{$name};
}

sub all_fixture_names { return keys %definition_for }
```


- there's a lot of details here and the packages can get quite large
- this example shows the necessities, real world the `%definition_for` hash is huge
- going to break this down and explain what's going on

We'll come back to this

```
package MyApp::Test::Fixtures;
use Moose;
extends 'DBIx::Class::EasyFixture';
use Crypt::Bcrypt::Easy ();

my %definition_for = (
  basic_user => {
    new => 'User',
    using => {
      user_name => 'tester',
      password => Crypt::Bcrypt::Easy->crypt( text => 'tester', cost => 10 )
    },
    next => [qw(user_entry)],
  },
  user_entry => {
    new => 'Entry',
    using => {
      user_id => \'basic_user',
      entry => 'This is my entry.',
    },
  },
);

sub get_definition {
    my ( $self, $name ) = @_;
    return $definition_for{$name};
}

sub all_fixture_names { return keys %definition_for }
```

Let's ignore this section for now and return after looking at the basics.

- This is the data definition, but more to come

Required overrides

```
package MyApp::Fixtures;
use Moose;
extends 'DBIx::Class::EasyFixture';
use Crypt::Bcrypt::Easy ();

my %definition_for = (
  basic_user => {
    new => 'User',
    using => {
      user_name => 'tester',
      password => Crypt::Bcrypt::Easy->crypt( text => 'tester', cost => 10 )
    },
    next => [qw(user_entry)],
  },
  user_entry => {
    new => 'Entry',
    using => {
      user_id => \'basic_user',
      entry => 'This is my entry.',
    },
  },
);

sub get_definition {
    my ( $self, $name ) = @_;
    return $definition_for{$name};
}

sub all_fixture_names { return keys %definition_for }
```


- DBIx::Class:EasyFixture requires that `get_definition` and `all_fixture_names` be overridden.
- These are the simplest definitions and can be reused

Data definition

```
...
my %definition_for = (
  basic_user => {
    new => 'User',
    using => {
      user_name => 'tester',
      password => Crypt::Bcrypt::Easy->crypt( text => 'tester', cost => 10 )
    },
    next => [qw(user_entry)],
  },
  user_entry => {
    new => 'Entry',
    using => {
      user_id => \'basic_user',
      entry => 'This is my entry.',
    },
  },
);
...
```


- `basic_user` is a unique identifier
- new is the result set full package name is `MyApp::Schema::Result::User`
- `using` is the record data
- able to mix in function calls like password
- `next` and `requires` can be used to specify fixture dependencies
- `STRINGREF` is a reference to an existing definition and value is pulled from there

MyApp::Test::PostgreSQL::fixtures

```
has fixtures => (
   is       => 'ro',
   lazy     => 1,
   default => sub {
     MyApp::Test::Fixtures->new( { schema => $_[0]->dbic } )
   },
);
```


- Access to the fixtures object
- Fixtures by default are written in one transaction

MyApp::Test::PostgreSQL::perm_fixtures

```
has perm_fixtures => (
   is       => 'ro',
   lazy     => 1,
   default => sub {
     MyApp::Test::Fixtures->new( {
       schema           => $_[0]->dbic,
       no_transactions => 1,
     } )
   },
);
```


- Access to the fixtures object
- Disable transactions so that other sessions can access the data

Actually writing tests

```
use MyApp::Test::PostgreSQL;
use MyApp::Test::Fixtures;

use Test::Most;

my $db = MyApp::Test::PostgreSQL->new;
my $dbic = $db->dbic;

my $dsn = $db->dsn;
$dsn =~ s/^dbi/dbi/i;

diag $dsn;

lives_ok { $db->deploy } 'database set up';
$db->perm_fixtures->load('basic_user');

# Add all your test methods here

$db->perm_fixtures->unload;
done_testing;
```


- At this point you have a populated database
- New connections can access the database via `$dsn`
- `diag $dsn` is useful for connecting with an external client
- full access to the `MyApp::Test::PostgreSQL` methods for access to `dbh`, and `dbic`

Running tests

```
prove -lv -I t/lib t/simple.t
t/simple.t .. # dbi:Pg:dbname=test;host=127.0.0.1;port=15432;user=postgres

Adding registry tables to db:postgresql://postgres@127.0.0.1:15432/test
Deploying changes to db:postgresql://postgres@127.0.0.1:15432/test
  + user_table ... ok
  + entry_table .. ok
ok 1 - database set up
1..1
Pg refused to die gracefully; killing it violently.
Pg really didn't die.. WTF?
ok
All tests successful.
Files=1, Tests=1, 20 wallclock secs ( 0.03 usr  0.02 sys +  2.53 cusr  1.31 csys =  3.89 CPU)
Result: PASS
```


- `t/lib` required as that's where the `MyApp::Test` modules live

WTF?

```
prove -lv -I t/lib t/simple.t
t/simple.t .. # dbi:Pg:dbname=test;host=127.0.0.1;port=15432;user=postgres

Adding registry tables to db:postgresql://postgres@127.0.0.1:15432/test
Deploying changes to db:postgresql://postgres@127.0.0.1:15432/test
  + user_table ... ok
  + entry_table .. ok
ok 1 - database set up
1..1
Pg refused to die gracefully; killing it violently.
Pg really didn't die.. WTF?
ok
All tests successful.
Files=1, Tests=1, 20 wallclock secs ( 0.03 usr  0.02 sys +  2.53 cusr  1.31 csys =  3.89 CPU)
Result: PASS
```


- Yeah, these error messages are interesting
- non-impacting, might have something to do with fixtures

More Reading

- DBIx::Class::EasyFixture::Tutorial - It's complicated
- Test::Class::Moose - Serious testing for serious Perl
- Test::mongod - run a temporary instance of MongoDB by Jesse Shy

- Written by Curtis "Ovid" Poe
- Originally written by Curtis "Ovid" Poe, and now maintained by Dave Rolsky
- Haven't used Test::mongod though it's on my list of modules to try and tests to migrate, written by co-worker Jesse Shy

References

- Test::PostgreSQL
- Reply
- App::Sqitch
- DBIx::Class::EasyFixture

More

Testing with Postgresql // Toronto Perl Mongers // Shawn Sorichetti

Updating DBIC Schemas

```perl
#!/bin/env perl

use v5.20;
use autodie ':all';

use App::Sqitch;
use App::Sqitch::Command::deploy;
use App::Sqitch::Command::verify;
use Test::PostgreSQL;

my $db = Test::PostgreSQL->new;

my $sqitch = App::Sqitch->new(
  options => {
    engine => 'pg',
  },
);

my $deploy = App::Sqitch::Command::deploy->new(
  sqitch => $sqitch,
  target => $db->uri,
);

$deploy->execute();

say $db->dsn;
my $dsn = $db->dsn;

$dsn =~ s/^dbi/dbi/i;
$dsn =~ s/dbname\=/database=/i;

system "dbicdump -o dump_directory=./lib MyApp::Schema '$dsn'";
```

sqitch.t

```
use Test::PostgreSQL;
use Test::Most;

use App::Sqitch;
use App::Sqitch::Target;
use App::Sqitch::Command::deploy;
use App::Sqitch::Command::verify;
use App::Sqitch::Command::revert;

bail_on_fail;

my $db = Test::PostgreSQL->new;

my $sqitch = App::Sqitch->new(
  options => {
    engine => 'pg',
  },
);

my $target = App::Sqitch::Target->new( sqitch => $sqitch);
my $plan   = $target->plan;

my ( @revert );
while ( my $change = $plan->next ) {
  my $deploy = App::Sqitch::Command::deploy->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $deploy->execute() } 'deploy sqitch plan to ' . $change->name;

  my $verify = App::Sqitch::Command::verify->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change->name,
  );

  lives_ok { $verify->execute() } 'verify sqitch plan';

  push @revert, $change->name;
}

pop @revert;
while ( my $change_name = pop @revert ) {
  my $verify = App::Sqitch::Command::revert->new(
    sqitch => $sqitch,
    target => $db->uri,
    to_change => $change_name,
    no_prompt => 1,
  );

  lives_ok { $verify->execute() } 'revert sqitch plan to ' . $change_name;
}

done_testing();
```

Testing with Postgresql // Toronto Perl Mongers // Shawn Sorichetti