Experiences with DB Unit Testing

via DBI and DBIx::Class

Who am I?

- Ben Kolera: Dev team lead at iseek Communications. Not a perl expert, but know my way around well enough.
- iseek: Builds data centers. Glues together networks and wholesale telco products with perl for fun and profit.

Context of this talk

- Many apps we write talk to a DB via DBI or DBIC
- Unit tests are awesome.
- Often find ourselves wanting to mock / setup a DB for tests.
- Especially so for fat-modeled code.
- Maintainablility is paramount!

Goals of this talk

- To talk about existing solutions and experience at iseek.
- To encourage discussion about existing solutions we've missed.
- To discuss about how we can improve on what we have.
- Warning: this is going to be highly anecdotal and hand wavy, sorry.

Existing Solutions

- Avoiding DBI entirely by mocking away that layer.
- Mocking DBI.
- Setting up a test database just for tests.

Mocking away your DB layer

- Test::MockObject
- Pros: Simple (if you query layer is simple)
- Cons: Doesn't test your queries. Never try to mock DBIC::ResultSets.

Mocking DBI

- DBD::Mock
- Pros: Simple (if your SQL isn't generated)
- Cons: Complicated queries can lead to unmaintainable tests. Never use this for DBIx::Class.

Test DBs

- Two steps:
  - Getting a DB to connect to.
  - Setting up the DB (schema, data)

Test DBs:Creation

- DBD::SQLite
  - Pros: Quick to setup DB.
  - Cons: Doesn’t test target SQL dialect. Limits vendor specific SQL.

Test DBs: Creation

- `Test::{PostgreSQL,MySQL}`
  - Pros: Sets up a real DB in your target SQL server.
  - Cons: Slow (at least with Pg) to create DB.

Test DBs: Setup

- DBIx::Class
  - DBIx::Class::Schema->deploy
  - DBIx::Class::Schema->populate
- Pros: Easy & nice.
- Cons: Requires DBIC

Test DBs: Setup

- DBIx::MultiStatementDo
- Pros: Just works with a DDL string / file.
- Cons: You have to keep a dump of your DB DDL in your module. Could go stale.

Test::DBIx::Class

- Sets up a DB using Test::{Pg,MySql} and lets you populate it via a DBIC populate calls.
- Pros: Has easiness and power of all the good mentioned methods.
- Cons: Configured at compile time which is infuriating. Needs DBIC.

The Missing Pieces?

- No simple interface to give DDL and Data to to easily create a fixture independent of DBIC.
- Test::{Pg,MySQL} are slow. No interface to point to a already created DB to speed up developer tests.

Existing Solutions

- Creating a real DB is the most robust way to test, but is slow.
- Using DBIC is best if you have it, but sql files and MultiStatementDo is great when you aren't using DBIC.

The Goal? DBIC

- Something like:

```
my $fixture = Fixture->new(
    schema   => “MyApp::Schema”,
    backend => “Test::PostgreSQL”,
    data     => [ ... DBIC pop data ]
  OR data    => “t/data/00-test.pl”
);
$fixture->do( func($dbh) { ... } );
```

The Goal? Non DBIC

- Something like:

```
my $fixture = Fixture->new(
    ddl     => “t/sql/schema.ddl”,
    backend => “Test::PostgreSQL”,
    data    => “t/sql/00-test1.sql”
);
$fixture->do( func($dbh) { ... } );
```

Questions?

- Would anyone find that module useful?
- Do people have better ideas?
- Warstories to share?

Resources

- http://github.com/benkolera/bpm-db-testing-talk
- MetaCPAN search:
  - Test::MockObject
  - DBD::Mock
    - DBD::SQLite
    - Test::PostgreSQL
    - Test::MySQL
  - DBIx::Class (Cookbook and ResultSet Docs)
  - Test::DBIx::Class