Unit Test Your Database!

David E. Wheeler

Kineticode, Inc.
PostgreSQL Experts, Inc.

JPUG PGCon 2009

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code
- Users test the code
- App tests are sufficient
- For public interface only
- Prove nothing
- For stable code
- I really like Detroit

Test-Driven Development

- Say you need a Fibonacci Calculator
- Start with a test
- Write the simplest possible function
- Add more tests
- Update the function
- Wash, rinse, repeat...

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])----------

Simple Function

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN 0;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])----------

Run the Test

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
1..2
ok
All tests successful.
Files=1, Tests=2, 0 secs (0.03 usr + 0.00 sys = 0.03 CPU)
Result: PASS
%
```

That was easy

Add Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
```

--:-- try.sql All (SQL[ansi])----------

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
not ok 4 - fib(1) should be 1
# Failed test 4: "fib(1) should be 1"
#         have: 0
#         want: 1
1..4
# Looks like you failed 1 test of 4
Failed 1/4 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 4 Failed: 1)
  Failed test:  4
Files=1, Tests=4, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
% 
```

Modify for the Test

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN 0;fib_for;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])----------

Bare minimum

Tests Pass!

Terminal

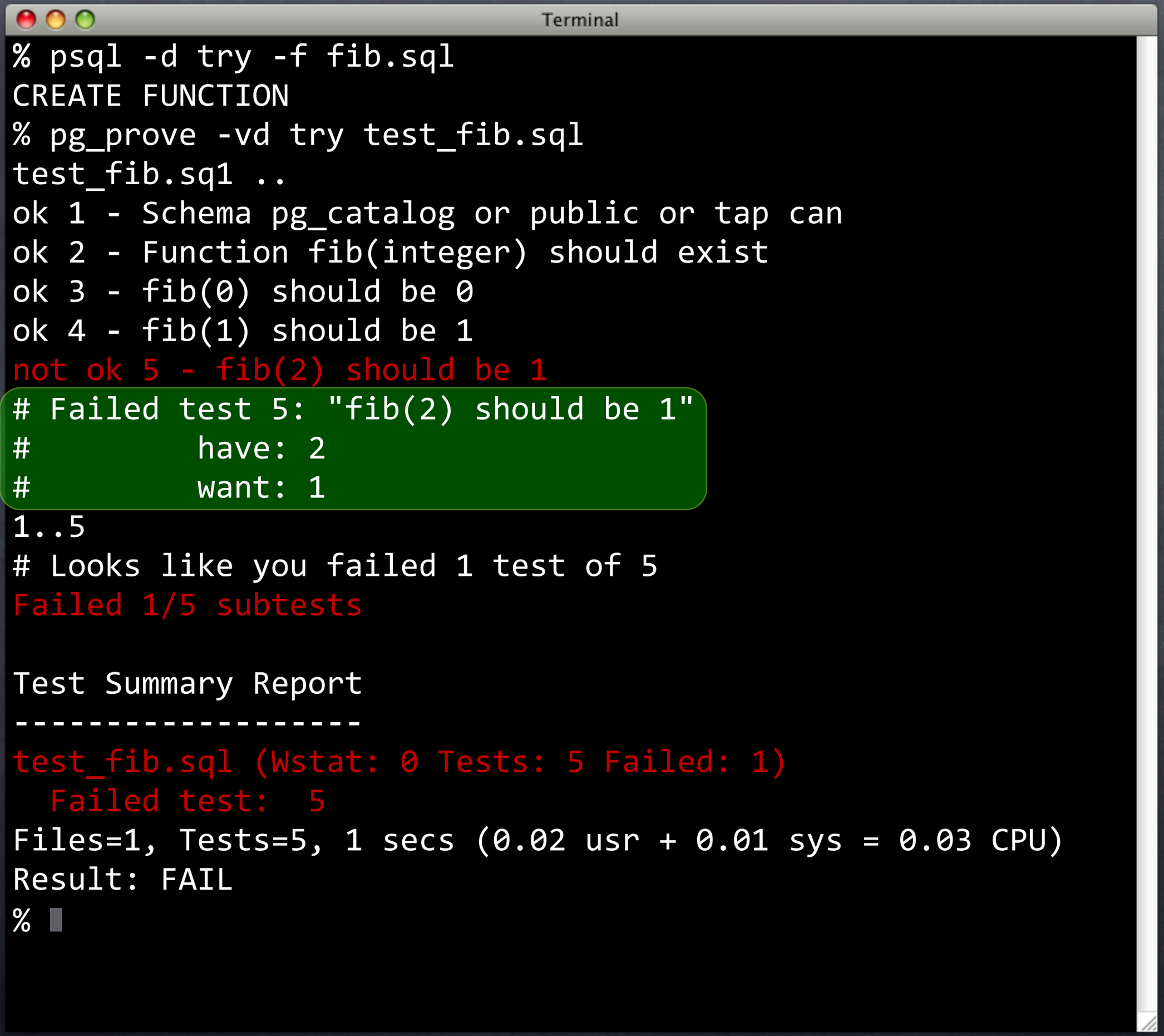
```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
ok 4 - fib(1) should be 1
1..4
ok
All tests successful.
Files=1, Tests=4, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
% 
```

Add Another Assertion

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
```

--:-- try.sql All (SQL[ansi])----------


Terminal
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sq1 ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
ok 4 - fib(1) should be 1
not ok 5 - fib(2) should be 1
Failed test 5: "fib(2) should be 1"
have: 2
want: 1
1..5
Looks like you failed 1 test of 5
Failed 1/5 subtests
Test Summary Report
----
test_fib.sql (Wstat: 0 Tests: 5 Failed: 1)
  Failed test: 5
Files=1, Tests=5, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%

Modify to Pass

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    IF fib_for < 2 THEN
        RETURN fib_for;
    END IF;
    RETURN fib(fib_for - 1) + fib(fib_for - 2);
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])----------

And...Pass!

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
ok 4 - fib(1) should be 1
ok 5 - fib(2) should be 1
1..5
ok
All tests successful.
Files=1, Tests=5, 0 secs (0.02 usr + 0.00 sys = 0.02 CPU)
Result: PASS
% 
```

Still More Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
```

--:-- try.sql All (SQL[ansi])----------

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 .. 1/?
not ok 8 - fib(5) should be 5
# Failed test 8: "fib(5) should be 5"
#         have: 4
#         want: 5
# Looks like you failed 1 test of 8
test_fib.sql .. Failed 1/8 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 8 Failed: 1)
  Failed test:  8
Files=1, Tests=8, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
% 
```

Fix The Function

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    IF fib_for < 2 THEN
        RETURN fib_for;
    END IF;
    RETURN fib(fib_for - 1)
END;     + fib(fib_for - 2);
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])----------

WOOT!

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. ok
All tests successful.
Files=1, Tests=8, 0 secs (0.02 usr + 0.00 sys = 0.02 CPU)
Result: PASS
%
```

A Few More Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
```

--:-- try.sql All (SQL[ansi])----------

We're Golden!

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. ok
All tests successful.
Files=1, Tests=11, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

Add a Regression Test

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
SELECT performs_ok( 'SELECT fib(30)', 500 );
```

--:-- try.sql All (SQL[ansi])----------

What've We Got?

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. 12/?
not ok 12 - Should run in less than 500 ms
# Failed test 12: "Should run in less than 500 ms"
#       runtime: 8418.816 ms
#       exceeds: 500 ms
# Looks like you failed 1 test of 12
test_fib.sql .. Failed 1/12 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 12 Failed: 1)
  Failed test:  12
Files=1, Tests=12, 8 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
% 
```

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
DECLARE
    ret integer := 0;
    nxt integer := 1;
    tmp integer;
BEGIN
    FOR num IN 0..fib_for LOOP
        tmp := ret;
        ret := nxt;
        nxt := tmp + nxt;
    END LOOP;

    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])----------

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 .. 1/?
not ok 3 - fib(0) should be 0
# Failed test 3: "fib(0) should be 0"
#         have: 1
#         want: 0
not ok 5 - fib(2) should be 1
# Failed test 5: "fib(2) should be 1"
#         have: 2
#         want: 1
not ok 6 - fib(3) should be 2
# Failed test 6: "fib(3) should be 2"
#         have: 3
#         want: 2
not ok 7 - fib(4) should be 3
# Failed test 7: "fib(4) should be 3"
#         have: 5
#         want: 3
not ok 8 - fib(5) should be 5
# Failed test 8: "fib(5) should be 5"
#         have: 8
#         want: 5
not ok 9 - fib(6) Should be 8
# Failed test 9: "fib(6) Should be 8"
#         have: 13
#         want: 8
not ok 10 - fib(7) Should be 13
# Failed test 10: "fib(7) Should be 13"
#         have: 21
#         want: 13
not ok 11 - fib(8) Should be 21
# Failed test 11: "fib(8) Should be 21"
#         have: 34
#         want: 21
# Looks like you failed 8 tests of 12
test_fib.sql .. Failed 8/12 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 12 Failed: 8)
  Failed tests:  3, 5-11
Files=1, Tests=12,  0 secs (0.03 usr + 0.01 sys = 0.04 CPU)
Result: FAIL
% 
```

WTF?

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
DECLARE
    ret integer := 0;
    nxt integer := 1;
    tmp integer;
BEGIN
    FOR num IN 0..fib_for LOOP
        tmp := ret;
        ret := nxt;
        nxt := tmp + nxt;
    END LOOP;

    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])----------

Back in Business

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. ok
All tests successful.
Files=1, Tests=12, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

What about Maintenance?

Emacs

```
CREATE FUNCTION find_by_birthday(integer, integer, integer, integer, text)
RETURNS SETOF integer AS $$
DECLARE
    p_day    ALIAS FOR $1;
    p_mon    ALIAS FOR $2;
    p_offset ALIAS FOR $3;
    p_limit  ALIAS FOR $4;
    p_state  ALIAS FOR $5;
    v_qry    TEXT;
    v_output RECORD;
BEGIN
    v_qry := 'SELECT * FROM users WHERE state = ''' || p_state || '''';
    v_qry := v_qry || ' AND birth_mon ~ ''^0?' || p_mon::text || '$''';
    v_qry := v_qry || ' AND birth_day = ''' || p_day::text || '''';
    v_qry := v_qry || ' ORDER BY user_id';
    IF p_offset IS NOT NULL THEN
        v_qry := v_qry || ' OFFSET ' || p_offset;
    END IF;
    IF p_limit IS NOT NULL THEN
        v_qry := v_qry || ' LIMIT ' || p_limit;
    END IF;
    FOR v_output IN EXECUTE v_qry LOOP
        RETURN NEXT v_output.user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])----------

The Situation

- This is production code
- Cannot afford downtime
- No room for mistakes
- Bugs must remain consistent

Test the existing implementation.

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_is_pk(    'users', 'user_id' );
SELECT col_not_null( 'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
--SELECT plan(15);
SELECT * FROM no_plan();

SELECT can('{find_by_birthday}');
SELECT can_ok(
    'find_by_birthday',
    ARRAY['integer', 'integer', 'integer', 'integer', 'text']
);

-- Set up fixtures.
ALTER SEQUENCE users_user_id_seq RESTART 1;
INSERT INTO users (name, birthdate, birth_mon, birth_day, birth_year)
VALUES ('David', '1968-12-19', '12', '19', '1968'),
       ('Josh',  '1970-03-12', '03', '12', '1970'),
       ('Dan',   '1972-06-03', '6',  '3',  '1972'),
       ('Anna',  '2005-06-03', '06', '3',  '2005');

SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])----------

Emacs

```
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])----------

Let's Go with SQL

Emacs

```
CREATE OR REPLACE FUNCTION find_by_birthday(
    p_day    integer,
    p_mon    integer,
    p_offset integer,
    p_limit  integer,
    p_state  text
) RETURNS SETOF integer AS $$
    SELECT user_id
      FROM users
     WHERE state = COALESCE($5, 'active')
       AND EXTRACT(day   FROM birthdate) = $1
       AND EXTRACT(month FROM birthdate) = $2
     ORDER BY user_id
    OFFSET COALESCE( $3, NULL )
     LIMIT COALESCE( $4, NULL )
$$ LANGUAGE sql;
```

--:-- try.sql All (SQL[ansi])----------

Tests are for Finding Bugs

- TDD not for finding bugs
- TDD for sanity and consistency
- Tests prevent future bugs

Tests are Hard

- Good frameworks easy
- pgTAP provides lots of assertions
- If you mean Hard to test interface:
  - Red flag
  - Think about refactoring
  - If it's hard to test...
  - It's hard to use

Never Find Relevant Bugs

- Tests don't find bugs
- Test **PREVENT** bugs
- If your code doesn't work...
- That failure is **RELEVANT**, no?

Time-Consuming

- Good frameworks easy to use
- Iterating between tests and code is natural
- Tests are as fast as your code
- Not as time-consuming as bug hunting
- When no tests, bugs repeat themselves
- And are harder to track down
- Talk about a time sink!

Running Tests is Slow

- Test what you're working on
- Set up automated testing for everything else
- Pay attention to automated test failures

For Inexperienced Developers

- I've been programming for 10 years
- I have no idea what I was thinking a year ago
- Tests make maintenance a breeze
- They give me the confidence to make changes without fearing the consequences
- Tests represent **FREEDOM** from the tyranny of fragility and inconsistency

Unnecessary for Simple Code

- I copied fib() from a Perl library
- It was dead simple
- And it was still wrong
- Tests keep even the simplest code working

Best for Fragile Code

- All code is fragile
- Tests make code ROBUST
- Add regression tests for bugs found by:
  - Integration tests
  - QA department
  - Your users

Users Test our Code

- Talk about fragility
- Staging servers never work
- QA departments are disappearing
- Users don't want to see bugs
- Find ways to test your code
- Users avoid fragile applications

It's a Private Function

- It still needs to work
- It still needs to always work
- Don't reject glass box testing
- Make sure that ALL interfaces work

Application Tests are Sufficient

- App tests should connect as as app user
- May well be security limitations for the app
  - Access only to functions
- Apps cannot adequately test the database
- Database tests should test the database
- Application tests should test the application

Tests Prove Nothing

- This is not a math equation
- This is about:
  - consistency
  - stability
  - robusticity
- If a test fails, it has proved a failure
- Think Karl Popper

Tests are for Stable Code

- How does it become stable?
- Tests the fastest route
- Ensure greater stability over time
- TDD help with working through issues
- TDD helps thinking through interfaces
- Tests encourage experimentation

What're You Waiting For?

- pgTAP: http://pgtap.projects.postgresql.org
- pgUnit: http://en.dklab.ru/lib/dklab_pgunit
- EpicTest: http://www.epictest.org
- pg_regress