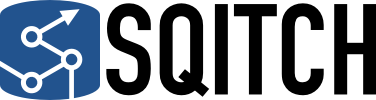
[](http://sqitch.org/)

 [Sobre](http://sqitch.org/about)

 [Docs](http://sqitch.org/docs/)

 [Comunidade](http://sqitch.org/community/)

 [Download](http://sqitch.org/download/)

**sqitchtutorial**

Nome

sqitchtutorial - Um tutorial de introdução ao gerenciamento de mudanças Sqitch no PostgreSQL

Sinopse

sqitch \*

Descrição

Este tutorial explica como criar um projeto PostgreSQL habilitado para sqitch, usar um VCS para planejamento de implantação e trabalhar com outros desenvolvedores para garantir que as alterações permaneçam sincronizadas e na ordem adequada.

Começaremos criando um novo projeto do zero, um site fictício de rede anti-social chamado Flipr. Todos os exemplos usam [Git](https://git-scm.com/) como VCS e [PostgreSQL](https://www.postgresql.org/) como mecanismo de armazenamento, mas na maioria das vezes você pode substituir outros VCSes e mecanismos de banco de dados nos exemplos conforme apropriado.

Se quiser gerenciar um banco de dados SQLite, consulte [sqitchtutorial-sqlite](http://sqitch.org/docs/manual/sqitchtutorial-sqlite) .

Se você gostaria de gerenciar um banco de dados Oracle, consulte [sqitchtutorial-oracle](http://sqitch.org/docs/manual/sqitchtutorial-oracle) .

Se desejar gerenciar um banco de dados MySQL, consulte [sqitchtutorial-mysql](http://sqitch.org/docs/manual/sqitchtutorial-mysql) .

Se você gostaria de gerenciar um banco de dados Firebird, consulte [sqitchtutorial-firebird](http://sqitch.org/docs/manual/sqitchtutorial-firebird) .

Se quiser gerenciar um banco de dados Vertica, consulte [sqitchtutorial-vertica](http://sqitch.org/docs/manual/sqitchtutorial-vertica) .

Se desejar gerenciar um banco de dados Exasol, consulte [sqitchtutorial-exasol](http://sqitch.org/docs/manual/sqitchtutorial-exasol) .

Se desejar gerenciar um banco de dados do Snowflake, consulte [sqitchtutorial-snowflake](http://sqitch.org/docs/manual/sqitchtutorial-snowflake) .

Iniciando um Novo Projeto

Normalmente, a primeira coisa a fazer ao iniciar um novo projeto é criar um repositório de código-fonte. Então, vamos fazer isso com o Git:

> mkdir flipr

> cd flipr

> git init .

Initialized empty Git repository in /flipr/.git/

> touch README.md

> git add .

> git commit -am 'Initialize project, add README.'

Se você é um usuário Git e deseja acompanhar o histórico, o repositório usado nesses exemplos está [no GitHub](https://github.com/sqitchers/sqitch-intro) .

Agora que temos um repositório, vamos começar com o Sqitch. Cada projeto Sqitch deve ter um nome associado a ele e, opcionalmente, um URI exclusivo. Recomendamos incluir o URI, pois aumenta a exclusividade dos identificadores de objeto internamente e evita a implantação de um projeto diferente com o mesmo nome. Então, vamos especificar um quando inicializarmos o Sqitch:

> sqitch init flipr --uri https://github.com/sqitchers/sqitch-intro/ --engine pg

Created sqitch.conf

Created sqitch.plan

Created deploy/

Created revert/

Created verify/

Vamos dar uma olhada em sqitch.conf:

> cat sqitch.conf

[core]

engine = pg

# plan\_file = sqitch.plan

# top\_dir = .

# [engine "pg"]

# target = db:pg:

# registry = sqitch

# client = psql

Bom, ele percebeu que estamos criando mudanças para o motor PostgreSQL, graças à --engine pgopção, e salvou no arquivo. Além disso, ele escreveu uma [engine "pg"]seção comentada com todas as configurações específicas do mecanismo PostgreSQL disponíveis comentadas e prontas para serem editadas conforme apropriado.

Por padrão, o Sqitch lerá sqitch.confno diretório atual as configurações. Mas também lerá ~/.sqitch/sqitch.confas configurações específicas do usuário. Como o psqlcliente PostgreSQL não está no caminho do meu sistema, vamos prosseguir e dizer onde encontrar o cliente em nosso computador (não se preocupe se você estiver usando a [imagem Docker](https://hub.docker.com/r/sqitch/sqitch/) porque ela usa o cliente dentro do contêiner, não no sua máquina host):

> sqitch config --user engine.pg.client /opt/local/pgsql/bin/psql

E digamos também quem somos, já que esses dados serão usados ​​em todos os nossos projetos:

> sqitch config --user user.name 'Marge N. O’Vera'

> sqitch config --user user.email 'marge@example.com'

Dê uma olhada em ~/.sqitch/sqitch.confe você verá o seguinte:

> cat ~/.sqitch/sqitch.conf

[engine "pg"]

client = /opt/local/pgsql/bin/psql

[user]

name = Marge N. O’Vera

email = marge@example.com

O que significa que o Sqitch deve ser capaz de encontrar psqlpara qualquer projeto e que sempre nos identificará adequadamente ao planejar e comprometer mudanças.

De volta ao repositório. Dê uma olhada no arquivo do plano sqitch.plan:

> cat sqitch.plan

%syntax-version=1.0.0

%project=flipr

%uri=https://github.com/sqitchers/sqitch-intro/

Observe que ele pegou o nome e o URI do aplicativo que estamos construindo. O Sqitch usa esses dados para gerenciar dependências entre projetos. O %syntax-versionpragma é sempre definido pelo Sqitch, para que ele sempre saiba como analisar o plano, mesmo se o formato mudar no futuro.

Vamos confirmar essas mudanças e começar a criar as mudanças do banco de dados.

> git add .

> git commit -am 'Initialize Sqitch configuration.'

[master 85e8d7c] Initialize Sqitch configuration.

2 files changed, 19 insertions(+)

create mode 100644 sqitch.conf

create mode 100644 sqitch.plan

Nossa Primeira Mudança

Primeiro, nosso projeto precisará de um esquema. Isso cria um bom namespace para todos os objetos que farão parte do aplicativo flipr. Execute este comando:

> sqitch add appschema -n 'Add schema for all flipr objects.'

Created deploy/appschema.sql

Created revert/appschema.sql

Created verify/appschema.sql

Added "appschema" to sqitch.plan

O [add](http://sqitch.org/docs/manual/sqitch-add)comando adiciona uma mudança de banco de dados ao plano e grava scripts de implementação, reversão e verificação que representam a mudança. Agora vamos editar esses arquivos. O deploytrabalho do script é criar o esquema. Então, adicionamos isso a deploy/appschema.sql:

CREATE SCHEMA flipr;

A reverttarefa do script é reverter precisamente a alteração no script de implantação, então adicionamos isso a revert/appschema.sql:

DROP SCHEMA flipr;

Agora podemos tentar implantar essa mudança. Primeiro, precisamos criar um banco de dados para implantar em:

> createdb flipr\_test

Agora diremos ao Sqitch para onde enviar a alteração por meio de um [URI de banco de dados](https://github.com/libwww-perl/uri-db/) :

> sqitch deploy db:pg:flipr\_test

Adding registry tables to db:pg:flipr\_test

Deploying to db:pg:flipr\_test

+ appschema .. ok

O primeiro Sqitch criou tabelas de registro usadas para rastrear alterações no banco de dados. A estrutura e o nome do registro variam entre os bancos de dados (PostgreSQL usa um esquema para definir o namespace de seu registro, enquanto o SQLite e o MySQL usam bancos de dados separados). Em seguida, Sqitch implanta mudanças. Só temos um até agora; o + reforça a ideia de que a mudança está sendo addedà base de dados.

Com essa mudança implantada, se você se conectar ao banco de dados, poderá ver o esquema:

> psql -d flipr\_test -c '\dn flipr'

List of schemas

Name | Owner

-------+-------

flipr | marge

Confie mas verifique

Mas isso dá muito trabalho. Você realmente deseja fazer algo assim após cada implantação?

É aqui que o verifyscript entra. Seu trabalho é testar se o deploy fez o que deveria. Isso deve ser feito sem levar em consideração quaisquer dados que possam estar no banco de dados e deve gerar um erro se a implementação não for bem-sucedida. No PostgreSQL, a maneira mais simples de fazer isso para objetos não consultáveis, como esquemas, é aproveitar as [funções de consulta de privilégio de acesso](https://www.postgresql.org/docs/current/static/functions-info.html#FUNCTIONS-INFO-ACCESS-TABLE) . Essas funções lançam exceções convenientemente se o objeto que está sendo consultado não existir. Para nosso novo esquema, has\_schema\_privilege()funcionará muito bem. Coloque esta consulta em verify/appschema.sql:

SELECT pg\_catalog.has\_schema\_privilege('flipr', 'usage');

**Importante!**Esta consulta não está verificando se o usuário tem usageprivilégios no esquema flipr. A verificação será aprovada mesmo se o usuário atual não tiver direitos de uso.

**Importante!**Ambos SELECT false;e SELECT true;consultas vai passar com êxito verifyetapa. Apenas as consultas que geram uma exceção falharão.

Essa funcionalidade pode não estar disponível para outros bancos de dados, mas você pode usar *qualquer* consulta que lançará uma exceção se o esquema não existir. Uma maneira prática de fazer isso é dividir por zero se um objeto não existir. Portanto, para outros bancos de dados, supondo que a divisão por zero seja fatal, você poderia fazer algo assim:

SELECT 1/COUNT(\*) FROM information\_schema.schemata WHERE schema\_name = 'flipr';

No Postgres 9.5+, você pode usar PL/pgSQLfunções anônimas com instruções ASSERT/ RAISE.

DO $$

BEGIN

ASSERT (SELECT has\_schema\_privilege('flipr', 'usage'));

END $$;

Você pode usar variáveis ​​para realizar verificações mais complexas:

DO $$

DECLARE

result varchar;

BEGIN

result := (SELECT name FROM flipr.pipelines WHERE id = 1);

ASSERT result = 'Example';

END $$;

Este exemplo garante que o registro id=1na pipelinestabela tenha namecampos iguais 'Example'.

De qualquer maneira, execute o verifyscript com o [verify](http://sqitch.org/docs/manual/sqitch-verify) comando:

> sqitch verify db:pg:flipr\_test

Verifying db:pg:flipr\_test

\* appschema .. ok

Verify successful

Parece bom! Se você quiser ter certeza de que o script de verificação morre corretamente se o esquema não existir, altere temporariamente o nome do esquema no script para algo que não existe, algo como:

SELECT pg\_catalog.has\_schema\_privilege('nonesuch', 'usage');

Então, [verify](http://sqitch.org/docs/manual/sqitch-verify)novamente:

> sqitch verify db:pg:flipr\_test

Verifying db:pg:flipr\_test

\* appschema .. psql:verify/appschema.sql:5: ERROR: schema "nonesuch" does not exist

# Verify script "verify/appschema.sql" failed.

not ok

Verify Summary Report

---------------------

Changes: 1

Errors: 1

Verify failed

É até bom o suficiente para nos dizer qual é o problema. Ou, para o exemplo de divisão por zero, altere o nome do esquema:

SELECT 1/COUNT(\*) FROM information\_schema.schemata WHERE schema\_name = 'nonesuch';

Em seguida, a verificação será semelhante a:

> sqitch verify db:pg:flipr\_test

Verifying db:pg:flipr\_test

\* appschema .. psql:verify/appschema.sql:5: ERROR: division by zero

# Verify script "verify/appschema.sql" failed.

not ok

Verify Summary Report

---------------------

Changes: 1

Errors: 1

Verify failed

Saída de erro menos útil, mas o suficiente para nos alertar de que algo deu errado.

Não se esqueça de alterar o nome do esquema de volta antes de continuar!

Status, reverter, registrar, repetir

Para fins puramente informativos, sempre podemos ver como uma implantação foi registrada por meio do [status](http://sqitch.org/docs/manual/sqitch-status)comando, que lê as tabelas de registro do banco de dados:

> sqitch status db:pg:flipr\_test

# On database db:pg:flipr\_test

# Project: flipr

# Change: c7981df861183412b01be706889e508a63d445ca

# Name: appschema

# Deployed: 2013-12-30 15:27:15 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Vamos nos certificar de que podemos reverter a mudança:

> sqitch revert db:pg:flipr\_test

Revert all changes from db:pg:flipr\_test? [Yes]

- appschema .. ok

O [revert](http://sqitch.org/docs/manual/sqitch-revert)comando primeiro avisa para ter certeza de que realmente queremos reverter. Isso evita acidentes desnecessários. Você pode passar a -yopção de desabilitar o prompt. Além disso, observe o -antes do nome da alteração na saída, o que reforça que a alteração está sendo *removida* do banco de dados. E agora o esquema deve ter desaparecido:

> psql -d flipr\_test -c '\dn flipr'

List of schemas

Name | Owner

------+-------

E a mensagem de status deve refletir o máximo:

> sqitch status db:pg:flipr\_test

# On database db:pg:flipr\_test

No changes deployed

Claro, como nada é implantado, o [verify](http://sqitch.org/docs/manual/sqitch-verify)comando não tem nada para verificar:

> sqitch verify db:pg:flipr\_test

Verifying db:pg:flipr\_test

No changes deployed

No entanto, ainda temos um registro de que a mudança aconteceu, visível por meio do [log](http://sqitch.org/docs/manual/sqitch-log)comando:

> sqitch log db:pg:flipr\_test

On database db:pg:flipr\_test

Revert c7981df861183412b01be706889e508a63d445ca

Name: appschema

Committer: Marge N. O’Vera <marge@example.com>

Date: 2013-12-30 15:38:17 -0800

Add schema for all flipr objects.

Deploy c7981df861183412b01be706889e508a63d445ca

Name: appschema

Committer: Marge N. O’Vera <marge@example.com>

Date: 2013-12-30 15:27:15 -0800

Add schema for all flipr objects.

Observe que as ações que tomamos são mostradas em ordem cronológica reversa, com a reversão primeiro e depois a implantação.

Legal. Agora vamos cometer.

> git add .

> git commit -m 'Add flipr schema.'

[master d812132] Add flipr schema.

4 files changed, 22 insertions(+)

create mode 100644 deploy/appschema.sql

create mode 100644 revert/appschema.sql

create mode 100644 verify/appschema.sql

E então implante novamente. Desta vez, vamos usar a --verifyopção, para que o verifyscript seja aplicado quando a mudança for implantada:

> sqitch deploy --verify db:pg:flipr\_test

Deploying changes to db:pg:flipr\_test

+ appschema .. ok

E agora o esquema deve estar de volta:

> psql -d flipr\_test -c '\dn flipr'

List of schemas

Name | Owner

-------+-------

flipr | marge

Quando olhamos para o status, a implantação estará lá:

> sqitch status db:pg:flipr\_test

# On database db:pg:flipr\_test

# Project: flipr

# Change: c7981df861183412b01be706889e508a63d445ca

# Name: appschema

# Deployed: 2013-12-30 15:40:53 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

No alvo

Estou ficando um pouco cansado de ter que digitar sempre db:pg:flipr\_test, você não? Este [URI de conexão de banco de dados](https://github.com/libwww-perl/uri-db/) informa ao Sqitch como se conectar ao destino de implantação, mas não precisamos continuar usando o URI. Podemos nomear o alvo:

> sqitch target add flipr\_test db:pg:flipr\_test

O [target](http://sqitch.org/docs/manual/sqitch-target)comando, inspirado em [git-remote](https://git-scm.com/docs/git-remote), permite o gerenciamento de um ou mais destinos de implementação nomeados. Acabamos de adicionar um destino chamado flipr\_test, o que significa que podemos usar a string flipr\_testpara o destino, em vez do URI. Mas como estamos fazendo muitos testes, também podemos usar o [engine](http://sqitch.org/docs/manual/sqitch-engine)comando para dizer ao Sqitch para implantar no flipr\_testdestino por padrão:

> sqitch engine add pg flipr\_test

Agora podemos omitir o argumento de destino completamente, a menos que precisemos implantar em outro banco de dados. O que faremos, eventualmente, mas pelo menos nossos exemplos serão mais simples de agora em diante, por exemplo:

> sqitch status

# On database flipr\_test

# Project: flipr

# Change: c7981df861183412b01be706889e508a63d445ca

# Name: appschema

# Deployed: 2013-12-30 15:40:53 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Isso permite que as coisas sejam um pouco mais concisas. Também vamos nos certificar de que as mudanças sejam verificadas após implantá-las:

> sqitch config --bool deploy.verify true

> sqitch config --bool rebase.verify true

Veremos o [rebase](http://sqitch.org/docs/manual/sqitch-rebase)comando um pouco mais tarde. Enquanto isso, vamos confirmar a nova configuração e fazer mais algumas alterações!

> git commit -am 'Set default deployment target and always verify.'

[master a6267d3] Set default deployment target and always verify.

1 file changed, 8 insertions(+)

Implantar com dependência

Let’s add another change, this time to create a table. Our app will need users, of course, so we’ll create a table for them. First, add the new change:

> sqitch add users --requires appschema -n 'Creates table to track our users.'

Created deploy/users.sql

Created revert/users.sql

Created verify/users.sql

Added "users [appschema]" to sqitch.plan

Note that we’re requiring the appschema change as a dependency of the new users change. Although that change has already been added to the plan and therefore should always be applied before the users change, it’s a good idea to be explicit about dependencies.

Now edit the scripts. When you’re done, deploy/users.sql should look like this:

-- Deploy flipr:users to pg

-- requires: appschema

BEGIN;

SET client\_min\_messages = 'warning';

CREATE TABLE flipr.users (

nickname TEXT PRIMARY KEY,

password TEXT NOT NULL,

timestamp TIMESTAMPTZ NOT NULL DEFAULT NOW()

);

COMMIT;

A few things to notice here. On the second line, the dependence on the appschema change has been listed. This doesn’t do anything, but the default deploy PostgreSQL template lists it here for your reference while editing the file. Useful, right?

Notice that all of the SQL code is wrapped in a transaction. This is handy for PostgreSQL deployments, because PostgreSQL DDLs are transactional. The upshot is that if any part of this deploy script fails, the whole change fails. Such may work less-well for database engines that don’t support transactional DDLs.

The table itself will be created in the flipr schema. This is why we need to require the appschema change.

Now for the verify script. The simplest way to check that the table was created and has the expected columns without touching the data? Just select from the table with a false WHERE clause. Add this to verify/users.sql:

SELECT nickname, password, timestamp

FROM flipr.users

WHERE FALSE;

Now for the revert script: all we have to do is drop the table. Add this to revert/users.sql:

DROP TABLE flipr.users;

Couldn’t be much simpler, right? Let’s deploy this bad boy:

> sqitch deploy

Deploying changes to flipr\_test

+ users .. ok

We know, since verification is enabled, that the table must have been created. But for the purposes of visibility, let’s have a quick look:

> psql -d flipr\_test -c '\d flipr.users'

Table "flipr.users"

Column | Type | Modifiers

-----------+--------------------------+------------------------

nickname | text | not null

password | text | not null

timestamp | timestamp with time zone | not null default now()

Indexes:

"users\_pkey" PRIMARY KEY, btree (nickname)

We can also verify all currently deployed changes with the [verify](http://sqitch.org/docs/manual/sqitch-verify) command:

> sqitch verify

Verifying flipr\_test

\* appschema .. ok

\* users ...... ok

Verify successful

Now have a look at the status:

> sqitch status

# On database flipr\_test

# Project: flipr

# Change: 77398e1dbc5fbce58b05eb67d201f15774718727

# Name: users

# Deployed: 2013-12-30 15:51:09 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Success! Let’s make sure we can revert the change, as well:

> sqitch revert --to @HEAD^ -y

Reverting changes to appschema from flipr\_test

- users .. ok

Note that we’ve used the --to option to specify the change to revert to. And what do we revert to? The symbolic tag @HEAD, when passed to [revert](http://sqitch.org/docs/manual/sqitch-revert), always refers to the last change deployed to the database. (For other commands, it refers to the last change in the plan.) Appending the caret (^) tells Sqitch to select the change *prior* to the last deployed change. So we revert to appschema, the penultimate change. The other potentially useful symbolic tag is @ROOT, which refers to the first change deployed to the database (or in the plan, depending on the command).

Back to the database. The users table should be gone but the flipr schema should still be around:

> psql -d flipr\_test -c '\d flipr.users'

Did not find any relation named "flipr.users".

The [status](http://sqitch.org/docs/manual/sqitch-status) command politely informs us that we have undeployed changes:

> sqitch status

# On database flipr\_test

# Project: flipr

# Change: c7981df861183412b01be706889e508a63d445ca

# Name: appschema

# Deployed: 2013-12-30 15:40:53 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Undeployed change:

\* users

As does the [verify](http://sqitch.org/docs/manual/sqitch-verify) command:

> sqitch verify

Verifying flipr\_test

\* appschema .. ok

Undeployed change:

\* users

Verify successful

Note that the verify is successful, because all currently-deployed changes are verified. The list of undeployed changes (just “users” here) reminds us about the current state.

Okay, let’s commit and deploy again:

> git add .

> git commit -am 'Add users table.'

[master d58ea2f] Add users table.

4 files changed, 31 insertions(+)

create mode 100644 deploy/users.sql

create mode 100644 revert/users.sql

create mode 100644 verify/users.sql

> sqitch deploy

Deploying changes to flipr\_test

+ users .. ok

Looks good. Check the status:

> sqitch status

# On database flipr\_test

# Project: flipr

# Change: 77398e1dbc5fbce58b05eb67d201f15774718727

# Name: users

# Deployed: 2013-12-30 15:57:14 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Excellent. Let’s do some more!

Add Two at Once

Let’s add a couple more changes to add functions for managing users.

> sqitch add insert\_user --requires users --requires appschema \

-n 'Creates a function to insert a user.'

Created deploy/insert\_user.sql

Created revert/insert\_user.sql

Created verify/insert\_user.sql

Added "insert\_user [users appschema]" to sqitch.plan

> sqitch add change\_pass --requires users --requires appschema \

-n 'Creates a function to change a user password.'

Created deploy/change\_pass.sql

Created revert/change\_pass.sql

Created verify/change\_pass.sql

Added "change\_pass [users appschema]" to sqitch.plan

Now might be a good time to have a look at the deployment plan:

> cat sqitch.plan

%syntax-version=1.0.0

%project=flipr

%uri=https://github.com/sqitchers/sqitch-intro/

appschema 2013-12-30T23:19:45Z Marge N. O’Vera <marge@example.com> # Add schema for all flipr objects.

users [appschema] 2013-12-30T23:49:00Z Marge N. O’Vera <marge@example.com> # Creates table to track our users.

insert\_user [users appschema] 2013-12-30T23:57:36Z Marge N. O’Vera <marge@example.com> # Creates a function to insert a user.

change\_pass [users appschema] 2013-12-30T23:57:45Z Marge N. O’Vera <marge@example.com> # Creates a function to change a user password.

Each change appears on a single line with the name of the change, a bracketed list of dependencies, a timestamp, the name and email address of the user who planned the change, and a note.

Let’s write the code for the new changes. Here’s what deploy/insert\_user.sql should look like:

-- Deploy flipr:insert\_user to pg

-- requires: users

-- requires: appschema

BEGIN;

CREATE OR REPLACE FUNCTION flipr.insert\_user(

nickname TEXT,

password TEXT

) RETURNS VOID LANGUAGE SQL SECURITY DEFINER AS $$

INSERT INTO flipr.users VALUES($1, md5($2));

$$;

COMMIT;

Here’s what verify/insert\_user.sql might look like:

BEGIN;

SELECT has\_function\_privilege('flipr.insert\_user(text, text)', 'execute');

ROLLBACK;

We simply take advantage of the fact that has\_function\_privilege() throws an exception if the specified function does not exist.

And revert/insert\_user.sql should look something like this:

-- Revert flipr:insert\_user from pg

BEGIN;

DROP FUNCTION flipr.insert\_user(TEXT, TEXT);

COMMIT;

Now for change\_pass; deploy/change\_pass.sql might look like this:

-- Deploy flipr:change\_pass to pg

-- requires: users

-- requires: appschema

BEGIN;

CREATE OR REPLACE FUNCTION flipr.change\_pass(

nick TEXT,

oldpass TEXT,

newpass TEXT

) RETURNS BOOLEAN LANGUAGE plpgsql SECURITY DEFINER AS $$

BEGIN

UPDATE flipr.users

SET password = md5($3)

WHERE nickname = $1

AND password = md5($2);

RETURN FOUND;

END;

$$;

COMMIT;

Use has\_function\_privilege() in verify/change\_pass.sql again:

BEGIN;

SELECT has\_function\_privilege('flipr.change\_pass(text, text, text)', 'execute');

ROLLBACK;

And of course, its revert script, revert/change\_pass.sql, should look something like:

-- Revert flipr:change\_pass from pg

BEGIN;

DROP FUNCTION flipr.change\_pass(TEXT, TEXT, TEXT);

COMMIT;

Try em out!

> sqitch deploy

Deploying changes to flipr\_test

+ insert\_user .. ok

+ change\_pass .. ok

Do we have the functions? Of course we do, they were verified. Still, have a look:

> psql -d flipr\_test -c '\df flipr.\*'

List of functions

Schema | Name | Result data type | Argument data types | Type

--------+-------------+------------------+---------------------------------------+--------

flipr | change\_pass | boolean | nick text, oldpass text, newpass text | normal

flipr | insert\_user | void | nickname text, password text | normal

And what’s the status?

> sqitch status

# On database flipr\_test

# Project: flipr

# Change: 01a4f6964b89284525cb5877d222df8be70d1647

# Name: change\_pass

# Deployed: 2013-12-30 15:59:44 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Looks good. Let’s make sure revert works:

> sqitch revert -y --to @HEAD^^

Reverting changes to users from flipr\_test

- change\_pass .. ok

- insert\_user .. ok

> psql -d flipr\_test -c '\df flipr.\*'

List of functions

Schema | Name | Result data type | Argument data types | Type

--------+------+------------------+---------------------+------

Note the use of @HEAD^^ to specify that the revert be to two changes prior the last deployed change. Looks good. Let’s do the commit and re-deploy dance:

> git add .

> git commit -m 'Add `insert\_user()` and `change\_pass()`.'

[master c9b4d68] Add `insert\_user()` and `change\_pass()`.

7 files changed, 65 insertions(+)

create mode 100644 deploy/change\_pass.sql

create mode 100644 deploy/insert\_user.sql

create mode 100644 revert/change\_pass.sql

create mode 100644 revert/insert\_user.sql

create mode 100644 verify/change\_pass.sql

create mode 100644 verify/insert\_user.sql

> sqitch deploy

Deploying changes to flipr\_test

+ insert\_user .. ok

+ change\_pass .. ok

> sqitch status

# On database flipr\_test

# Project: flipr

# Change: 01a4f6964b89284525cb5877d222df8be70d1647

# Name: change\_pass

# Deployed: 2013-12-30 16:00:50 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

> sqitch verify

Verifying flipr\_test

\* appschema .... ok

\* users ........ ok

\* insert\_user .. ok

\* change\_pass .. ok

Verify successful

Great, we’re fully up-to-date!

Ship It!

Let’s do a first release of our app. Let’s call it 1.0.0-dev1 Since we want to have it go out with deployments tied to the release, let’s tag it:

> sqitch tag v1.0.0-dev1 -n 'Tag v1.0.0-dev1.'

Tagged "change\_pass" with @v1.0.0-dev1

> git commit -am 'Tag the database with v1.0.0-dev1.'

[master 0acef3e] Tag the database with v1.0.0-dev1.

1 file changed, 1 insertion(+)

> git tag v1.0.0-dev1 -am 'Tag v1.0.0-dev1'

We can try deploying to make sure the tag gets picked up like so:

> createdb flipr\_dev

> sqitch deploy db:pg:flipr\_dev

Adding registry tables to db:pg:flipr\_dev

Deploying changes to db:pg:flipr\_dev

+ appschema ................. ok

+ users ..................... ok

+ insert\_user ............... ok

+ change\_pass @v1.0.0-dev1 .. ok

Great, all four changes were deployed and change\_pass was tagged with @v1.0.0-dev1. Let’s have a look at the status:

> sqitch status db:pg:flipr\_dev

# On database db:pg:flipr\_dev

# Project: flipr

# Change: 01a4f6964b89284525cb5877d222df8be70d1647

# Name: change\_pass

# Tag: @v1.0.0-dev1

# Deployed: 2013-12-30 16:02:19 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Note the listing of the tag as part of the status message. Now let’s bundle everything up for release:

> sqitch bundle

Bundling into bundle/

Writing config

Writing plan

Writing scripts

+ appschema

+ users

+ insert\_user

+ change\_pass @v1.0.0-dev1

Now we can package the bundle directory and distribute it. When it gets installed somewhere, users can use Sqitch to deploy to the database. Let’s try deploying it:

> cd bundle

> createdb flipr\_prod

> sqitch deploy db:pg:flipr\_prod

Adding registry tables to db:pg:flipr\_prod

Deploying changes to db:pg:flipr\_prod

+ appschema ................. ok

+ users ..................... ok

+ insert\_user ............... ok

+ change\_pass @v1.0.0-dev1 .. ok

Looks much the same as before, eh? Package it up and ship it!

> cd ..

> mv bundle flipr-v1.0.0-dev1

> tar -czf flipr-v1.0.0-dev1.tgz flipr-v1.0.0-dev1

Flip Out

Now that we’ve got the basics of user management done, let’s get to work on the core of our product, the “flip.” Since other folks are working on other tasks in the repository, we’ll work on a branch, so we can all stay out of each other’s way. So let’s branch:

> git checkout -b flips

Switched to a new branch 'flips'

Now we can add a new change to create a table for our flips.

> sqitch add flips -r appschema -r users -n 'Adds table for storing flips.'

Created deploy/flips.sql

Created revert/flips.sql

Created verify/flips.sql

Added "flips [appschema users]" to sqitch.plan

You know the drill by now. Edit deploy/flips.sql:

-- Deploy flipr:flips to pg

-- requires: appschema

-- requires: users

BEGIN;

SET client\_min\_messages = 'warning';

CREATE TABLE flipr.flips (

id BIGSERIAL PRIMARY KEY,

nickname TEXT NOT NULL REFERENCES flipr.users(nickname),

body TEXT NOT NULL DEFAULT '' CHECK ( length(body) <= 180 ),

timestamp TIMESTAMPTZ NOT NULL DEFAULT clock\_timestamp()

);

COMMIT;

Edit verify/flips.sql:

-- Verify flipr:flips on pg

BEGIN;

SELECT id

, nickname

, body

, timestamp

FROM flipr.flips

WHERE FALSE;

ROLLBACK;

And edit revert/flips.sql:

-- Revert flipr:flips from pg

BEGIN;

DROP TABLE flipr.flips;

COMMIT;

And give it a whirl:

> sqitch deploy

Deploying changes to flipr\_test

+ flips .. ok

Look good?

> sqitch status --show-tags

# On database flipr\_test

# Project: flipr

# Change: 4d164ef5986450f00a565735518b1d126f8ee69d

# Name: flips

# Deployed: 2013-12-30 16:34:38 -0800

# By: Marge N. O’Vera <marge@example.com>

#

# Tag:

# @v1.0.0-dev1 - 2013-12-30 16:34:38 -0800 - Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Note the use of --show-tags to show all the deployed tags. Now make it so:

> git add .

[flips e8f4655] Add flips table.

> git commit -am 'Add flips table.'

4 files changed, 37 insertions(+)

create mode 100644 deploy/flips.sql

create mode 100644 revert/flips.sql

create mode 100644 verify/flips.sql

Wash, Rinse, Repeat

Now comes the time to add functions to manage flips. I’m sure you have things nailed down now. Go ahead and add insert\_flip and delete\_flip changes and commit them. The insert\_flip deploy script might look something like:

-- Deploy flipr:insert\_flip to pg

-- requires: flips

-- requires: appschema

-- requires: users

BEGIN;

CREATE OR REPLACE FUNCTION flipr.insert\_flip(

nickname TEXT,

body TEXT

) RETURNS BIGINT LANGUAGE sql SECURITY DEFINER AS $$

INSERT INTO flipr.flips (nickname, body)

VALUES ($1, $2)

RETURNING id;

$$;

COMMIT;

And the delete\_flip deploy script might look something like:

-- Deploy flipr:delete\_flip to pg

-- requires: flips

-- requires: appschema

-- requires: users

BEGIN;

CREATE OR REPLACE FUNCTION flipr.delete\_flip(

flip\_id BIGINT

) RETURNS BOOLEAN LANGUAGE plpgsql SECURITY DEFINER AS $$

BEGIN

DELETE FROM flipr.flips WHERE id = flip\_id;

RETURN FOUND;

END;

$$;

COMMIT;

The verify scripts are:

-- Verify flipr:insert\_flip on pg

BEGIN;

SELECT has\_function\_privilege('flipr.insert\_flip(text, text)', 'execute');

ROLLBACK;

And:

-- Verify flipr:delete\_flip on pg

BEGIN;

SELECT has\_function\_privilege('flipr.delete\_flip(bigint)', 'execute');

ROLLBACK;

The revert scripts are:

-- Revert flipr:insert\_flip from pg

BEGIN;

DROP FUNCTION flipr.insert\_flip(TEXT, TEXT);

COMMIT;

And:

-- Revert flipr:delete\_flip from pg

BEGIN;

DROP FUNCTION flipr.delete\_flip(BIGINT);

COMMIT;

Check the [example git repository](https://github.com/sqitchers/sqitch-intro) for the complete details. Test [deploy](http://sqitch.org/docs/manual/sqitch-deploy) and [revert](http://sqitch.org/docs/manual/sqitch-revert), then commit it to the repository. The status should end up looking something like this:

> sqitch status --show-tags

# On database flipr\_test

# Project: flipr

# Change: 9a645034b35fa46df37a3725c480982628cc64ec

# Name: delete\_flip

# Deployed: 2013-12-30 16:37:51 -0800

# By: Marge N. O’Vera <marge@example.com>

#

# Tag:

# @v1.0.0-dev1 - 2013-12-30 16:34:38 -0800 - Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

Good, we’ve finished this feature. Time to merge back into master.

Emergency

Let’s do it:

> git checkout master

Switched to branch 'master'

> git pull

Updating 0acef3e..d4cbd7d

Fast-forward

deploy/delete\_list.sql | 20 ++++++++++++++++++++

deploy/insert\_list.sql | 17 +++++++++++++++++

deploy/lists.sql | 16 ++++++++++++++++

revert/delete\_list.sql | 7 +++++++

revert/insert\_list.sql | 7 +++++++

revert/lists.sql | 7 +++++++

sqitch.plan | 4 ++++

verify/delete\_list.sql | 7 +++++++

verify/insert\_list.sql | 7 +++++++

verify/lists.sql | 9 +++++++++

10 files changed, 101 insertions(+)

create mode 100644 deploy/delete\_list.sql

create mode 100644 deploy/insert\_list.sql

create mode 100644 deploy/lists.sql

create mode 100644 revert/delete\_list.sql

create mode 100644 revert/insert\_list.sql

create mode 100644 revert/lists.sql

create mode 100644 verify/delete\_list.sql

create mode 100644 verify/insert\_list.sql

create mode 100644 verify/lists.sql

Hrm, that’s interesting. Looks like someone made some changes to master. They added list support. Well, let’s see what happens when we merge our changes.

> git merge --no-ff flips

Auto-merging sqitch.plan

CONFLICT (content): Merge conflict in sqitch.plan

Automatic merge failed; fix conflicts and then commit the result.

Oh, a conflict in sqitch.plan. Not too surprising, since both the merged lists branch and our flips branch added changes to the plan. Let’s try a different approach.

The truth is, we got lazy. Those changes when we pulled master from the origin should have raised a red flag. It’s considered a bad practice not to look at what’s changed in master before merging in a branch. What one *should* do is either:

* Rebase the flips branch from master before merging. This “rewinds” the branch changes, pulls from master, and then replays the changes back on top of the pulled changes.
* Create a patch and apply *that* to master. This is the sort of thing you might have to do if you’re sending changes to another user, especially if the VCS is not Git.

So let’s restore things to how they were at master:

> git reset --hard HEAD

HEAD is now at ff60b9b Merge branch 'lists'

That throws out our botched merge. Now let’s go back to our branch and rebase it on master:

> git checkout flips

Switched to branch 'flips'

> git rebase master

First, rewinding head to replay your work on top of it...

Applying: Add flips table.

Using index info to reconstruct a base tree...

M sqitch.plan

Falling back to patching base and 3-way merge...

Auto-merging sqitch.plan

CONFLICT (content): Merge conflict in sqitch.plan

Failed to merge in the changes.

Patch failed at 0001 Add flips table.

The copy of the patch that failed is found in:

.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".

If you prefer to skip this patch, run "git rebase --skip" instead.

To check out the original branch and stop rebasing, run "git rebase --abort".

Oy, that’s kind of a pain. It seems like no matter what we do, we’ll need to resolve conflicts in that file. Except in Git. Fortunately for us, we can tell Git to resolve conflicts in sqitch.plan differently. Because we only ever append lines to the file, we can have it use the “union” merge driver, which, according to [its docs](https://git-scm.com/docs/gitattributes#_built-in_merge_drivers):

Run 3-way file level merge for text files, but take lines from both versions, instead of leaving conflict markers. This tends to leave the added lines in the resulting file in random order and the user should verify the result. Do not use this if you do not understand the implications.

This has the effect of appending lines from all the merging files, which is exactly what we need. So let’s give it a try. First, back out the botched rebase:

> git rebase --abort

Now add the union merge driver to .gitattributes for sqitch.plan and rebase again:

> echo sqitch.plan merge=union > .gitattributes

> git rebase master

First, rewinding head to replay your work on top of it...

Applying: Add flips table.

Using index info to reconstruct a base tree...

M sqitch.plan

Falling back to patching base and 3-way merge...

Auto-merging sqitch.plan

Applying: Add functions to insert and delete flips.

Using index info to reconstruct a base tree...

M sqitch.plan

Falling back to patching base and 3-way merge...

Auto-merging sqitch.plan

Ah, that looks a bit better. Let’s have a look at the plan:

> cat sqitch.plan

%syntax-version=1.0.0

%project=flipr

%uri=https://github.com/sqitchers/sqitch-intro/

%syntax-version=1.0.0

%project=flipr

%uri=https://github.com/sqitchers/sqitch-intro/

appschema 2013-12-30T23:19:45Z Marge N. O’Vera <marge@example.com> # Add schema for all flipr objects.

users [appschema] 2013-12-30T23:49:00Z Marge N. O’Vera <marge@example.com> # Creates table to track our users.

insert\_user [users appschema] 2013-12-30T23:57:36Z Marge N. O’Vera <marge@example.com> # Creates a function to insert a user.

change\_pass [users appschema] 2013-12-30T23:57:45Z Marge N. O’Vera <marge@example.com> # Creates a function to change a user password.

@v1.0.0-dev1 2013-12-31T00:01:22Z Marge N. O’Vera <marge@example.com> # Tag v1.0.0-dev1.

lists [appschema users] 2013-12-31T00:39:40Z Marge N. O’Vera <marge@example.com> # Adds table for storing lists.

insert\_list [lists appschema users] 2013-12-31T00:41:29Z Marge N. O’Vera <marge@example.com> # Creates a function to insert a list.

delete\_list [lists appschema users] 2013-12-31T00:41:37Z Marge N. O’Vera <marge@example.com> # Creates a function to delete a list.

flips [appschema users] 2013-12-31T00:32:39Z Marge N. O’Vera <marge@example.com> # Adds table for storing flips.

insert\_flip [flips appschema users] 2013-12-31T00:35:59Z Marge N. O’Vera <marge@example.com> # Creates a function to insert a flip.

delete\_flip [flips appschema users] 2013-12-31T00:36:34Z Marge N. O’Vera <marge@example.com> # Creates a function to delete a flip.

Note that it has appended the changes from the merged “lists” branch, and then merged the changes from our “flips” branch. Test it to make sure it works as expected:

> sqitch rebase -y

Reverting all changes from flipr\_test

- delete\_flip ............... ok

- insert\_flip ............... ok

- flips ..................... ok

- change\_pass @v1.0.0-dev1 .. ok

- insert\_user ............... ok

- users ..................... ok

- appschema ................. ok

Deploying changes to flipr\_test

+ appschema ................. ok

+ users ..................... ok

+ insert\_user ............... ok

+ change\_pass @v1.0.0-dev1 .. ok

+ lists ..................... ok

+ insert\_list ............... ok

+ delete\_list ............... ok

+ flips ..................... ok

+ insert\_flip ............... ok

+ delete\_flip ............... ok

Note the use of [rebase](http://sqitch.org/docs/manual/sqitch-rebase), which combines a [revert](http://sqitch.org/docs/manual/sqitch-revert) and a [deploy](http://sqitch.org/docs/manual/sqitch-deploy) into a single command. Handy, right? It correctly reverted our changes, and then deployed them all again in the proper order. So let’s commit .gitattributes; seems worthwhile to keep that change:

> git add .

> git commit -m 'Add `.gitattributes` with union merge for `sqitch.plan`.'

[flips f5ad242] Add `.gitattributes` with union merge for `sqitch.plan`.

1 file changed, 1 insertion(+)

create mode 100644 .gitattributes

Merges Mastered

And now, finally, we can merge into master:

> git checkout master

Switched to branch 'master'

> git merge --no-ff flips

Merge made by the 'recursive' strategy.

.gitattributes | 1 +

deploy/delete\_flip.sql | 17 +++++++++++++++++

deploy/flips.sql | 16 ++++++++++++++++

deploy/insert\_flip.sql | 17 +++++++++++++++++

revert/delete\_flip.sql | 7 +++++++

revert/flips.sql | 7 +++++++

revert/insert\_flip.sql | 7 +++++++

sqitch.plan | 3 +++

verify/delete\_flip.sql | 7 +++++++

verify/flips.sql | 12 ++++++++++++

verify/insert\_flip.sql | 7 +++++++

11 files changed, 101 insertions(+)

create mode 100644 .gitattributes

create mode 100644 deploy/delete\_flip.sql

create mode 100644 deploy/flips.sql

create mode 100644 deploy/insert\_flip.sql

create mode 100644 revert/delete\_flip.sql

create mode 100644 revert/flips.sql

create mode 100644 revert/insert\_flip.sql

create mode 100644 verify/delete\_flip.sql

create mode 100644 verify/flips.sql

create mode 100644 verify/insert\_flip.sql

And double-check our work:

> cat sqitch.plan

%syntax-version=1.0.0

%project=flipr

%uri=https://github.com/sqitchers/sqitch-intro/

appschema 2013-12-30T23:19:45Z Marge N. O’Vera <marge@example.com> # Add schema for all flipr objects.

users [appschema] 2013-12-30T23:49:00Z Marge N. O’Vera <marge@example.com> # Creates table to track our users.

insert\_user [users appschema] 2013-12-30T23:57:36Z Marge N. O’Vera <marge@example.com> # Creates a function to insert a user.

change\_pass [users appschema] 2013-12-30T23:57:45Z Marge N. O’Vera <marge@example.com> # Creates a function to change a user password.

@v1.0.0-dev1 2013-12-31T00:01:22Z Marge N. O’Vera <marge@example.com> # Tag v1.0.0-dev1.

lists [appschema users] 2013-12-31T00:39:40Z Marge N. O’Vera <marge@example.com> # Adds table for storing lists.

insert\_list [lists appschema users] 2013-12-31T00:41:29Z Marge N. O’Vera <marge@example.com> # Creates a function to insert a list.

delete\_list [lists appschema users] 2013-12-31T00:41:37Z Marge N. O’Vera <marge@example.com> # Creates a function to delete a list.

flips [appschema users] 2013-12-31T00:32:39Z Marge N. O’Vera <marge@example.com> # Adds table for storing flips.

insert\_flip [flips appschema users] 2013-12-31T00:35:59Z Marge N. O’Vera <marge@example.com> # Creates a function to insert a flip.

delete\_flip [flips appschema users] 2013-12-31T00:36:34Z Marge N. O’Vera <marge@example.com> # Creates a function to delete a flip.

Much much better, a nice clean master now. And because it is now identical to the “flips” branch, we can just carry on. Go ahead and tag it, bundle, and release:

> sqitch tag v1.0.0-dev2 -n 'Tag v1.0.0-dev2.'

Tagged "delete\_flip" with @v1.0.0-dev2

> git commit -am 'Tag the database with v1.0.0-dev2.'

[master 230603b] Tag the database with v1.0.0-dev2.

1 file changed, 1 insertion(+)

> git tag v1.0.0-dev2 -am 'Tag v1.0.0-dev2'

> sqitch bundle --dest-dir flipr-1.0.0-dev2

Bundling into flipr-1.0.0-dev2

Writing config

Writing plan

Writing scripts

+ appschema

+ users

+ insert\_user

+ change\_pass @v1.0.0-dev1

+ lists

+ insert\_list

+ delete\_list

+ flips

+ insert\_flip

+ delete\_flip @v1.0.0-dev2

Note the use of the --dest-dir option to sqitch bundle. Just a nicer way to create the top-level directory name so we don’t have to rename it from bundle.

In Place Changes

Uh-oh, someone just noticed that MD5 hashing is not particularly secure. Why? Have a look at this:

> psql -d flipr\_test -c "

SELECT flipr.insert\_user('foo', 'secr3t'), flipr.insert\_user('bar', 'secr3t');

SELECT \* FROM flipr.users;

"

nickname | password | timestamp

----------+----------------------------------+-------------------------------

foo | 9695da4dd567a19f9b92065f240c6725 | 2013-12-31 00:56:20.240481+00

bar | 9695da4dd567a19f9b92065f240c6725 | 2013-12-31 00:56:20.240481+00

If user “foo” ever got access to the database, she could quickly discover that user “bar” has the same password and thus be able to exploit the account. Not a great idea. So we need to modify the insert\_user() and change\_pass() functions to fix that. How?

We’ll use [pgcrypto](https://www.postgresql.org/docs/current/static/pgcrypto.html)’s crypt() function to encrypt passwords with a salt, so that they’re all unique. We just add a change to add pgcrypto to the database, and then we can use it. The deploy script should be:

CREATE EXTENSION pgcrypto;

And the revert script should be:

DROP EXTENSION pgcrypto;

If you’re on PostgreSQL 9.0 or lower, you won’t be able to deploy pgcrypto with a Sqitch change, alas. You’ll have to install it manually, like so:

psql -d flipr\_test -f /path/to/pgsql/share/contrib/pgcrypto.sql

Don’t forget to do this with your staging and production databases, too. Or consider upgrading to PostgreSQL 9.1 or higher; the SQL-level extension support is amazingly useful.

We’re going to use the crypt() and gen\_salt() functions, so in the verify script, let’s make sure that the extension exists *and* that both those functions exist:

SELECT 1/count(\*) FROM pg\_extension WHERE extname = 'pgcrypto';

SELECT has\_function\_privilege('crypt(text, text)', 'execute');

SELECT has\_function\_privilege('gen\_salt(text)', 'execute');

Now we can use pgcrypto. But how to deploy the changes to insert\_user() and change\_pass()?

Normally, modifying functions in database changes is a [PITA](https://www.urbandictionary.com/define.php?term=pita). You have to make changes like these:

1. Copy deploy/insert\_user.sql to deploy/insert\_user\_crypt.sql.
2. Edit deploy/insert\_user\_crypt.sql to switch from MD5() to crypt() and to add a dependency on the pgcrypto change.
3. Copy deploy/insert\_user.sql to revert/insert\_user\_crypt.sql. Yes, copy the original change script to the new revert change.
4. Copy verify/insert\_user.sql to verify/insert\_user\_crypt.sql.
5. Edit verify/insert\_user\_crypt.sql to test that the function now properly uses crypt().
6. Test the changes to make sure you can deploy and revert the insert\_user\_crypt change.
7. Now do the same for the change\_pass scripts.

But you can have Sqitch do it for you. The only requirement is that a tag appear between the two instances of a change we want to modify. In general, you’re going to make a change like this after a release, which you’ve tagged anyway, right? Well we have, with @v1.0.0-dev2 added in the previous section. With that, we can let Sqitch do most of the hard work for us, thanks to the [rework](http://sqitch.org/docs/manual/sqitch-rework) command, which is similar to [add](http://sqitch.org/docs/manual/sqitch-add), including support for the --requires option:

> sqitch rework insert\_user --requires pgcrypto -n 'Change insert\_user to use pgcrypto.'

Added "insert\_user [insert\_user@v1.0.0-dev2 pgcrypto]" to sqitch.plan.

Modify these files as appropriate:

\* deploy/insert\_user.sql

\* revert/insert\_user.sql

\* verify/insert\_user.sql

Oh, so we can edit those files in place. Nice! How does Sqitch do it? Well, in point of fact, it has copied the files to stand in for the previous instance of the insert\_user change, which we can see via git status:

> git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: revert/insert\_user.sql

# modified: sqitch.plan

#

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# deploy/insert\_user@v1.0.0-dev2.sql

# revert/insert\_user@v1.0.0-dev2.sql

# verify/insert\_user@v1.0.0-dev2.sql

no changes added to commit (use "git add" and/or "git commit -a")

The “untracked files” part of the output is the first thing to notice. They are all named insert\_user@v1.0.0-dev2.sql. What that means is: “the insert\_user change as it was implemented as of the @v1.0.0-dev2 tag.” These are copies of the original scripts, and thereafter Sqitch will find them when it needs to run scripts for the first instance of the insert\_user change. As such, it’s important not to change them again. But hey, if you’re reworking the change, you shouldn’t need to.

The other thing to notice is that revert/insert\_user.sql has changed. Sqitch replaced it with the original deploy script. As of now, deploy/insert\_user.sql and revert/insert\_user.sql are identical. This is on the assumption that the deploy script will be changed (we’re reworking it, remember?), and that the revert script should actually change things back to how they were before. Of course, the original deploy script may not be [idempotent](https://en.wikipedia.org/wiki/Idempotence) – that is, able to be applied multiple times without changing the result beyond the initial application. If it’s not, you will likely need to modify it so that it properly restores things to how they were after the original deploy script was deployed. Or, more simply, it should revert changes back to how they were as-of the deployment of deploy/insert\_user@v1.0.0-dev2.sql.

Fortunately, our function deploy scripts are already idempotent, thanks to the use of the OR REPLACE expression. No matter how many times a deployment script is run, the end result will be the same instance of the function, with no duplicates or errors.

As a result, there is no need to explicitly add changes. So go ahead. Modify the script to switch to crypt(). Make this change to deploy/insert\_user.sql:

@@ -1,6 +1,7 @@

-- Deploy flipr:insert\_user to pg

-- requires: users

-- requires: appschema

+-- requires: pgcrypto

BEGIN;

@@ -8,7 +9,7 @@ CREATE OR REPLACE FUNCTION flipr.insert\_user(

nickname TEXT,

password TEXT

) RETURNS VOID LANGUAGE SQL SECURITY DEFINER AS $$

- INSERT INTO flipr.users VALUES($1, md5($2));

+ INSERT INTO flipr.users values($1, crypt($2, gen\_salt('md5')));

$$;

COMMIT;

Go ahead and rework the change\_pass change, too:

> sqitch rework change\_pass --requires pgcrypto -n 'Change change\_pass to use pgcrypto.'

Added "change\_pass [change\_pass@v1.0.0-dev2 pgcrypto]" to sqitch.plan.

Modify these files as appropriate:

\* deploy/change\_pass.sql

\* revert/change\_pass.sql

\* verify/change\_pass.sql

And make this change to deploy/change\_pass.sql:

@@ -1,6 +1,7 @@

-- Deploy flipr:change\_pass to pg

-- requires: users

-- requires: appschema

+-- requires: pgcrypto

BEGIN;

@@ -11,9 +12,9 @@ CREATE OR REPLACE FUNCTION flipr.change\_pass(

) RETURNS BOOLEAN LANGUAGE plpgsql SECURITY DEFINER AS $$

BEGIN

UPDATE flipr.users

- SET password = md5($3)

+ SET password = crypt($3, gen\_salt('md5'))

WHERE nickname = $1

- AND password = md5($2);

+ AND password = crypt($2, password);

RETURN FOUND;

END;

$$;

And then try a deployment:

> sqitch deploy

Deploying changes to flipr\_test

+ insert\_user .. ok

+ change\_pass .. ok

So, are the changes deployed?

> psql -d flipr\_test -c "

DELETE FROM flipr.users;

SELECT flipr.insert\_user('foo', 'secr3t'), flipr.insert\_user('bar', 'secr3t');

SELECT \* FROM flipr.users;

"

nickname | password | timestamp

----------+------------------------------------+-------------------------------

foo | $1$pRNfJjI9$CdcEXJ9xCoJPD.R5Z/7.R1 | 2013-12-31 01:03:15.398572+00

bar | $1$Nf1LcU.p$B9sKzdu8vMgu5oxbimo5P1 | 2013-12-31 01:03:15.398572+00

Awesome, the stored passwords are different now. But can we revert, even though we haven’t written any reversion scripts?

> sqitch revert --to @HEAD^^ -y

Reverting changes to pgcrypto from flipr\_test

- change\_pass .. ok

- insert\_user .. ok

Did that work, are the MD5() passwords back?

> psql -d flipr\_test -c "

DELETE FROM flipr.users;

SELECT flipr.insert\_user('foo', 'secr3t'), flipr.insert\_user('bar', 'secr3t');

SELECT \* FROM flipr.users;

"

nickname | password | timestamp

----------+----------------------------------+-------------------------------

foo | 9695da4dd567a19f9b92065f240c6725 | 2013-12-31 01:03:57.263583+00

bar | 9695da4dd567a19f9b92065f240c6725 | 2013-12-31 01:03:57.263583+00

Yes, it works! Sqitch properly finds the original instances of these changes in the new script files that include tags.

But what about the verify script? How can we verify that the functions have been modified to use crypt()? I think the simplest thing to do is to examine the body of the function, using [pg\_get\_functiondef()](https://www.postgresql.org/docs/9.2/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE). So the insert\_user verify script looks like this:

-- Verify flipr:insert\_user on pg

BEGIN;

SELECT has\_function\_privilege('flipr.insert\_user(text, text)', 'execute');

SELECT 1/COUNT(\*)

FROM pg\_catalog.pg\_proc

WHERE proname = 'insert\_user'

AND pg\_get\_functiondef(oid) LIKE $$%crypt($2, gen\_salt('md5'))%$$;

ROLLBACK;

And the change\_pass verify script looks like this:

-- Verify flipr:change\_pass on pg

BEGIN;

SELECT has\_function\_privilege('flipr.change\_pass(text, text, text)', 'execute');

SELECT 1/COUNT(\*)

FROM pg\_catalog.pg\_proc

WHERE proname = 'change\_pass'

AND pg\_get\_functiondef(oid) LIKE $$%crypt($3, gen\_salt('md5'))%$$;

ROLLBACK;

Make sure these pass by re-deploying:

> sqitch deploy

Deploying changes to flipr\_test

+ insert\_user .. ok

+ change\_pass .. ok

Excellent. Let’s go ahead and commit these changes:

> git add .

> git commit -m 'Use pgcrypto to encrypt passwords.'

[master 4257ae6] Use pgcrypto to encrypt passwords.

13 files changed, 107 insertions(+), 9 deletions(-)

create mode 100644 deploy/change\_pass@v1.0.0-dev2.sql

create mode 100644 deploy/insert\_user@v1.0.0-dev2.sql

create mode 100644 revert/change\_pass@v1.0.0-dev2.sql

create mode 100644 revert/insert\_user@v1.0.0-dev2.sql

create mode 100644 verify/change\_pass@v1.0.0-dev2.sql

create mode 100644 verify/insert\_user@v1.0.0-dev2.sql

> sqitch status

# On database flipr\_test

# Project: flipr

# Change: d3ffa30b72abaf9619ae1f0e726026667612f2b1

# Name: change\_pass

# Deployed: 2013-12-30 17:05:08 -0800

# By: Marge N. O’Vera <marge@example.com>

#

Nothing to deploy (up-to-date)

More to Come

Sqitch is a work in progress. Better integration with version control systems is planned to make managing idempotent reworkings even easier. Stay tuned.

Author

David E. Wheeler [david@justatheory.com](mailto:david@justatheory.com)

License

Copyright (c) 2012-2020 iovation Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

O SOFTWARE É FORNECIDO “NO ESTADO EM QUE SE ENCONTRA”, SEM QUALQUER TIPO DE GARANTIA, EXPRESSA OU IMPLÍCITA, INCLUINDO, SEM LIMITAÇÃO, AS GARANTIAS DE COMERCIALIZAÇÃO, ADEQUAÇÃO A UM DETERMINADO FIM E NÃO VIOLAÇÃO. EM NENHUMA HIPÓTESE OS AUTORES OU TITULARES DOS DIREITOS AUTORAIS SERÃO RESPONSÁVEIS POR QUALQUER RECLAMAÇÃO, DANOS OU OUTRA RESPONSABILIDADE, SEJA EM UMA AÇÃO DE CONTRATO, DELITO OU DE OUTRA FORMA, DECORRENTE DE, FORA DE OU EM CONEXÃO COM O SOFTWARE OU O USO OU OUTRAS NEGOCIAÇÕES NO PROGRAMAS.

[Sobre este site](http://sqitch.org/site/)  
Correções, sugestões e comentários são [bem-vindos](https://github.com/sqitchers/sqitch.org) .