Bike Rental Report

version 1

Johannes Gårdsted Jørgensen - s093457

May 26, 2015

Contents

Bike rental report

Introduction

This report will describe the the creation of a database system for a bike rental company called "Wheels". "Wheels" is a company renting out bikes and its staff includes managers that take care of customers and mechanics that take care of bikes. The "Wheels" company currently only have a single shop featuring three mechanics, four managers where one of them are admins. They hold 20 bikes, and have 35 customers.

Problem formulation

The bike rental company "Wheels" wants an application to manage and hold data on their employees, customers and bikes. They want a system where customers and employees can login and where managers can create rental contracts with customers. The system shall hold basic information on all its users like address, phonenumber, name etc and all users will have their own user-id. Furthermore the system should give special privileges according to the whether the user is a customer, mechanic, manager or admin. They only want admins to be able to create new employees, but managers should be able to create new customers. Mechanics should be responsible of receiving bikes from customers create repair contracts on them so their condition can be monitored. The system should detect if a bike is rented out and only offer bikes for rent that are not currently rented out. Each bike should have its unique serial number as well as details on type of bike stored in the system.

Requirements

Name	Description
R1	Information bikes must be stored in the system, such as serial number, color, description etc.
R2	Information on all users should be stored in the system such as name, sir-name username, password, address etc.
R3	Passwords has to be hashed with a random salt before saved and both salt and hashed password will be saved.
R4	Three types of employees must exist and all must inherit from the abstract employee who again inherits from user. The kinds of employees are: manager, admin and mechanic
R5	Contracts must be saved in the database and must include dates of start and end. The rental contracts must include information on whether they are paid and which customer they are associated to. Repair contracts must include information on which mechanic they are attached to as well as space for an optional repair report. Both must have fields to tell if they have been fulfilled
R6	A commandline application must be created implementing the system features
R7	Users should only be able to view and edit their own contracts

Process

I started out with a different project that ended up taking nearly all mty time until i recognized its lack of importance in regards to database aspects. See more on this in the *appendice/proposed projects*

After i terminated the old project i decided to implement the classic rental system but with a twist being a bike rental instead of cars.

ORM's

I made the mistake to focus on the development of an application and secondarily the database which led me to the use of different Object Relational Mapping libraries like Pony ORM.

ORM's give you the ability to treat the database as a second class citizen and focus on the business logic however when i saw which kind of sql the Pony ORM propuced i was convinced i went in a wrong direction.

Early on i had established a sketch of the overall system with different kinds of users, contracts and bikes and their relations but the ORM gave me the ability to create inheritance.

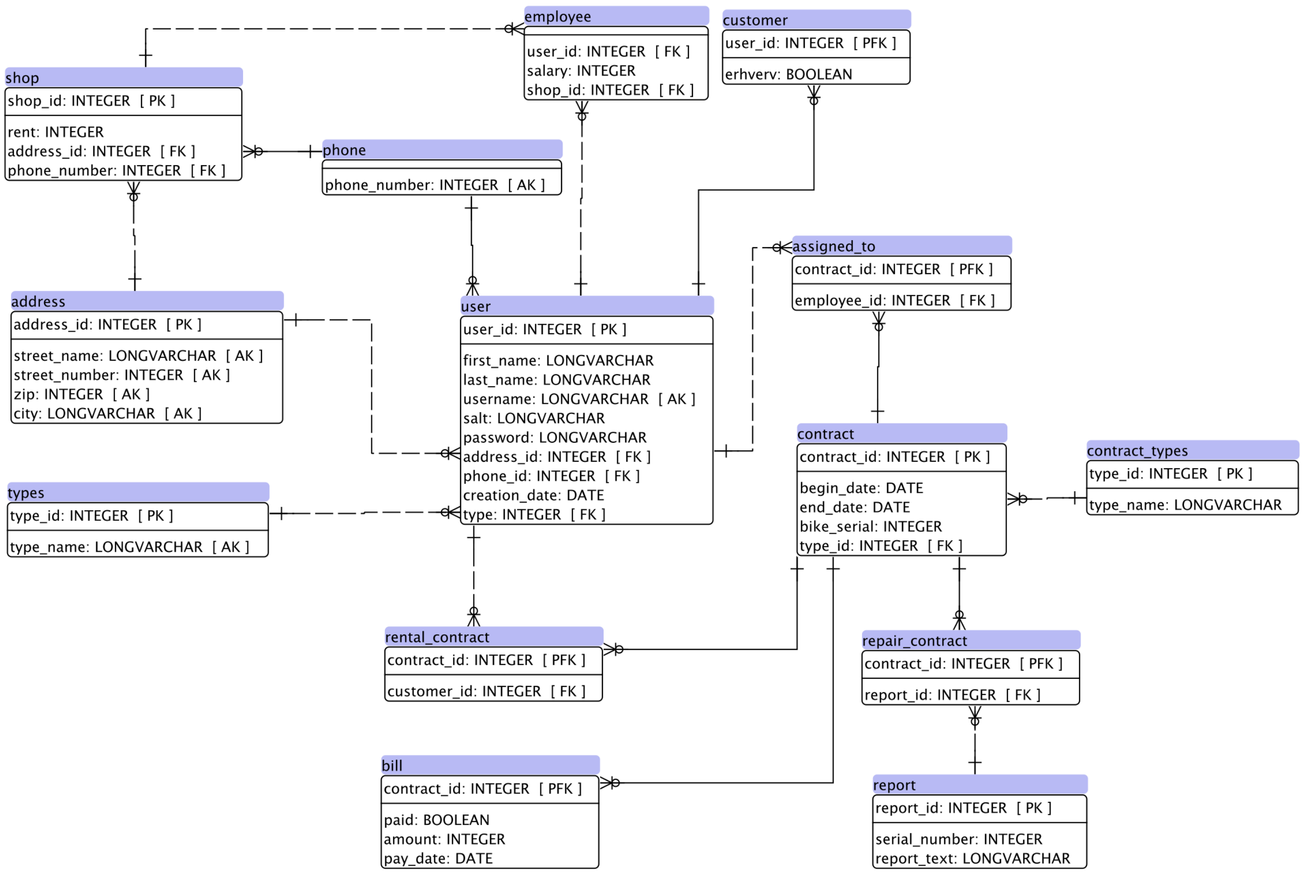
Pony ORM used table per hierachy inheritace where it would store all attributes for all different subclasses in the same table. A special *"class type"* attribute would then be used in the application layer to discriminate which attributes a query would produce. This kind of inheritance produces a lot of unpopulated cells in the database and it lowers the speed of query significantly.

I relized i had implemented a sql-antipattern by producing my database with a focus on being object-oriented.

Database in focus

After skipping the ORM's and diving straight for sql i also lost the rapid development in the process. I decided to focus on implementing the sql and developing the needed queries for an eventual application layer. I wanted to use the Stored Procedures and the Python extension to develop complex functions that would replace a lot of the heavy business logic in the application layer. In this way i could present a solution that would in precise words tell how to implement the application logic without having it as the main focus.

Design



Entity relationship diagram of the bike rental database

The database is designed around the users. Different users create contracts between eachother regarding bikes.

The users are the central aspect and they are represented by a base user table with an attribute telling their type. A table exist for customer specific attributes as well as employee specific attributes. Each have the users.user_id foreign key as their primary key.

The contracts are the glue between users and bikes. At least two kinds of contracts can exist, the rental and the repair contract. All contracts hold the same info and the ability to add a report, could be a repair report. A contract has an attribute telling which type it is. This can be used to discriminate from an application on which users can access which kinds of contracts.

The *contracts.assigned_to* table tells of which users are assigned to which contracts. All contracts has to be assigned an employee but only *rental_contracts* require a customer. The design is flexible enough to allow for new kinds of contracts to be introduced as the *contract_types* table functions as a lookup table to figure out which other table you should look in to find additional information regarding a custom contract.

A bike is conceptualized as a combination between a unique serial number and some type specific details. I decided to place the details in their own table, since they can be shared by many bikes, and use the serial number as the primary key.

Implementation

I will here go through the different tables with a focus on the most complex functionality. All examples depend on the execution of the *appendice/test script*.

Please see the the *appendice/setup script* for a complete walkthrough of the creation of all tables, schemas and functions.

Users

The *user* table holds all attributes held by all users. The attributes hold personal data such as contact information as well as system critical data such as usernames and passwords.

The *user* table hold three foreign keys:

address_id

An int pointing to an address record

phone_id

An int pointing to a phone number

type

An int pointing to a *users.type* record

The *type* attribute is special since it acts as a descriminator in the application layer. The application layer will descriminate different functionalities based on which user type you pocess.

Security

I wanted a secure way of logging in so i implemented password hashing so the database only stores hashed passwords. When you hash passwords you use a random salt to make a hash digest of a clear text password. The output password is regarded as inreversable and secure however you have to store the random salt in order to verify future clear text passwords against the stored password hash.

If the connection to the database is secure, then my implementation is also secure or at least standard.

The diagram below describes the process where the application layer is:

- Creating a user
- Updating the user with a hashed password
- Authenticating the user through the database with a clear text password.

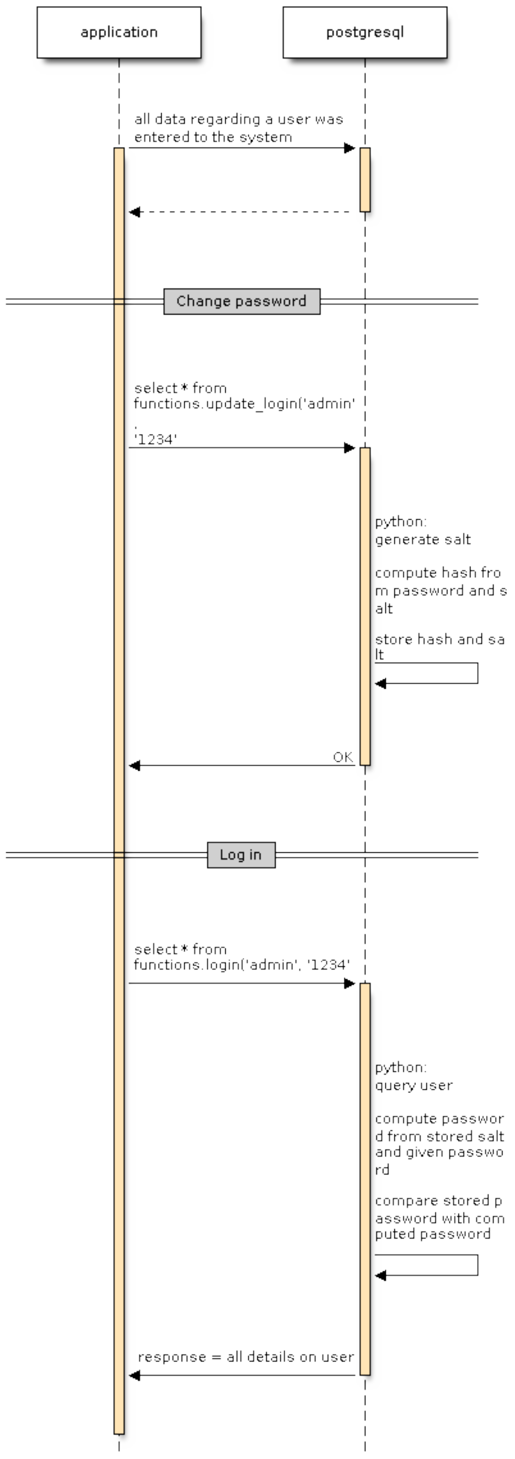

application
postgresql
all data regarding a user was entered to the system
Change password
select * from functions.update_login('admin' , '1234'
python: generate salt
compute hash from password and salt
store hash and salt
OK
Log in
select * from functions.login('admin', '1234'
python: query user
compute password from stored salt and given password
compare stored password with computed password
response = all details on user

Diagram shows the creation of user, the updating of a user's password and the authentication of a user.

I chose to implement the above functionality in python and use the python procedural language for postgresql for execution. I wanted the functionality as a stored procedure so it didn't affect my database design.

If i wanted a function to return a predecided set of attributes i needed a custom type. Postgresqls PL lets you return composite attributes only if you have defined a *"holder"* type for these. The following *login_type* was developed for the *login* function.

```sql
CREATE TYPE functions.login_type AS (
  username text,
  user_id integer,
  first_name text,
  last_name text,
  password text,
  salt text,
  type_name text,
  erhverv boolean,
  salary integer,
  shop_id integer,
  street_name text,
  street_number integer,
  zip integer,
  city text,
  phone_number int

```

I developed an authentication python module with the needed functions for creating hashes and so forth. But i did not install it system wide so my two sotred procedures, *login* and *update_login*, each have it embedded in their source. I have omitted these parts and instead left a print of the python module in the *appendice/hashing*.

Login

Applications use the *login* PL function when logging in. The *login* function uses python as language and depends on a few python modules for hashing and digesting.

```
CREATE OR REPLACE FUNCTION functions.login (username text, password text)
  RETURNS functions.login_type
AS $$
query = """
select u.username, u.user_id, u.first_name, u.last_name, u.password, u.salt,
        t.type_name,
        c.erhverv,
        e.salary, e.shop_id,
        a.street_name, a.street_number, a.zip, a.city,
        p.phone_number
from
        users.user u
left join
        users.customer c
on
        u.user_id = c.user_id
left join
        users.employee e
on
        u.user_id = e.user_id
left join
        details.phone p
on
        p.phone_number = u.phone_id

left join
        details.address a
on
        a.address_id = u.address_id
left join
        users.types t
on
        u.type = t.type_id
where u.username = \'%s\'
""" % username

rows = plpy.execute(query)

```

```
#if results have at least one row
if len(rows) > 0:
    row = rows[0]
    #if the results salt and input password is equal to results password
    if authenticate(row['password'], row['salt'], password):
            #then return all fields from query
        return row
#else return null for all fields
return [None] * 15

$$ LANGUAGE plpythonu;
```

As you see this is not straight python, a few extras are assumed existing as you type. These extras are given at runtime by the postgresql python PL extension.

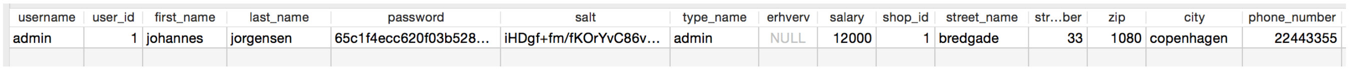
Amongst these are the *SD* dictionary which is a global dictionary offered to handle sharing of data between python scripts. But more interesting is the *plpy* object which is the gateway to execute queries from within the python script. I am using the *plpy* object both in the *login* and *update_login* functions.

Example

The *login* function is used through a select query. This example depends on the existence of an *admin* user with the username *"admin"* and the password *"1234"*. The *login* function will return the fully populated *login_type* if succesfull or a list of null values of similar length.

```
select * from functions.login('admin', '1234');
```

The query will yield the following resultset:



username	user_id	first_name	last_name	password	salt	type_name	erhverv	salary	shop_id	street_name	str...ber	zip	city	phone_number
admin	1	johannes	jorgensen	65c1f4ecc620f03b528...	iHDgf+fm/fKOrYvC86v...	admin	NULL	12000	1	bredgade	33	1080	copenhagen	22443355

Update_login

An application can update a users password only by also generating a new salt and using this to hash the given clear text password.

Applications use the *update_login* function in two scenarios:

1. When creating a user the first step involves creating a user, the second updating the user with a password/salt.
2. When a user wants to alter its password, think *"forgotten password"* functionality.

The *update_login* function uses the *plpy* object for updating a *user* record with a new hashed password/salt pair.

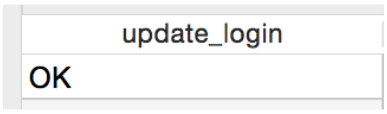
```
CREATE OR REPLACE FUNCTION functions.update_login (username text, password text)
  RETURNS text
AS $$
salt, new_password = create_password_from_clear(password)
query = """
UPDATE users.user SET
salt = \'%s\',
password = \'%s\'
WHERE username = \'%s\';
""" % (salt, new_password, username)
records = plpy.execute(query)
if records.nrows():
    return "OK"
return "ERROR"

$$ LANGUAGE plpythonu;
```

Example

This example will generate take the new password, generate a random salt and hash the given password. This new hashed password will be stored and if all goes well an *"OK"* response will be returned as text.

```
select * from functions.update_login('admin', '5678');
```

The query yields the following result:



update_login
OK

Contracts

Contracts are the business in this project. The contracts tell about which bikes are occupied at any given time, they are responsible for billing and they are also used when the bikes are in need of repair.

The contracts are designed as a base contract table and a type table. The base contract table, *contracts.contract*, only hold universal information such as begin and end date of the contract, which bike it involves and a foreign key pointing to a contract type.

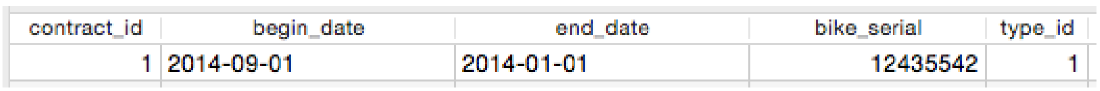
The *contracts.contract_type* is used for telling the application code which other tables to join when querying for all data regarding a contract.

Example

For example if you want to query a specific rental contract you might do the following:

```
select * from contracts.contract c where c.contract_id = 1
```

Will yield



contract_id	begin_date	end_date	bike_serial	type_id
1	2014-09-01	2014-01-01	12435542	1

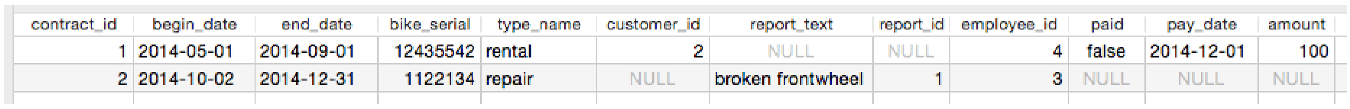
This result tells us that the contract is a *rental_contract* but it does include the *rental_contract* specific attributes in its result set.

The test script creates two kinds of contracts *rental* and *repair* if we include these in our query we will receive resultset including attribute from both tables if existing. Furthermore we also get a contract_type collumn with the descriptive name instead of the *type_id*. Modified to show all contracts and all contract attributes it looks like this:

```
SELECT
contract.contract_id, contract.begin_date,
contract.end_date, contract.bike_serial,
type.type_name,
rental.customer_id,
report.report_text, report.report_id,
assigned.employee_id,
bill.paid, bill.pay_date, bill.amount

FROM contracts.contract contract

LEFT JOIN
contracts.rental_contract rental
ON
rental.contract_id = contract.contract_id
LEFT JOIN
contracts.repair_contract repair
ON
repair.contract_id = contract.contract_id
LEFT JOIN
contracts.contract_types type
ON
```

```sql
type.type_id = contract.type_id
LEFT JOIN
contracts.report report
ON
report.report_id = repair.report_id
LEFT JOIN
contracts.assigned_to assigned
ON
assigned.contract_id = contract.contract_id
LEFT JOIN
contracts.bill bill
ON
bill.contract_id = contract.contract_id

ORDER BY contract_id;
```

This yields the complete data set of a contract including id's of customers if its a rental contract or the assigned employee if its a repair contract.



contract_id	begin_date	end_date	bike_serial	type_name	customer_id	report_text	report_id	employee_id	paid	pay_date	amount
1	2014-05-01	2014-09-01	12435542	rental	2	NULL	NULL	4	false	2014-12-01	100
2	2014-10-02	2014-12-31	1122134	repair	NULL	broken frontwheel	1	3	NULL	NULL	NULL

The result is a denormalized resultset filled with nulls where rows are missing in the involved tables. As expected only attributes regarding the contract type exists in each row.

Bikes

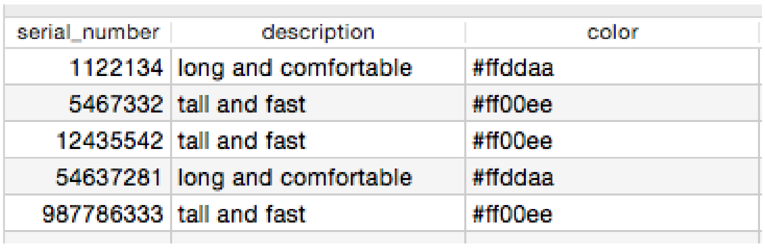
Bikes are the product. They are the unique combination of a bike type and a serial number. A special collumn hold all information regarding the bike type. Like description and color. Bikes are also the subject of reports. Reports can be everything from a report on what was fixed during the repair of a bike, to the planning of bike modificatins etc.

Example

Bikes can be queried in a similar fashion as the other entities. and if you want to include the attributes from the bike type table you can use the following query:

```sql
SELECT
bike.serial_number,
type.description,
type.color

FROM bikes.bike bike

LEFT JOIN
bikes.type type
ON
type.type_id = bike.type_id

ORDER BY bike.serial_number;
```

Which yields the following result set:

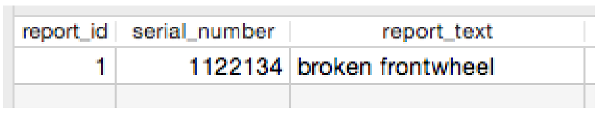


serial_number	description	color
1122134	long and comfortable	#ffddaa
5467332	tall and fast	#ff00ee
12435542	tall and fast	#ff00ee
54637281	long and comfortable	#ffddaa
987786333	tall and fast	#ff00ee

If you want to query which reports there is on a bike you can use the following query:

```
SELECT *
FROM contracts.report report
WHERE report.serial_number = 1122134;
```

Which yields the following resultset:



report_id	serial_number	report_text
1	1122134	broken frontwheel

Normalization

I have normalized the database to a certain extend. I have not gone into the extremes where you reach nearly a "singleton"-ness of all data.

Noprmalization is when you seek to reduce redundancy in your database and try to design it without a certain functionality in mind but instead with focus on the stored data and the ability to query this freely.

First normal form

I have been carefull to design the database so redundancy in table rows are avoided. Furthermore i am using unique identifiers for all tables except those where their attributes are directly coupled to other tables through their primary key being a foreign key.

Second normal form

I have spread out the data and produced primary keys that through foreign keys link together the bits of information that are assoociated.

For example the users are a diverse entity. All users share basic data however their responsibilities and specialities are very diverse. I made sure to include only the necersary information in the base user, and whenever i found attributes which could be shared by many users or other entities i forked it into its own table and left a foreign key pointing to it, the users are pointing to an address row for example.

Another example is that i am using *type* id's in both the user and contract tables to identify which other table i should look in to find more info. This is very flexible since i can add more contract types and more types of users without modifying the *User* table or the *Contract* table. Only the queries would have to be manipulated with extra *joins*.

Third normal form

My project is implementing the third normal form by keeping redundancies at minimum. However the project has not gone all the way.

In the current design i have not normalized areas such as:

user.first_name / user.last_name

These info could be spread into each their own tables like i have done with the details.phone.

However i have succesfully normalized the *contracts* for example. Contracts are defined by a *contract_id*. This id is the foreign key for a lot of attributic tables that all have the ability of adding more data to the context of a contract.

The *contracts.bill* is a unique item. Its fields are sparse and specific for each instance. Only one bill can exist for each contract but a bill can be associated with any given *Type* of contract. therefor the bill is not an attribute of the contract but more an attribute you can associate with a contract_type like the *rental_contract*. The *rental_contract* needs a bill because the rental contract involves outbound busines. The *repair contract* on the other hand is inbound and relies on the mechanic who allready receives a paycheck.

Redundancy is minimal in this setup and tables are only populated if they are needed to. The only downside is the rather complex queries and inserts when you are working with these *splitted out* tables.

Conclusion

I am very happy with the resulting database design and i find it very robust and future proof. I am happy that i went through so many iteraions ending up where i did because i learned a better more clean database design that way.

I sacrified the development of an application because time was running out and i had to choose between good database design and a good application.

I think i could have planned my process much better if i had given the requirements some extra thoughts in the beginning of the semester.

Requirements conclusion

R1 Is fulfilled by the *bikes* schema and its tables.

R2

Is fulfilled by the combination of the *users.user* table and the *details* schema and its tables.

R3

Is fulfilled by the two python PL functions *login* and *update_login*.

R4

This is not directlym implemented since i have learned that inheritance is not a good SQL design. Instead they are implemented as user types together with the customer. Each can of course have their own special table for extra attributes.

R5

Partly fulfilled as the *repair_contract* does not contain information on whether it has been completd. However the rest of R5 is fulfilled by the *contracts* schema, its tables and the collaborating tables from *bikes* etc.

R6

Is not fulfilled since i too late learned that an ORM would conceil bad SQL from me, so instead i focussed on the creation of a good database design with a few helping functions to ease the development of an application. For example all logging in has been taken care of by the database.

R7

Not implemented because of two things:

- I have not implemented an application to handle permissions etc.
- I have not instead implemented SQL roles and access control permissions for individual users.

Bibliography

Appendices

Hashing

The following hashing functionality was implemented in python as a module. _ is used as a descriptor for private, *internal*, functions.

Proposed projects

In the beginning i wanted to extend a project i had been working on for a year called *"the obama puppet"*. The project was a database system where a application would analyse president Obama's weekly addresses and cut every word out as a video sample. These samples would then be saved in a postgresql database and a Procedural Language function would be able to generate a new video from a query based on the stored samples.

I went through with the project but realized too late that it didn't involve enough interesting aspects regarding the database. I set the project on pause for a day and evaluated which features could extend it into a full database project and realized it would be more work than starting a new project instead.

The *"obama puppet"* was terminated right after i had implemented a twitter bot for it, allowing everybody to let him speak their speeches.

It can be found at: http://github.com/sloev/obama-puppet

Inheritance in sql

In general two kinds of inheritance exist in sql. Each have their downsides but both are generally seen as antipatterns where you develop the database from an object oriented paradigm instead of a relational database system paradigm.

Table Per Hierarchy Inheritance

You implement all attributes of all children in a single table with an extra attribute telling the kind of child.

Requires the application to implement logic that creates a meta layer over the existing table. This meta layer will differentiate the table into virtual sub tables, and know which attributes belong to which subclasses. This is not trivial. Might suffer in speed if there is not equality between the quantity of the children.

For example if parent a hast 10 children, where 9 of them are of type b and one of them is of type a. Then in worst case you could say that the database would have to go through one miss before finding 9 hits if searching for type b, which is regarded as fast. But it would have to go through 9 miss before finding 1 hit if searching for type a, which is awful.

Since the table must have all attributes for all sub classes it cant have required attributes that are not shared by all children. This leaves a lot of validation to the application.

Creation/deletion of new types of subclasses would require mangling with the whole table including and potentially adding/removing attributes from all instances of all subclasses.

Table Per Type Inheritance

Or you create an abstract parent implementing all common attributes of its children and then create a specific table for each kind of child with a reference to its parent. You could have an abstract employee table that held all information shared by employees like salary, employer-id etc. And then create a new table pr type of child all having a reference to a unique row in the parent.

Requires the application to join data from multiple tables pr query. One join pr level of inheritance at the least. This is a trivial task but reduces speed pr query.

Setup script

The following script is used to construct all tables, functions, schemas etc for the system.

```
create schema details;
        create table details.address(
                address_id serial primary key,
                street_name text not null,
                street_number int not null,
                zip int not null,
                city text not null,
                unique (street_name, street_number, zip, city)
        );

        create table details.phone(
                phone_number int unique not null
        );

create schema shops;
        create table shops.shop(
                shop_id serial primary key,
                rent int not null,
                address_id int not null references details.address(address_id),
                phone_number int not null references details.phone(phone_number)
        );

create schema users;
        create table users.types(
                type_id serial primary key,
                type_name text not null unique
        );
        create table users.user(
                user_id serial primary key,
                first_name text not null,
                last_name text not null,
                username text not null unique,
                salt text not null,
                password text not null,
                address_id int not null references details.address(address_id),
                phone_id int references details.phone(phone_number),
                creation_date date not null default CURRENT_DATE,
                type int not null references users.types(type_id)
        );

        create table users.customer(
                erhverv bool default false,
                user_id int primary key references users.user(user_id)
        );



        create table users.employee(
                salary integer not null,
                shop_id integer references shops.shop(shop_id),
                user_id int primary key references users.user(user_id)
        );



create schema bikes;
        create table bikes.type(
                type_id serial primary key,
                description text not null,
                color text not null
        );
        create table bikes.bike(
                serial_number serial primary key,
                type_id int references bikes.type(type_id)
        );


create schema contracts;
        create table contracts.contract_types(
                type_id serial primary key,
                type_name text not null
        );

        create table contracts.contract(
                contract_id serial primary key,
                begin_date date not null,
                end_date date not null,
                bike_serial int references bikes.bike(serial_number),
                type_id int references contracts.contract_types(type_id)
        );

        create table contracts.rental_contract(
                contract_id int primary key references contracts.contract(contract_id),
                customer_id int references users.user(user_id)
        );
        create table contracts.report(
                report_id serial primary key,
                serial_number int references bikes.bike(serial_number),
                report_text text not null
        );

        create table contracts.repair_contract(
                contract_id int primary key references contracts.contract(contract_id),
                report_id int references contracts.report(report_id)
        );

        create table contracts.assigned_to(
                contract_id int primary key references contracts.contract(contract_id),
                employee_id int not null references users.user(user_id)
        );

        create table contracts.bill(
                contract_id int primary key references contracts.contract(contract_id),
                paid bool default false,
                amount int not null,
                pay_date date not null
        );




create schema functions;

create language plpythonu;

CREATE TYPE functions.login_type AS (
  username text,
  user_id integer,
  first_name text,
  last_name text,
  password text,
  salt text,
  type_name text,
  erhverv boolean,
  salary integer,
  shop_id integer,
  street_name text,
  street_number integer,
  zip integer,
  city text,
  phone_number int
);


CREATE OR REPLACE FUNCTION functions.login (username text, password text)
  RETURNS functions.login_type
AS $$

from base64 import b64encode
from hashlib import sha256

def authenticate(stored_password, stored_salt, new_password):

    _new_password = _encrypt_password(stored_salt, new_password)
    return _new_password == stored_password

def _encrypt_password(salt, password):

    if isinstance(password, str):
        password_bytes = password.encode("UTF-8")
    else:
        password_bytes = password

    hashed_password = sha256()
    hashed_password.update(password_bytes)
    hashed_password.update(salt)
    hashed_password = hashed_password.hexdigest()
    if not isinstance(hashed_password, str):
        hashed_password = hashed_password.decode("UTF-8")
    return hashed_password


query = """
select u.username, u.user_id, u.first_name, u.last_name, u.password, u.salt,
        t.type_name,
        c.erhverv,
        e.salary, e.shop_id,
        a.street_name, a.street_number, a.zip, a.city,
        p.phone_number
from
        users.user u
left join
        users.customer c
on
        u.user_id = c.user_id
left join
        users.employee e
on
        u.user_id = e.user_id
```

```
left join
        details.phone p
on
        p.phone_number = u.phone_id

left join
        details.address a
on
        a.address_id = u.address_id
left join
        users.types t
on
        u.type = t.type_id
where u.username = \'%s\'
""" % username

rows = plpy.execute(query)

#if results have at least one row
if len(rows) > 0:
    row = rows[0]
    #if the results salt and input password is equal to results password
    if authenticate(row['password'], row['salt'], password):
            #then return all fields from query
        return row
#else return null for all fields
return [None] * 15

$$ LANGUAGE plpythonu;

CREATE OR REPLACE FUNCTION functions.update_login (username text, password text)
  RETURNS text
AS $$
from base64 import b64encode
import os
from hashlib import sha256

def create_password_from_clear( password):
    salt = b64encode(os.urandom(16))
    _password = _encrypt_password(salt, password)
    return salt, _password

def _encrypt_password(salt, password):

    if isinstance(password, str):
        password_bytes = password.encode("UTF-8")
    else:
        password_bytes = password

    hashed_password = sha256()
    hashed_password.update(password_bytes)
    hashed_password.update(salt)
    hashed_password = hashed_password.hexdigest()
    if not isinstance(hashed_password, str):
        hashed_password = hashed_password.decode("UTF-8")
    return hashed_password

salt, new_password = create_password_from_clear(password)
query = """
UPDATE users.user SET
```

```
salt = \'%s\',
password = \'%s\'
WHERE username = \'%s\';
""" % (salt, new_password, username)
records = plpy.execute(query)
if records.nrows():
    return "OK"
return "ERROR"

$$ LANGUAGE plpythonu;
```

Drop script

This script is used to drop all related to this project.

```
drop schema users cascade;
drop schema shops cascade;
drop schema details cascade;
drop schema bikes cascade;
drop schema contracts cascade;
drop type functions.login_type cascade;
drop schema functions cascade;
drop language plpythonu;
```

Test script

All examples in this project are based on the following *test.sql* sql script. When run it will insert:

- Some addresses
- Some phone numbers
- A shop
- 4 users: customer, manager, mechanic, admin

```
insert into users.types (type_name) values
        ('customer'),
        ('mechanic'),
        ('manager'),
        ('admin')
        returning * ;

insert into details.address (street_name, street_number, zip, city) values
        ('slotsgade', 3, 2200, 'copenhagen'),
        ('bredgade', 33, 1080,'copenhagen'),
        ('omvej', 243, 2100,'copenhagen'),
        ('smalgade', 88, 2300,'copenhagen'),
        ('langgade', 63, 2870,'copenhagen')
        returning *;

insert into details.phone(phone_number) values
        (88776633),
        (22443355)
        returning *;

insert into shops.shop (rent, address_id, phone_number) values
        (10000,1, 88776633)
        returning *;

/*
create admin, manager, mechanic, customer
```

```
all users have 1234 passwords
*/
insert into users.user
        (first_name, last_name, username, salt, password, phone_id, address_id, type) va
        ('johannes', 'jorgensen','admin','QZq6FcB1pjT4iAIt42VFvg==',
        '0a1b7bcf28010c62239504354219751aa5655f894a4de8fc2576509f0bfff030', 22443355, 2,
        ('kunde', 'kundesen','kunde','L7O+CHb6RML9eMIcbTJpNA==',
        '9be8e9a15baa8b2f5d5cb233f538cbd0ecb1b17ea74de0a86cb1b41686aafd25',null, 3, 1),
        ('mekaniker','mekanikersen','mekaniker','LUk2WbNqKXTgjnJRXTV7Cw==',
        'bda0cc2f97700f650a5cf9451e4d3060f9aef94c56c615d6a6faf783de6817ae', null, 4, 2),
        ('manager','managersen','manager','LUk2WbNqKXTgjnJRXTV7Cw==',
        'bda0cc2f97700f650a5cf9451e4d3060f9aef94c56c615d6a6faf783de6817ae', null, 5, 3)
        returning *;

insert into users.employee (salary, shop_id, user_id) values
        (12000, 1, 1),
        (5600, 1, 3),
        (7667, 1, 4)
        returning *;


insert into users.customer (user_id) values
        (2)
        returning * ;

insert into bikes.type (description, color) values
        ('tall and fast', '#ff00ee'),
        ('long and comfortable', '#ffddaa')
        returning *;

insert into bikes.bike(serial_number, type_id) values
        (12435542, 1),
        (5467332, 1),
        (1122134, 2),
        (987786333, 1),
        (54637281, 2)
        returning *;

insert into contracts.contract_types(type_name)
        values
        ('rental'),
        ('repair')
        returning * ;


insert into contracts.contract (begin_date, end_date, bike_serial, type_id)
        values
        ('1/5/2014', '1/9/2014', 12435542, 1),
        ('2/10/2014', '31/12/2014', 1122134, 2)
        returning *;

insert into contracts.rental_contract(contract_id, customer_id)
        values
        (1, 2)
        returning *;

insert into contracts.report(serial_number, report_text)
        values
        (1122134, 'broken frontwheel')
        returning *;
```

```sql

insert into contracts.repair_contract(contract_id, report_id)
        values
        (2, 1)
        returning *;

insert into contracts.bill(contract_id, amount, pay_date)
        values
        (1, 100, '1/12/2014')
        returning *;

insert into contracts.assigned_to(contract_id, employee_id)
        values
        (1, 4),
        (2,3)
        returning *;
```