Bike Rental Report

version 1

Johannes Gårdsted Jørgensen - s093457

May 25, 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

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 *test.sql* script found in the appendice.

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.

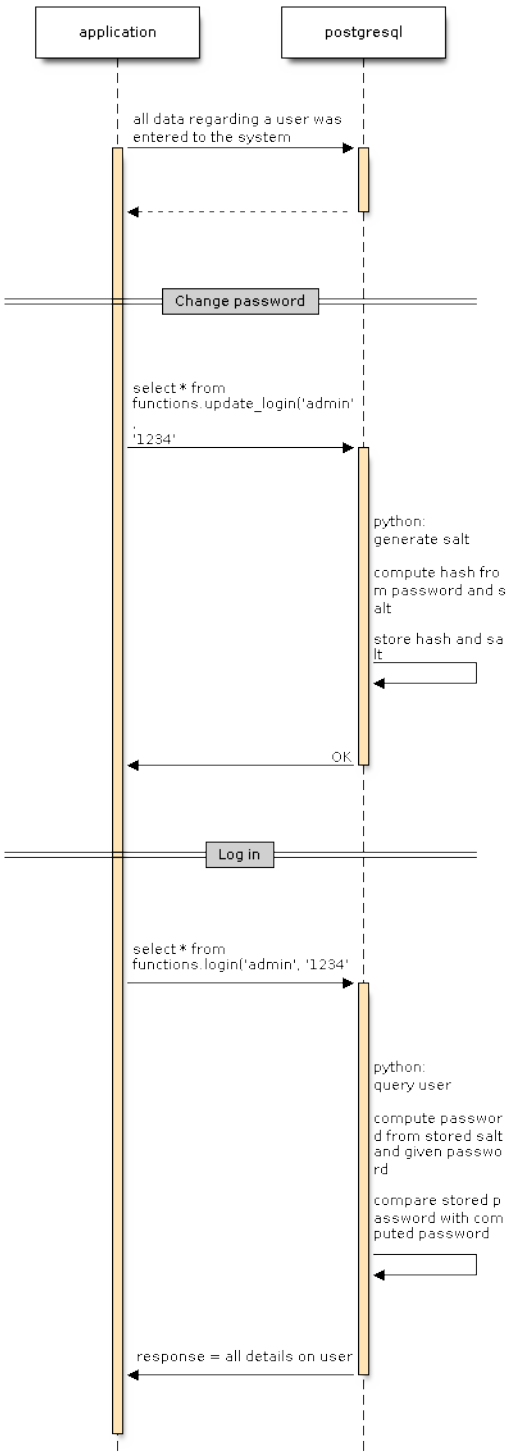

Diagram shows the creation of user, the updating of a user's password and the authentication of a user.

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.

```
1 CREATE TYPE functions.login_type AS (
2   username text,
3   user_id integer,
4   first_name text,
```

```
 5   last_name text,
 6   password text,
 7   salt text,
 8   type_name text,
 9   erhverv boolean,
10   salary integer,
11   shop_id integer,
12   street_name text,
13   street_number integer,
14   zip integer,
15   city text,
16   phone_number int
17 );
```

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.

```
 1 CREATE OR REPLACE FUNCTION functions.login (username text, password text)
 2   RETURNS functions.login_type
 3 AS $$
 4 query = """
 5 select u.username, u.user_id, u.first_name, u.last_name, u.password, u.salt,
 6         t.type_name,
 7         c.erhverv,
 8         e.salary, e.shop_id,
 9         a.street_name, a.street_number, a.zip, a.city,
10         p.phone_number
11 from
12         users.user u
13 left join
14         users.customer c
15 on
16         u.user_id = c.user_id
17 left join
18         users.employee e
19 on
20         u.user_id = e.user_id
21 left join
22         details.phone p
23 on
24         p.phone_number = u.phone_id
25
26 left join
27         details.address a
28 on
29         a.address_id = u.address_id
30 left join
31         users.types t
32 on
33         u.type = t.type_id
34 where u.username = \'%s\'
35 """ % username
36
37 rows = plpy.execute(query)
38
```

```
#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

Conclusion

hello

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)
        );
```

```
45
46
47
48          create table users.employee(
49                  salary integer not null,
50                  shop_id integer references shops.shop(shop_id),
51                  user_id int primary key references users.user(user_id)
52          );
53
54
55
56  create schema bikes;
57          create table bikes.type(
58                  type_id serial primary key,
59                  description text not null,
60                  color text not null
61          );
62          create table bikes.bike(
63                  serial_number serial primary key,
64                  bike_type int references bikes.type(type_id)
65          );
66
67          create table bikes.report(
68                  report_id serial primary key,
69                  serial_number int references bikes.bike(serial_number),
70                  report_text text not null
71          );
72
73  create schema contracts;
74          create table contracts.contract(
75                  contract_id serial primary key,
76                  begin_date date not null,
77                  end_date date not null,
78                  bike_serial int references bikes.bike(serial_number),
79                  report_id int references bikes.report(report_id),
80                  type_id int references contracts.contract_types(type_id)
81          );
82
83          create table contracts.assigned_to(
84                  contract_id int primary key references contracts.contract(contract_id),
85                  employee_id int not null references users.user(user_id)
86                  customer_id int references users.user(user_id)
87          );
88
89          create table contracts.bill(
90                  contract_id int primary key references contracts.contract(contract_id),
91                  paid bool default false,
92                  amount int not null,
93                  pay_date date not null
94          );
95
96          create table contracts.contract_types(
97                  type_id serial primary key,
98                  type_name text not null
99          );
100
101 create schema functions;
102
103 create language plpythonu;
104
```

```
105 CREATE TYPE functions.login_type AS (
106   username text,
107   user_id integer,
108   first_name text,
109   last_name text,
110   password text,
111   salt text,
112   type_name text,
113   erhverv boolean,
114   salary integer,
115   shop_id integer,
116   street_name text,
117   street_number integer,
118   zip integer,
119   city text,
120   phone_number int
121 );
122
123
124 CREATE OR REPLACE FUNCTION functions.login (username text, password text)
125   RETURNS functions.login_type
126 AS $$
127
128 from base64 import b64encode
129 from hashlib import sha256
130
131 def authenticate(stored_password, stored_salt, new_password):
132
133     _new_password = _encrypt_password(stored_salt, new_password)
134     return _new_password == stored_password
135
136 def _encrypt_password(salt, password):
137
138     if isinstance(password, str):
139         password_bytes = password.encode("UTF-8")
140     else:
141         password_bytes = password
142
143     hashed_password = sha256()
144     hashed_password.update(password_bytes)
145     hashed_password.update(salt)
146     hashed_password = hashed_password.hexdigest()
147     if not isinstance(hashed_password, str):
148         hashed_password = hashed_password.decode("UTF-8")
149     return hashed_password
150
151
152 query = """
153 select u.username, u.user_id, u.first_name, u.last_name, u.password, u.salt,
154         t.type_name,
155         c.erhverv,
156         e.salary, e.shop_id,
157         a.street_name, a.street_number, a.zip, a.city,
158         p.phone_number
159 from
160         users.user u
161 left join
162         users.customer c
163 on
164         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
```

```
225
226 salt, new_password = create_password_from_clear(password)
227 query = """
228 UPDATE users.user SET
229 salt = \'%s\',
230 password = \'%s\'
231 WHERE username = \'%s\';
232 """ % (salt, new_password, username)
233 records = plpy.execute(query)
234 if records.nrows():
235     return "OK"
236 return "ERROR"
237
238 $$ LANGUAGE plpythonu;
```

Drop script

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

```
1 drop schema users cascade;
2 drop schema shops cascade;
3 drop schema details cascade;
4 drop schema bikes cascade;
5 drop schema contracts cascade;
6 drop type functions.login_type cascade;
7 drop schema functions cascade;
8 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

```
 1 insert into users.types (type_name) values
 2         ('customer'),
 3         ('mechanic'),
 4         ('manager'),
 5         ('admin')
 6         returning * ;
 7
 8 insert into details.address (street_name, street_number, zip, city) values
 9         ('slotsgade', 3, 2200, 'copenhagen'),
10         ('bredgade', 33, 1080,'copenhagen'),
11         ('omvej', 243, 2100,'copenhagen'),
12         ('smalgade', 88, 2300,'copenhagen'),
13         ('langgade', 63, 2870,'copenhagen')
14         returning *;
15
16 insert into details.phone(phone_number) values
17         (88776633),
18         (22443355)
19         returning *;
20
21 insert into shops.shop (rent, address_id, phone_number) values
22         (10000,1, 88776633)
```

```
23          returning *;
24
25  /*
26  create admin, manager, mechanic, customer
27  all users have 1234 passwords
28  */
29  insert into users.user
30          (first_name, last_name, username, salt, password, phone_id, address_id, type) val
31          ('johannes', 'jorgensen','admin','QZq6FcB1pjT4iAIt42VFvg==',
32          '0a1b7bcf28010c62239504354219751aa5655f894a4de8fc2576509f0bfff030', 22443355, 2,
33          ('kunde', 'kundesen','kunde','L7O+CHb6RML9eMIcbTJpNA==',
34          '9be8e9a15baa8b2f5d5cb233f538cbd0ecb1b17ea74de0a86cb1b41686aafd25',null, 3, 1),
35          ('mekaniker','mekanikersen','mekaniker','LUk2WbNqKXTgjnJRXTV7Cw==',
36          'bda0cc2f97700f650a5cf9451e4d3060f9aef94c56c615d6a6faf783de6817ae', null, 4, 2),
37          ('manager','managersen','manager','LUk2WbNqKXTgjnJRXTV7Cw==',
38          'bda0cc2f97700f650a5cf9451e4d3060f9aef94c56c615d6a6faf783de6817ae', null, 5, 3)
39          returning *;
40
41  insert into users.employee (salary, shop_id, user_id) values
42          (12000, 1, 1),
43          (5600, 1, 3),
44          (7667, 1, 4)
45          returning *;
46
47
48  insert into users.customer (user_id) values
49          (2)
50          returning * ;
51
```