Bike Rental Report

version 1

Johannes Gårdsted Jørgensen - s093457

May 25, 2015

Contents

Bike rental report **1**
Introduction 1
Problem formulation 1
Requirements 1
Design 1
Table Per Hierarchy Inheritance 2
Table Per Type Inheritance 2
Problem solution 2
Users 2
Security 3
Login 5
Example 6
Update_login 6
Conclusion 7
Bibliography 7
Appendices 7
Hashing 7

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

Design

I started this project from the Python point of view. Which meant i focussed on how i would implement the application and secondary the application.

This proved early to be a wrong decision. Signs of this showed through my use of object relational mapping frameworks such as Pony ORM for python. Nothing is wrong with Pony ORM if you want to degrade your database to secondary citizen in the system. However since this class is about database design i wanted to focus on the development of a data centric system with lower priority on the implementation of the application layer.

In the start i had developed a schematic representation full of inheritance since i come from object oriented programming paradigms.

I wanted different roles for users in my system and wanted to be able to restrict actions based on these rules. Pony ORM gave me easy inheritance so i could implement a base user with details like password, username, address etc and then inherit from that user in my employee and customer. Admins, mechanics and managers would then inherit from the employee and so forth. Before long i recognized i needed to see which kind of tables Pony ORM were generating and i was horrorfied to see that it used table per hierachy inheritance.

As i said before inheritance is not a database concept in the normal sense. Postgresql has implemented inheritance but it lacks a lot of important functionality and is in general not recommended, they even have a disclaimer in their documentation telling of the faulty implementation [2].

In normal sql you implement inheritance by going with one of two popular concepts:

- Table per hierachy
- Table per type

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.

Problem solution

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.

---

1 https://pragprog.com/book/bksqla/sql-antipatterns

2 ndosn

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.

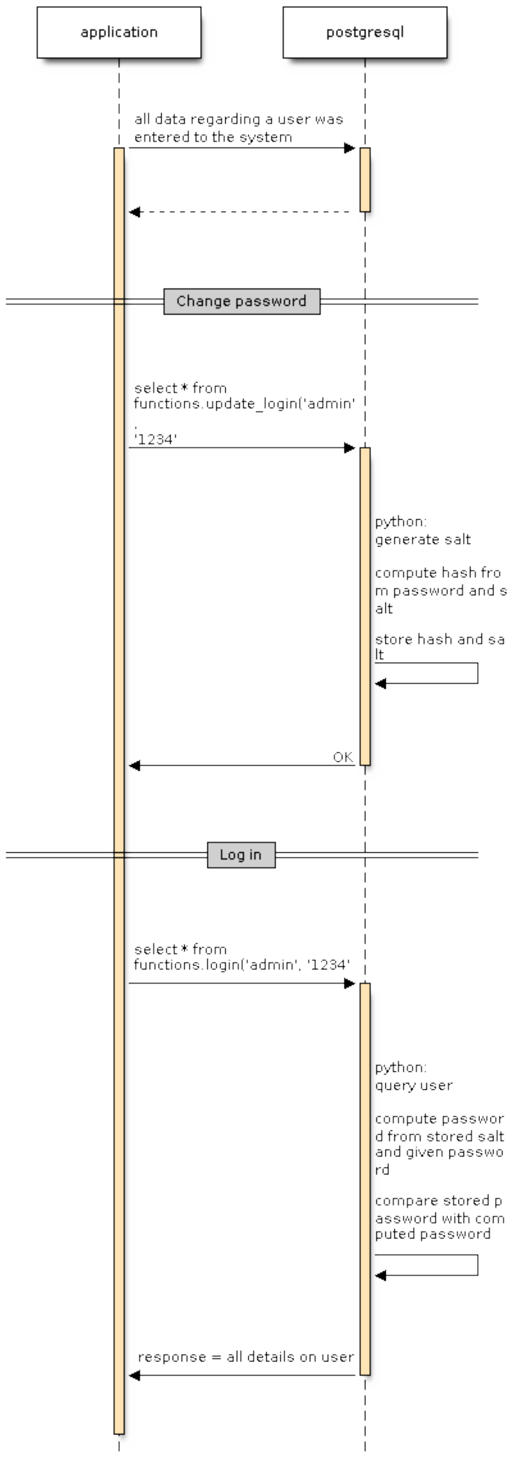

application
postgresql
all data regarding a user was entered to the system
Change password
select * from functions.update_login('admin', '1234'
python:
generate salt
compute hash from password and salt
store hash and salt
OK
Log in
select * from functions.login('admin', '1234'
python:
query user
compute 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 #stringformatted username

```

```
resultset = plpy.execute(query)

#if results have at least one row
if len(resultset) > 0:
    row = resultset[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.

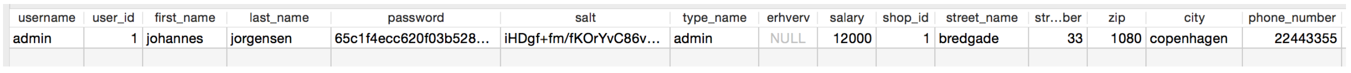
Amongst these are the *SD* dictionary which is a global dictionary offered to handle sharing of data between python scripts. But more interesting is the *plpy* object which is the gateway to execute queries from within the python script. I am using the *plpy* object both in the *login* and *update_login*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

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.

```
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)

resultset = plpy.execute(query)

if resultset.nrows():
    return "OK"
else:
  return "ERROR"

```

```
$$ LANGUAGE plpythonu;

```

Here the *plpy* object is used for updating a *user* record with a new hashed password/salt pair.

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.

```
from base64 import b64encode
import os
from hashlib import sha256
import logging

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

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

```