Coupon System - Project Notes

Maor Barazani

May 11, 2018
Authored by: Maor Barazani

Coupon System - Project Notes

Maor Barazani

Design Decisions

Regarding **parameter values** for the different entities:

	NOT NULL	***UNIQUE***	***UPDATEABLE***
COMPANY	comp_name, password, email	comp_name	password, email
COUPON	title, amount, price	title	price, end date
CUSTOMER	cust_name, password	cust_name	password

some of these parameter restrictions are required in the project description, and some I had chose to restrict or not. this is done mainly for simplicity and the ability to create objects (especially coupons) without having to provide lot's of details, but also to reflect some kind of logic - a coupon description, for example, is not required for it to exist and function properly in the system for now, in my opinion.
moreover, **some** of these restrictions are checked and ensured on the Facade level, before any communication with the DB has happened- for example, checking a password's length to be under 8 chars, and checking if a name is available for use or is already taken.
some attributes are **NOT** checked and this is to prevent the Facade level programming from getting too entangled and hard to understand. for example, the coupon description length restriction (VARCHAR(50)) is only enforced on the DB level and will result in an SQL Exception (that will trigger a DAO-level exception) if exceeded. so in some cases the DAO-SQL level is only the 2nd line of defense, while in other cases it's the only one (coupon description length, for example)
Also I'm assuming some common-sense will be used when testing and using the system - specifically I'm not restricting any 'formatting' related issues; for example, a company name like "$# a_)", coupon price of "0.00053" and emails that look like "name*(&.com@munist" are **allowed** in the system, for now, for simplicity sake.
I understand more logical, esthetical and business-related restrictions can be coded, but the system is designed to allow the implementation of more of these in the Facade level in the future with ease, if required. I didn't go crazy on this one.

Regarding **JOIN tables**:

I chose to stick with the instructions in regarding to the joined data tables, listing a company's coupons, and a customer's coupons. these tables only list id values for both company/customer and coupon, acting as indexes. for this reason I'm using 'population' methods, to insert/remove from these tables on any coupon creation, purchase or removal, and SQL INNER-JOIN statements to retrieve data. this also means I'm **NOT** building actual complex SQL JOIN tables.

Regarding **Login**:

On the project instructions we are required to include a 'login' method on the Façade level- on the 'CouponClientFacade' interface, and on all of its implementations.
Although this **can** work, it doesn't seem to be very logical- so I chose to discard the login procedure on the facade level, and work with it on the CouponSystem level. Each successful login return's a new Facade instance, which also initializes a company/customer java bean to be used on some of the facade methods. The **Factory** design pattern can also come in handy here to manufacture the different Facades, but I chose to keep it all in the CouponSystem.login()- seems more appropriate this way.

Regarding **JavaBeans**:

On the project instructions we are required to add a Collection attribute to some of the java beans, for example- a Company object holds a Collection<Coupon>. These attributes are currently not used anywhere in the project, and not initialized on anything. They only have a get-set method which is irrelevant for the time being. In the future, this can be 'populated' during the DAO method `public Collection<Coupon> getCoupons(Company comp)` if needed.

Regarding **DateMaker**:

In order to easily generate SQL Dates in a year-month-day input format, I've built a DateMaker utility class, that has 1 method - `public static Date setDate(int year, int month, int day)` it's used statically to generate dates when creating a new coupon. it's also set up to increment moths by one so that month=03 equals March, for example.

File Resources:

In order to demonstrate the ability of not having potentially-updatable data hard-coded, and instead read that data from a file, I chose to implement a Scanner to get text input from text files for the server url and the admin login details. this of course allows for the quick and easy changing of these values, even by non-coders, on an easily accessible file.

Alternate solutions and Additional Methods

There are a few places in the System that 1 task can be done in different ways; I would mostly prefer to code the more efficient and elegant solution, and will try to note my decisions here. Also, I added a few additions that are not required in the project description but make life easier for me as the Admin while building-testing the system, as well as providing some flexibility or efficiency. I will **try** to note them all here.

- Name Availability: I wrote an additional method for all DAO interfaces that looks roughly like this `public Company getCompanyByName(String name)`. this is used to get an **entity** (company, customer or coupon) by its **name** (which is always unique and enforced that way). it's mainly used to check if the name is available for use before creating a new entity, but it also returns the existing object (if exists)- it might come in handy in the future (but is not currently being used in the project).

an alternate way to check name availability (for Company, for example) is to use the DAO method `public Collection<Company> getAllCompanies()`, iterate over the returned collection and checking each objects name and compare it to the name of the new Company. however I find this to be inefficient- it involves potentially creating a LOT of java objects in memory. the `getCompanyByName` method achieves this on the DB level easily. **as a general rule-** I try to use DAO level sorting and sifting with designated methods that execute a specifically designed SQL statements, and not create a ton of unnecessary java objects, and iterate over them for sorting and picking the right ones.

- `public long createCompany(Company comp)` - every 'create' method on the DAO level is also configured to return a number- which is the automatically-generated-unique-primary-key '**id**' attribute in the DB. for now, this is done for self checking, and easily accessing entities on the DB using their id value.
- the Facade level updating methods like `public void updateCoupon(Coupon newCoupon)` have built in parameter value restrictions for the updatable attributes, but are using the 'old' object that exists in the DB to update its updatable parameters only, and execute the SQL query. this is because the method is required to take in a Coupon object as a parameter, and potentially this object might have his **other** attributes changed. for example, `newCoupon.setMessage("this is the wrong coupon description");` might give a wrong/illegal value to the newCoupon object, but this will not reflect in the update. there are other ways to achieve this, but I find this pretty elegant, and easy to change in the future.
- The CustomerFacade `public void purchaseCoupon(Coupon coupon)`method checks the purchased coupon is not expired `if (coupon.getEndDate().before(new Date()))`- this seems a bit redundant as there is a daily thread on the system that will delete expired coupons, but I implemented this check anyway as a 2nd line of defense, and as this is required in the project instructions.
- DailyCouponExpirationTask.stopTask() method is required in the project description so I coded it in, but... using the 'quit' boolean in the task's run() while loop means that 'stopping' that thread might actually take up to 24hrs. to actually STOP the daily task I implemented some different code on the CouponSystem.shutdown() method, to kill the daily thread: `dailyThread.interrupt();` `dailyThread.join();`. this will ensure the successful termination of the daily thread, before we can proceed to the closing of the connection pool.
- I added a system to allow the documentation of income for companies. this involves an extra DB table 'company_income' and an additional method in the CompanyDAO `public void updateCompanyIncome(Coupon coupon)` that get's a coupon object after a successful CustomerFacade purchase, finds out the company it is associated with, and creates/updates an income for this company, based on the coupon's price. this feature is being demonstrated near the end of the project test.
- `getOneCompanyCoupon(long compId, long couponId)` in the CouponDbDao is used from the CompanyFacade, for a specific company to get one of it's own coupons. the 'regular' `getCoupon(long couponId)` method as requested in the project instructions only take a couponId as parameter, and can get ANY coupon from the system, not one for a specific company.

Test layout

in the code, every intentional ILLEGAL action specified here in the script, 'lives' in it's own try-catch block, so the whole test flow will not be compromised on individual exceptions. every one of these is treated in the same way with: `System.err.println(e.getMessage());`. the entire test layout 'lives' in one big try-catch-finally block, where system shutdown is invoked at the end.

the test program can be found under `package z.utilities;`

Test Script

drop all tables

create all tables

initialize coupon system

ILLEGAL login as admin wrong password

login as admin

Admin Facade:
create company 'nike'

ILLEGAL create company with long password

create company 'apple'

ILLEGAL create duplicate company name

create company 'orange'

ILLEGAL remove company that doesn't exist

update company 'orange'

ILLEGAL update company that doesn't exist

print get company 'nike'

ILLEGAL get company

print get all companies

ILLEGAL get all customers

create customer 'Shelly'

create customer 'Ricky'

create customer 'Ziv'

ILLEGAL create customer with null name

update customer 'Shelly'

ILLEGAL update customer with long password

print get customer

print get all customers

****

ILLEGAL login as company that doesn't exist

login as company 'nike'

NIKE Company facade:

ILLEGAL get all company coupons

create coupon

ILLEGAL create coupon with negative amount

create coupon

create coupon

ILLEGAL remove coupon that doesn't exist

update coupon

ILLEGAL update coupon with end date before start date

print get all company coupons

print get coupons by type

ILLEGAL get coupons by nonexistent type

print get coupons up to date

print get coupons by price

******

login as company 'apple'

APPLE Company facade:

create coupon

create coupon

ILLEGAL create duplicate coupon name

******

login with customer 'Shelly'

SHELLY Customer facade:

ILLEGAL get customer coupon

purchase coupon from 'nike'

purchase coupon from 'apple'

purchase coupon from 'nike'

ILLEGAL purchase duplicate coupon from 'apple'

print get all customer coupon

print get coupons by type

ILLEGAL get coupons by price

******

login with customer 'Ziv'

ZIV Customer facade:

purchase coupon from 'apple'

purchase coupon from 'apple'

purchase coupon from 'nike'

print get all coupons

ADMIN: remove company 'apple'

ZIV: print get all coupons

login with customer 'Ricky'

RICKY Customer facade:

purchase coupon from 'nike'

purchase coupon from 'nike'

NIKE: remove an already purchased coupon

RICKY: print get all coupons

ADMIN: remove customer 'Ricky'

RICKY: ILLEGAL get all coupons

NIKE: get company income

NIKE: create coupons with end_date in the past

NIKE: print get all coupons

ADMIN: MANAULLY call the DAO methods to delete expired coupons

NIKE: print get all coupons - expired is removed

SYSTEM: shutdown