**CONCORIDA UNIVERSITY**

Department of Computer Science & Software Egineering

Files and Databases

COMP 5531

Monday, August 12th, 2013

**MAIN PROJECT**

Esteban Garro

**ID: 9778292**

**INTRODUCTION**

The project consists of a full, working system for a Computer Store. The Computer Sales and Repair Store System (CSRSS) is available on-line at:

<https://clipper.encs.concordia.ca/~kxc55311/>

CSRSS allows you to simulate sales, upgrades and repairs, mock on-line sales, search for parts and update them, keep an inventory and simulate purchase batches of the same part at different price points, keep a history of devices brought to the store for repairs, process payments for employees and see their activity, add or update employee information, view a daily store report and calculate and compare store revenues over different periods of time.

The system uses Twitter Bootstrap Framework for the layouts and the CSS. Javascript, JQuery and its built-in AJAX handlers were used to deal with user interaction. The server (backend) was programmed from scratch on plain PHP using a model-view-controller (MVC) design paradigm. The database system used is MySQL and all calls to the database are handled by PHP's native MySQL support. No other technology was used on this project.

For the user interface, a friendly dynamic approach was taken, where the user is presented with only one option at the time, and based on that selection other options populate the user interface in a progressive manner. This way, the workflow is completely intuitive and requires no further training/thought.

**POINTS TO IMPROVE**

Even though CSRSS is a working system supporting all its features flawlessly, CSRSS has not yet been fully revised/tested for commercial or real-life use. The purpose of the project was to demonstrate skills implementing a client-server system while using all learnt database concepts. For production, there are many areas of improvement (especially those dealing with security issues) that were not fully implemented because that is beyond the scope of the purpose of this course/project.

The sessions are handled using basic PHP’s native sessions support. A better implementation of the sessions using encrypted-cookies is desirable. The site is not protected against Cross-site Request Forgery (CSRF), and this could be addressed on a second version of this system by means of a CSRF Token.

Even though string filtering was used on all form submissions to the server, there is no guarantee that the site is not vulnerable to Cross-site scripting (XSS). A check/test on all submission forms is desirable.

No test has been conducted to assure that AJAX calls to the server are being properly authenticated.

Passwords are being stored using and old MD5-with-constant-salt hash mechanism, which has been proven to be weak and vulnerable to users attacks. No password history is being maintained and there are no restrictions on the length or characters of the username and passwords. All these issues would have to be fixed before releasing the system to real-world exposure.

**THE DATA MODEL**

The following design considerations were taken into account while developing the ER diagram that follows:

1. Since passwords are sensitive information, username and passwords are kept on a separate table.
2. Payments to employees reflect only the cumulative amount based on commission made through repairs, upgrades and online sales for the selected periods and they don’t include employee’s salary.
3. For employees, address and phone number are entirely optional.
4. A Service Instance (an Upgrade, Repair, Sale or Online Sale) is performed by exactly one employee.
5. Each completed Service Instance stores the amount that goes to the employee and the store revenue using current inventory and commission information valid at the time the service instance was created. This way, increases or decreases in part’s price or changes in employee’s commissions don’t affected Store Revenue calculations.
6. The purchase of a device on the store doesn’t generate a Device History entry.
7. The last whole-price of a part (the price paid by the store owner to acquire it) is stored on the Inventory table for that part only as a reference and it’s not used on store-revenue calculations. Store-revenue calculations use the purchase history of that part to determine the proper amounts that go to the employee and that contribute to store revenues. For example, if a batch of 5 items for part 123 was purchased for $4.00 a piece and a second batch of 5 items of the same part was purchased for $2.50, there is now 10 available items that would generate different store revenues when selling the part for let’s say $6.00. A logic of high-price goes first was applied so that the first 5 items sold would contribute $2.00 to the store revenue, whereas subsequent sales of the same item would contribute $3.50. If a sale is online through an employee with an online commission of 50%, then half of those $3.50 would go to the employee and the other half contributes to the store revenue.
8. The default-cost for a Service Instance is stored on the DefaultCost table and it’s automatically populated upon creation of a Service Instance. This amount, however, is editable at the time of the transaction according to project requirements.
9. The seniority of an employee is calculated based on the number of years from the first\_day\_of\_work date stored on the Employees database. Upon creation of a new employee identity, a MySQL trigger is active to generate the appropriate service commission for that employee (this value is stored as service\_fee on the Employee’s table). The amounts are hard-coded on the trigger and they could be changed by modifying the actual trigger:

delimiter $$

create trigger defaultServiceFeeTrigger

before insert on CSRSS\_Employees

for each row

begin

if AGE(NEW.first\_day\_of\_work) < 1 THEN

set NEW.service\_fee = 10;

elseif AGE(NEW.first\_day\_of\_work) < 2 THEN

set NEW.service\_fee = 15;

elseif AGE(NEW.first\_day\_of\_work) < 3 THEN

set NEW.service\_fee = 20;

elseif AGE(NEW.first\_day\_of\_work) < 4 THEN

set NEW.service\_fee = 25;

else

set NEW.service\_fee = 30;

end if;

end;$$

delimiter ;

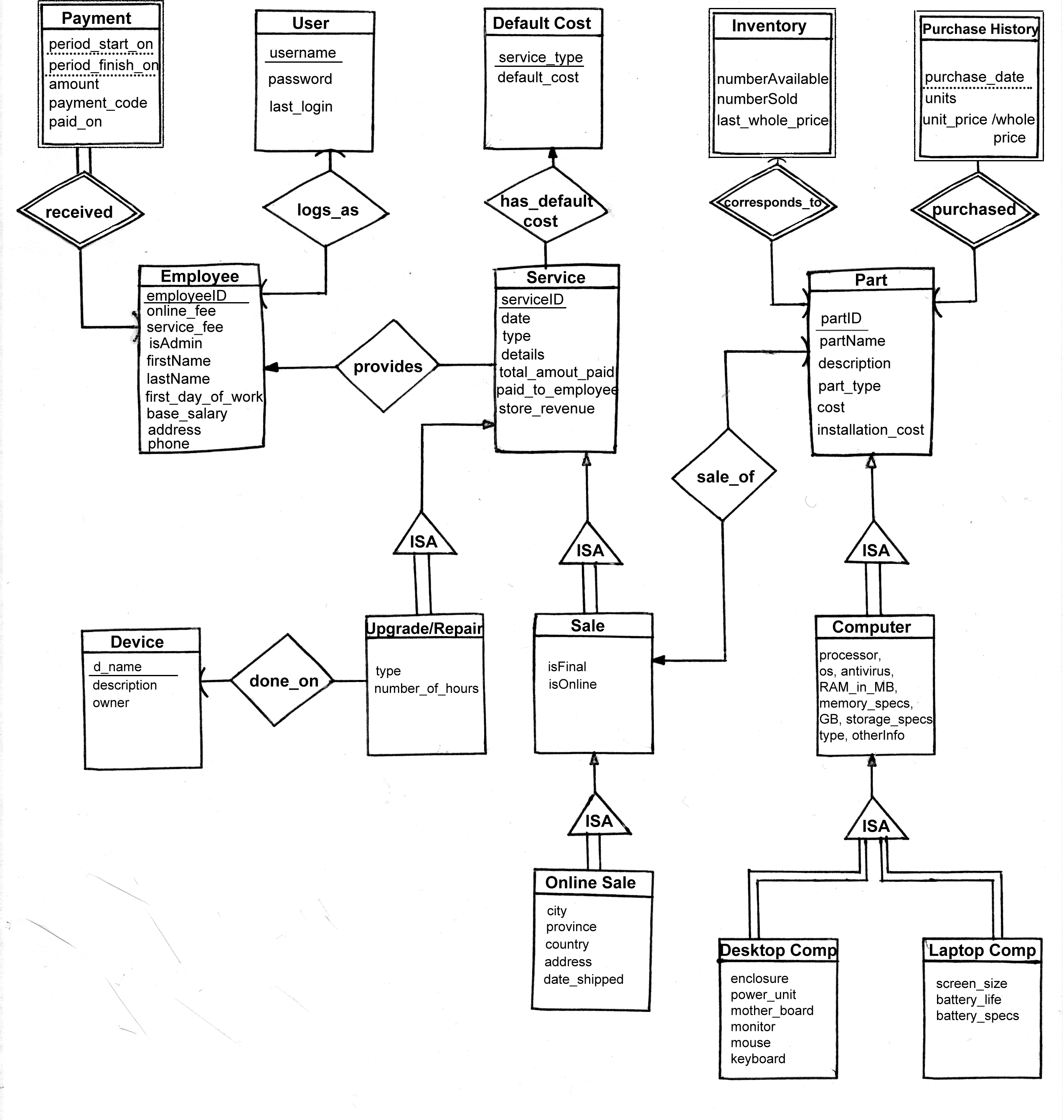
Where AGE is a convenience MySQL function currently active on the server that was created using the following query:

CREATE FUNCTION AGE(BDAY DATE)

RETURNS tinyint unsigned

RETURN (YEAR(CURDATE())-YEAR(BDAY)) - (RIGHT(CURDATE(),5) < RIGHT(BDAY,5));

The resulting ER diagram resulting from the project requirements and the previous considerations is shown here:



From this diagram, the following relationships arose:

* received(employeeID, period\_start\_on, period\_finish\_on, amount, payment\_code, paid\_on)
* logs\_as(employeeID, username, password, last\_login)
* provides(serviceID, employeeID)
* has\_default\_cost(serviceID,service\_type)
* done\_on(serviceID,d\_name)
* sale\_of(serviceID,partID)
* corresponds\_to(partID, numberAvailable, numberSold, last\_whole\_price)
* purchased(partID, purchase\_date, units, unit\_price)

Where the PRIMARY keys have been underlined. After eliminating redundancies and in fact merging the totality of these relationships into the corresponding strong entity schemas (all relationships in our case are one-to-one, many-to-one, and one-to-many), the final Database consists of the following relationship schemas:

Payments(**employeeID**, period\_start\_on, period\_finish\_on, amount, payment\_code, paid\_on)

Employees(employeeID, **username**, online\_fee, service\_fee, isAdmin, firstName, lastName, first\_day\_of\_work); where username references Users, on DELETE CASCADE because when an employee leaves, he shouldn’t be allowed to log in again.

Users(username, password, last\_login); where this apparently redundant relationship is kept there instead of being merged to Employees to keep all the passwords on a separate table.

Services(serviceID, **type**, **employeeID**, date, details, total\_amount\_paid, paid\_to\_employee, store\_revenue); where employeeID references Employee, on DELETE SET NULL because we don’t want to lose Service history items when an Employee is gone, and type references DefaultCost(service\_type) on DELETE SET NULL again for the same reason.

DefaultCost(service\_type, default\_cost);

Devices(d\_name, description, owner);

UpgradesRepairs(**serviceID**, **d\_name**, type, number\_of\_hours); where serviceID references Services on DELETE CASCADE and d\_name references Devices on DELETE SET NULL.

OnlineSales(**serviceID**, city, province, country, address, date\_shipped); where serviceID references Sales on DELETE CASCADE.

Sales(**serviceID**, **partID**, isFinal, isOnline); where serviceID references Services, on DELETE CASCADE and partID references Parts on DELETE SET NULL because in the event a Part is no longer in existence we still want to keep the record of it sale in the past.

Parts(partID, partName, description, part\_type, cost, installation\_cost);

Inventory(**partID**, numberAvailable, numberSold, last\_whole\_price); where partID references Parts on DELETE CASCADE.

PurchaseHistory(**partID**, purchase\_date, units, unit\_price) where partID references Parts on DELETE SET NULL.

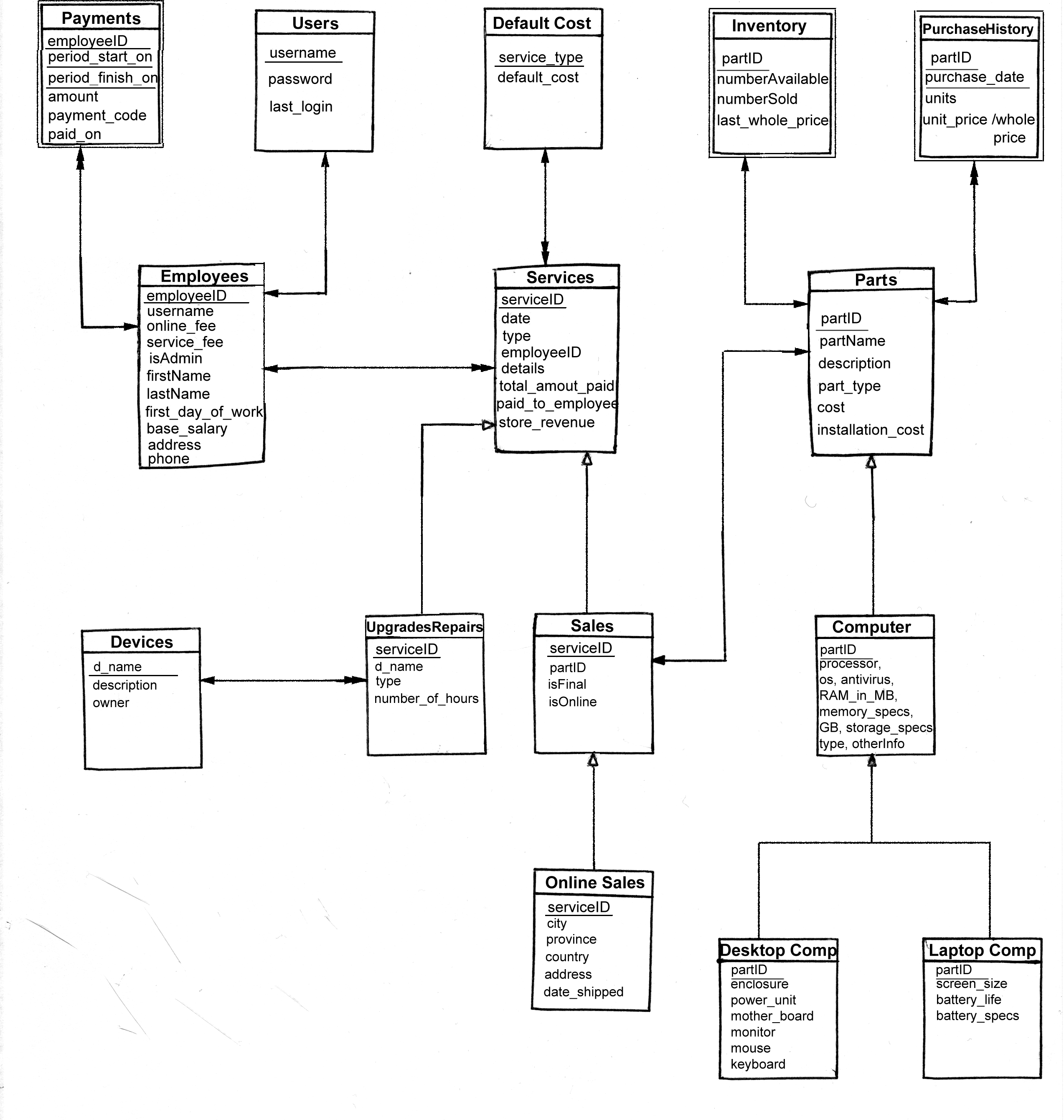
Computer(**partID**, processor, os, antivirus, RAM\_in\_MB, memory\_specs, GB, storage\_specs, type, otherInfo) where partID references Parts on DELETE CASCADE.

DesktopComputer(**partID**, enclosure, power\_unit, mother\_board, monitor, mouse, keyboard); where partID references Computer on DELETE CASCADE.

LaptopComputer(**partID**, screen\_size, battery\_life, battery\_specs); where partID references Computer on DELETE CASCADE.

In the previous schemas, primary keys are always underlined, and **foreign keys** are shown in bold letters. All referential integrity constraints are also shown.

The final Table Relationship diagram from the previous database schema is shown below. Here, the cardinality of the relationships is shown by single arrows(one) or double arrows (many).



In our case, is very easy to see that every relationship in our final schema is in First Normal Form (**1NF**) because we don’t have any non-atomic values in any tables. Since all our relationships have no non-atomic values, all our relationships are 1NF.

In the same way, all our relationships are also in Second Normal Formal (**2NF**) because for all relationships, all non-prime attributes are fully functionally dependent on the relation key. For example, both description and owner are fully functionally dependent on d\_name in the Devices relationship.

Our database schema is also in Third Normal Form (**3NF**). To see this, we first deal with the simpler relationships that have a single Prime Attribute and only one (the same) Candidate Key, for all these relationships, we only have functional dependencies of the form X->A where X is precisely the key. Since no other functional dependencies are present, by definition, all those relationships are en 3NF:

**DefaultCost**(service\_type, default\_cost); **Users**(username, password, last\_login); **Services**(serviceID, type, employeeID, date, details, total\_amount\_paid, paid\_to\_employee, store\_revenue); **Inventory**(partID, numberAvailable, numberSold, last\_whole\_price); **Devices**(d\_name, description, owner); **UpgradesRepairs**(serviceID, d\_name, type, number\_of\_hours); **Sales**(serviceID, partID, isFinal, isOnline); **OnlineSales**(serviceID, city, province, country, address, date\_shipped); **Computer**(partID, processor, os, antivirus, RAM\_in\_MB, memory\_specs, GB, storage\_specs, type, otherInfo); **DesktopComputer**(partID, enclosure, power\_unit, mother\_board, monitor, mouse, keyboard); **LaptopComputer**(partID, screen\_size, battery\_life, battery\_specs);

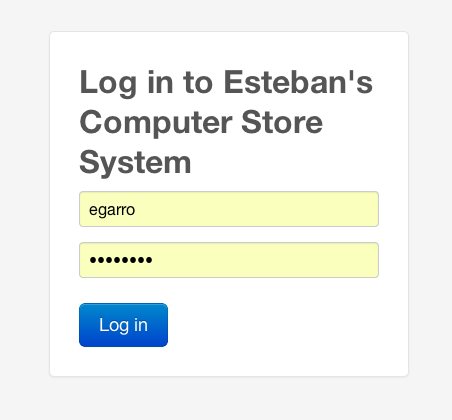
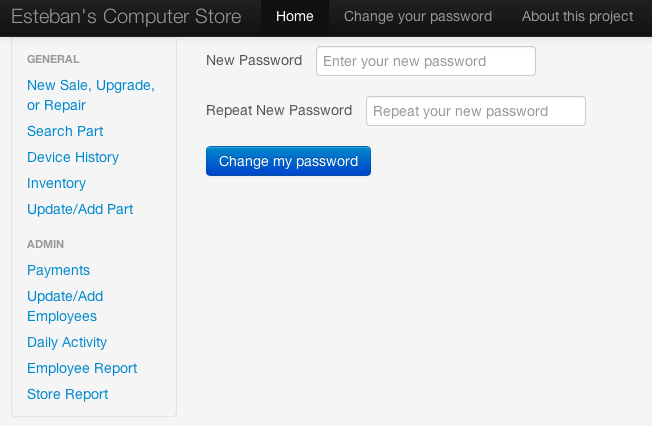
For the **Employees** and **Parts** relationships, we have two prime attributes and the same two Candidate Keys. In the case of the Employee relationship, those keys are A=employeeID and B=username, and for the Parts relationship, A=partID and B=partName.

Since either A or B can be used as a primary key, and all functional dependencies on these two tables are of the form A->X or B->X, by definition, those relationships are also in 3NF form.

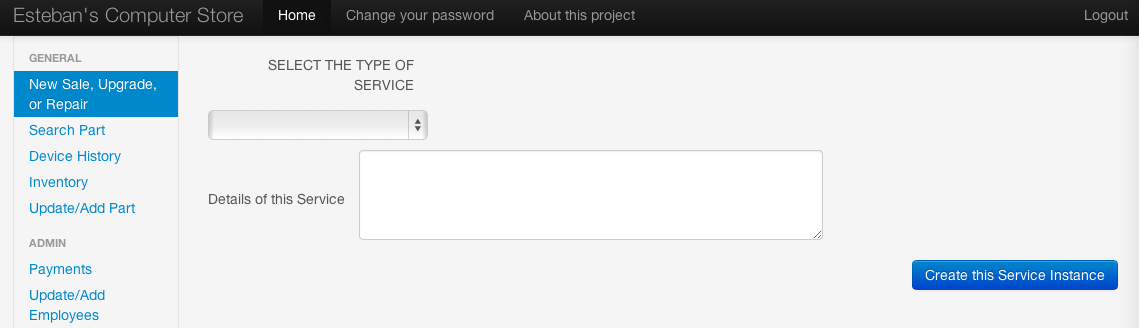
Finally, we have two relationships, **Payments** and **PurchaseHistory** that have composite primary keys. If Payments={ESFACP}, we only have functional dependencies ESF->A, ESF->C, ESF->P, and if PurchaseHistory={IDUP} all our functional dependencies are ID->U, ID->P. In both cases, since the left hand side of all functional dependencies is a Key, these relationships are also in 3NF!

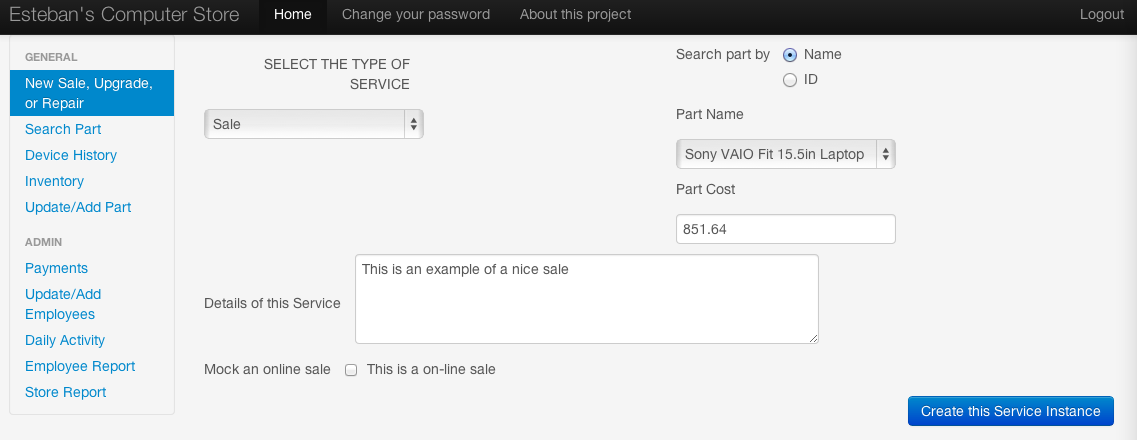
**THE USER INTERFACE/SAMPLE SESSION**

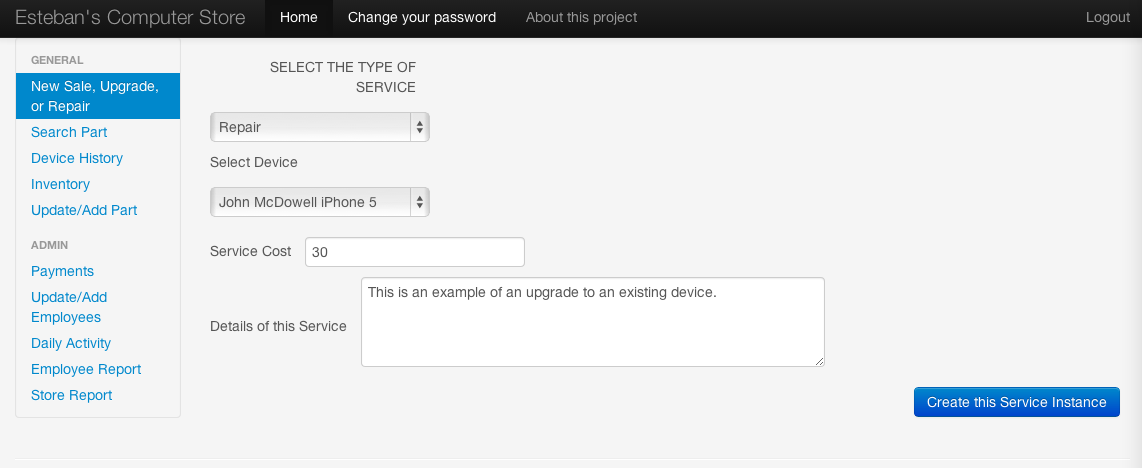
A normal CSRSS session starts by logging in to the system. When logging in, the server starts a session and maintains a $\_SESSION array so that is not possible to see any internal pages unless logged-in to the system. The user can change their password whenever they want by clicking on the appropriate link on the top bar:



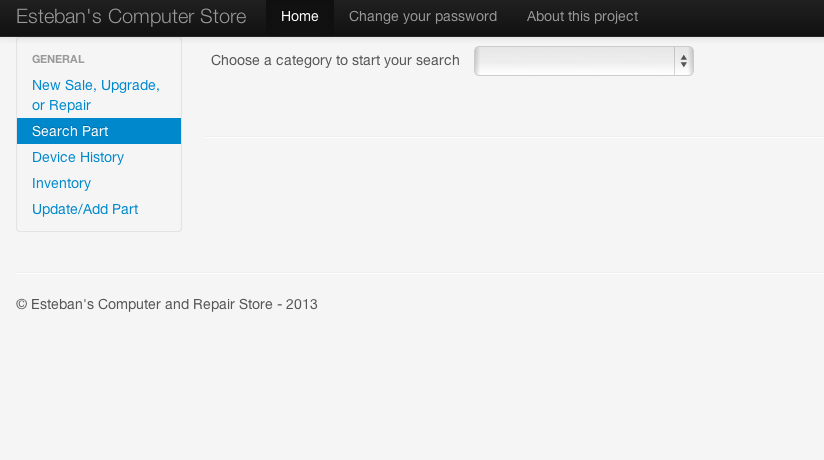
Once logged in, the user sees the following main interface, where they can start by selecting a type of service for processing a new Sale, Upgrade or Repair:







Here we can see that depending on what the user selects first, the rest of the user interface updates accordingly. This way, we only present the users with the next possible option to choose from making it less confusing for people to use.

As you can see on the left-hand-side menu, there are two sections: GENERAL and ADMIN. The ADMIN part of the menu is only visible when a user with administrative privileges logs in: we of course don’t want anybody seeing someone else’s Employee Report or creating/deleting random users.

A normal user

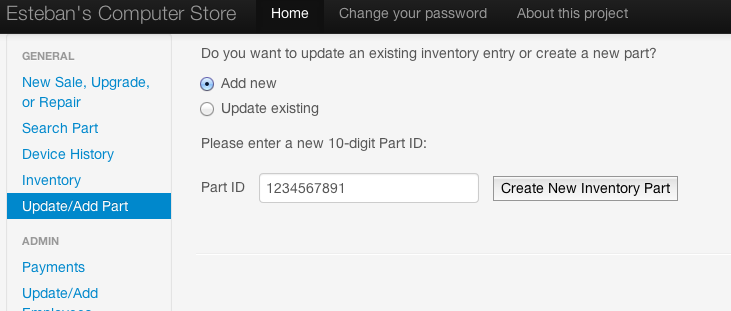
logs in:

When clicking on any item on the left-menu, a new, self-explanatory, auto-adjustable UI appears to help the user find its way to the next step. One item on the left menu corresponds to exactly one db-functionality or to a group of related db-functions, such as Sale, Upgrade and Repair.

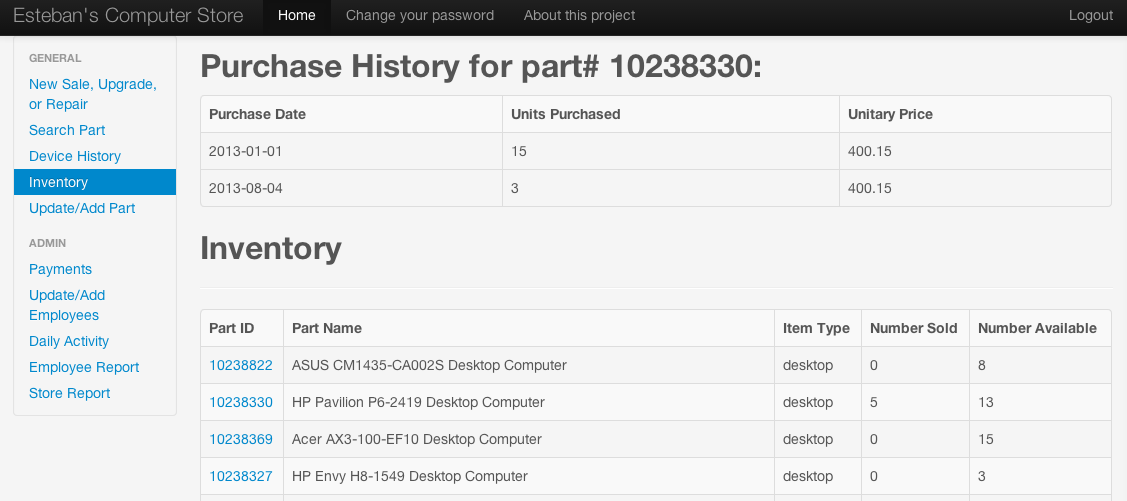
To search for a Part, just select out of the 4 possible Part Categories: Hardware, Software, Desktop or Laptop. Then you’ll be asked whether you like to search by Part Name or Part ID, pick one, get a list of parts, pick one and observe the results:



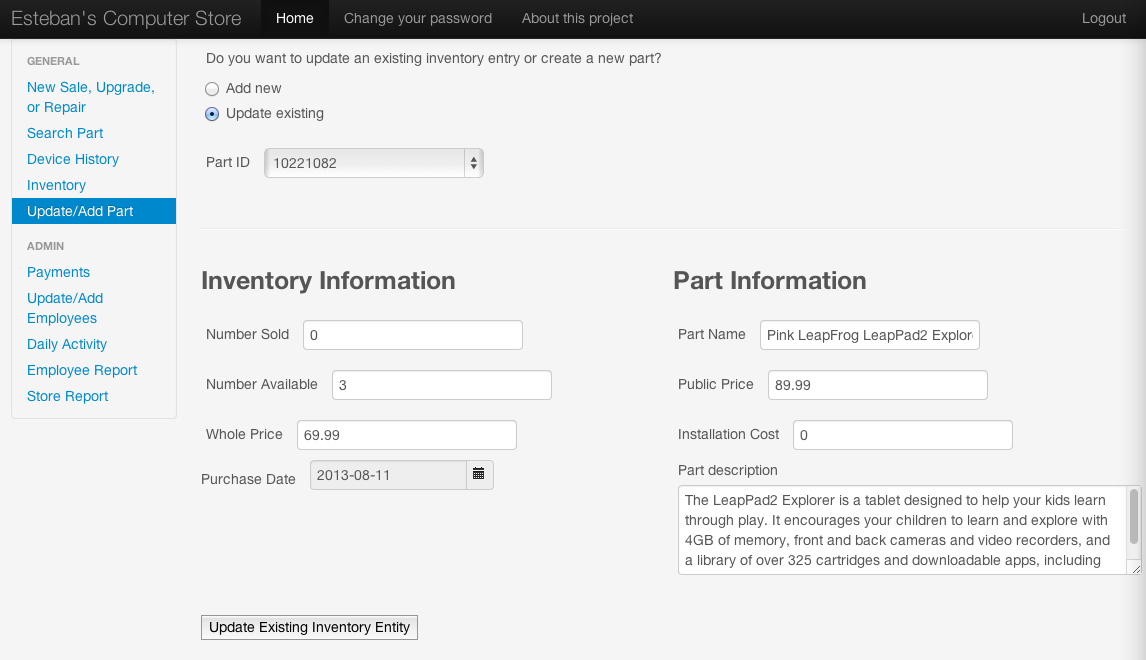
All other sections are self-explanatory. All inventory parts have 10 digits, and all employee IDs have 5 digits. When creating a new employee or adding a new part to the inventory, Javascript checks that this condition is met before it lets you continue. The Create New Inventory Part button is disabled until you input 10 digits:



To view the Purchase History for certain item, simply go to the inventory and click on the corresponding partID you wish to see. The Purchase History for this item is revealed on top of the page:



The creation of new Purchase History instances is fully automated and can be done through the Update/Add Part menu option. Once there, simply modify the Number Available field and click on Update Existing Inventory Entity. The code is smart enough to notice that now you have more available items than you had before and understands this as a new purchase for that part with the whole\_price you indicated here.



Payments for all commissions earned by an employee follow this logic: