## Website

|  |  |  |
| --- | --- | --- |
| Testing data: | Tony Montana | Jennifer Burges |
| Email | [tm@gmail.com](mailto:tm@gmail.com) | [jb@gmail.com](mailto:jb@gmail.com) |
| Password | tnmntna | jnfr |
| Ticket | Yes | No |
| Tent | None | None |
| Activities | X2 reserved | X1 reserved |
| Items | X3 loaned | X0 loaned |

### Technologies used

Our team used different IDEs, with the mainly used being:

* Atom
* Netbeans
* Sublime
* Visual Studio Code

Our web application uses the following technologies

* PHP for the back-end of the website
* MySQL for the database
* CSS and JavaScript for the front-end

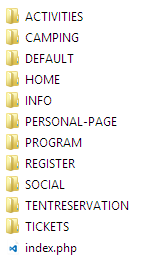
We also used some libraries and frameworks like:

* jQuery - We used because of the easier way of implementing animations and interacting with the DOM tree
* Bootstrap – Mainly we made use of the grid that the framework gives
* Sweet Alerts - Used for displaying nice looking messages (examples of use at Personal Page, Registration/Login page, and others)  
  http://t4t5.github.io/sweetalert/
* Alertifyjs – Used for displaying nice looking log notifications. Used at the Activities page and also in the framework to display the terms and conditions.  
  http://alertifyjs.com/

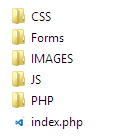
### Website file system organization

Our group chose this structure because it best suited the purpose of the application.  
The root folder has the following structure:

* It is divided into 11 subfolders and an ‘index’ file.
* The starting point of the application is the file ‘index.php’.
* The website directoroty is organised in such a way that the files can be easily accessed by the main file.
* Every subfolder contains data about one particular page.
* Every page folder which also contain a starting index file and subfolders with folder for every particular thing? that is used.
* The folder called ‘DEFAULT’ serves to store files that are shared between all the pages. There can be found images, header, footer, some default styles and scripts, libraries fonts and etc



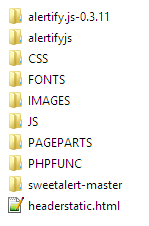
Each page folder is then separated into subfolders for the page specific media, scripts, PHP code and other data.

This is an example taken from the activities page directory, other separate pages follow the same/similar structure:

* CSS folder stores all the page specific stylings
* Forms – stores all the additional forms and templates that are loaded in the page by AJAX during runtime
* Images – stores the media and images that are used in this page
* JS – stores the JavaScript code for the page
* PHP – stores the PHP codes for the page to process the server queries and interaction

The “DEFAULT” directory is used to store shared media, codes, frameworks and styles. A lot of references, requires and reuses in the pages refer to this folder’s contents.

* It has 9 subfolders
* Aertify.js-0.3.11 – is a folder that stores the framework



* Alertifyjs – is a folder storing a slightly different version of the Alertifyjs framework
* CSS – a directory storing the shared styles for every page  
  e.g: footer.css, integral.css
* Fonts – a directory that stores the fonts that are used on the website. These are Bauhaus93 and Odin Rounded (<http://www.1001fonts.com/odin-rounded-font.html>)
* Images – a directory for the common images. It includes the images that are used in the footer, logos, social network logos and others.
* Pageparts – a directory that stores the reused/shared parts of the pages: Footer.php, FooterLinks.php (which are nested inside of the Footer.php), FooterLinksHome.php – the part that is used in the home page (root/index.php), since it’s located on a higher level than other pages
* PHPFUNC – a directory that stores the serverside codes that are used at every page, e.g Logout.php (since the logout is accessible in the header navbar at every page)
* Sweetalert-master

### Functionality

#### Overview

For every page, except the ‘Home page’, we use the same menu and footer, which are loaded with PHP from the ‘DEFAULT’ folder. Every page shares the styling of those elements from the DEFAULT/CSS directory as well.

The header’s last link – account does not only serve as a redirect mechanism to either login/register page or personal page, but also provides the way to log out on the mouse hover.  
The footer functionality as follows: *“news & social”* redirects to a respective page, *“contact us”* opens a new window for a default client’s mailing provider with a brief template for an email to [jobsdoneuniversal@gmail.com](mailto:jobsdoneuniversal@gmail.com) (this is a valid email address), *“terms and conditions”* prompts with a message box with the terms and conditions – made using alertifyjs framework. This functionality is available at every page

#### JS and PHP

What is specific is that all our PHP scripts that are invoked on some button click, are executed via AJAX which means that to any button of that kind(e.g. register, login) is attached a click event which calls a JavaScript file who executes the PHP script, processes it and retunes a relevant response asynchronously.

While developing the codes for the website (both JS and PHP) the quality of them and their patterns were changed gradually as more good practice conventions were learned. The PHP code is very far from perfect. It uses pure PHP, and does not implement any frameworks or additional libraries, which makes it a) overly complicated; b) the separation of logic and visualization is not possible using such way. In the AJAX-PHP interaction pattern the server responses are categorized by the unique messages that are echoed from in the PHP code upon the termination of a certain method. However, for all the server and MySQL database interaction the object oriented PDO pattern was used.

E.G from the login js-php interaction at the REGISTER/LOGIN page:

|  |  |
| --- | --- |
| *// from the js/auth/ajax-login.js*  *…*  success: function(response) {  if (response.**includes('logged'**)) {  …..  } else {  ……  }  } | *// from the php/login.php*  *….*  $stmt->execute(array(":email"=>$user\_email));  $row = $stmt->fetch(PDO::FETCH\_ASSOC);  // $count = $stmt->rowCount();  if($row['PASSWORD']==$password && $password !="" ){  **echo "ok-you logged in"**.print\_r ($row);  $\_SESSION["USER\_ID"] = $row['USER\_ID'];  }  else{  echo "email or password does not exist.";  }  … |

PHP code was used to display the user/condition relevant data on the webpages. It was used to populate the contents of the Personal Page modules, create the forms to reserve activities, show the user reserved activities and so on. Example code provided below:

*// from the ACTIVITIES page index.php*<?php

for ($i = 0; $i < count($activities); $i++) {  
 echo '  
 <li class="act\_item">  
 …   
 <!--<form action="php/processreservation.php" method="post">-->

<form class="formreservation">

<div class="activity\_info">

<p class="text actname">' . $activities[$i] . /\* the name 0 7 \*/'</p>

…  
 Places available: ' . $activities[$i+=1] . /\* places 4 \*/'

…  
 <input type="checkbox" class="chkbox" required   
 name="reserve" checked="true" value="yes">  
 <label class="chkbox"> I want to reserve </label>

…

<button type="submit" class="btn text resbtntext">RESERVE</button>

…  
 </li>';

} ?>

As for the JS codes, a lot of JQuery code was written to interact with the DOM. It made it easy to animate the webpages and to work with the ‘static’ and user-input data. E.G: a lot of JQuery code was used in the TENT RESERVATION page.

#### APIs

For this website the following APIs were used:

* Twitter API   
  In the ‘SOCIAL and NEWS’ page reached from the footer, the twitter provided APIs are used both to present a twitter news feed and also to share the page via a ‘tweet’ by pressing the according button.
* Facebook API  
  In the ‘SOCIAL and NEWS’ page the ‘share’ button is implemented to post directly to the Facebook timeline.
* Google+ API  
  In the ‘SOCIAL and NEWS’ page the ‘share’ button is implemented to post directly to the Google+ timeline.
* Google Maps API  
  In the INFO -> LOCATION page the Google maps API was used to show display the event place and facilitate locating the venue by providing an easy way to set a route.

### Page description

#### Activieties page

This page is used to view the activities that can be attended during the festival. The registered users can book a place at the activity and cancel it at any moment. This ‘web application’ replaces the activity reservation windows application that was declared in the project plan. This page is more reasonable and user-friendly than a C# windows application with the same functionality.

The PHP and JavaScript are very faulty and is not implemented to run effectively.   
The code specifications on this page:

* JQuery to animate the activity boxes
* Ajax – to – PHP communication pattern to process reservations and cancelations
* On-page PHP to retrieve the activities from the database
* On-page PHP to display the user reserved activities, if he is registered, or a message to login if he is not.
* Alerifyjs notifications to display logs in the left lower corner of the screen about successful/unsuccessful operations.

#### Camping page

The page consist mostly of static information and a dynamic PHP script for posting a link to the tent reservation for several conditions: a) more than one camping spot available; b) only one spot available; c) no places are left. This code places an image-link by pressing which the visitor is redirected to the tent-reservation page.

a) b) c) 

#### Home and Info pages

Consist of static information and information about the event.   
The home page is specific as at first, a large banner appears at the top part of the window and the navigation bar is below it. Once the page is scrolled down, the navigation bar is lifted to the top of the window and fixed. For this, the JQuery code was created that changes the class of a navbar so it would fit the appropriate styling. Below is the snippet of the script from the index.php.

var logo = document.getElementById('logo');  
 var potato = document.getElementById("potato");  
 $(document).on("scroll", function() {  
 if ($(this).scrollTop() > 580) {  
 $("#header").removeClass("headernotscrolled");  
 …  
 logo.src = "DEFAULT/IMAGES/logohead.png";  
 potato.style.display = "inline";

} else {  
 $("#header").removeClass("scrolled");  
 ….  
 potato.style.display = "none";  
 }  
 });  
 </script>

#### Personal page

There the user can find all his personal information he has provided at the register page, information about his camping status, finances and etc. It consists of dynamic and static parts. The information is separated into separate modules such as “*General*” – for the general user data such as name, address, email; *“Balance”* – used to manage the visitors balance; *“Bracelet”* – used for viewing the visitor’s ticket payment and bracelet information; *“Your Tipi”* – shows the data (number and the tent leader) about the camping spot for which the visitor is registered; *“Items”* – displaying the information about the items which the visitor had loaned; “Activities” – showing the list of the activities.



The buttons for hiding and showing are implemented with jQuery and some easy showing and hiding html elements, manipulating styles and loading content with AJAX. Every button has an attached click event, which executes a corresponding script.

Every time the user accesses the page, the system executes few PHP scripts. Via these PHP scripts are made queries to the database and the information about the particular user is retrieved. Whenever the user wants to update his balance, he can go to the balance field and fill the desired amount to be added to his account. Considering the PHP scripts they are still not very optimized in terms of security and quality of the code (a lot of repetitive code).

#### Program page

It consists of static information about the program of the event. There you can see a few tabs whose content is dynamically loaded with jQuery and AJAX. It also features some animations.

#### Register page

It has 2 tabs one of them is for registering the other is for logging in. The content is loaded dynamically with the help of jQuery and Ajax. On the page, there is also some HTML5 and JavaScript validation of inputs, e.g. the email entered is forced to be the same as the confirm email (same applies for the password).

If we look at the PHP script and the way the whole registering and logging in works, we will see as mentioned in the overview that the scripts are executed via AJAX code that is executed by clicking on the corresponding button. Considering the PHP scripts, they are still not very optimized in terms of security and the code quality (a lot of repetitive code).

#### Social page

There the users can find a twitter feed and the share buttons for Facebook, Twitter and Google+. Everything is implemented by using the respective APIs.

#### Tent reservation page

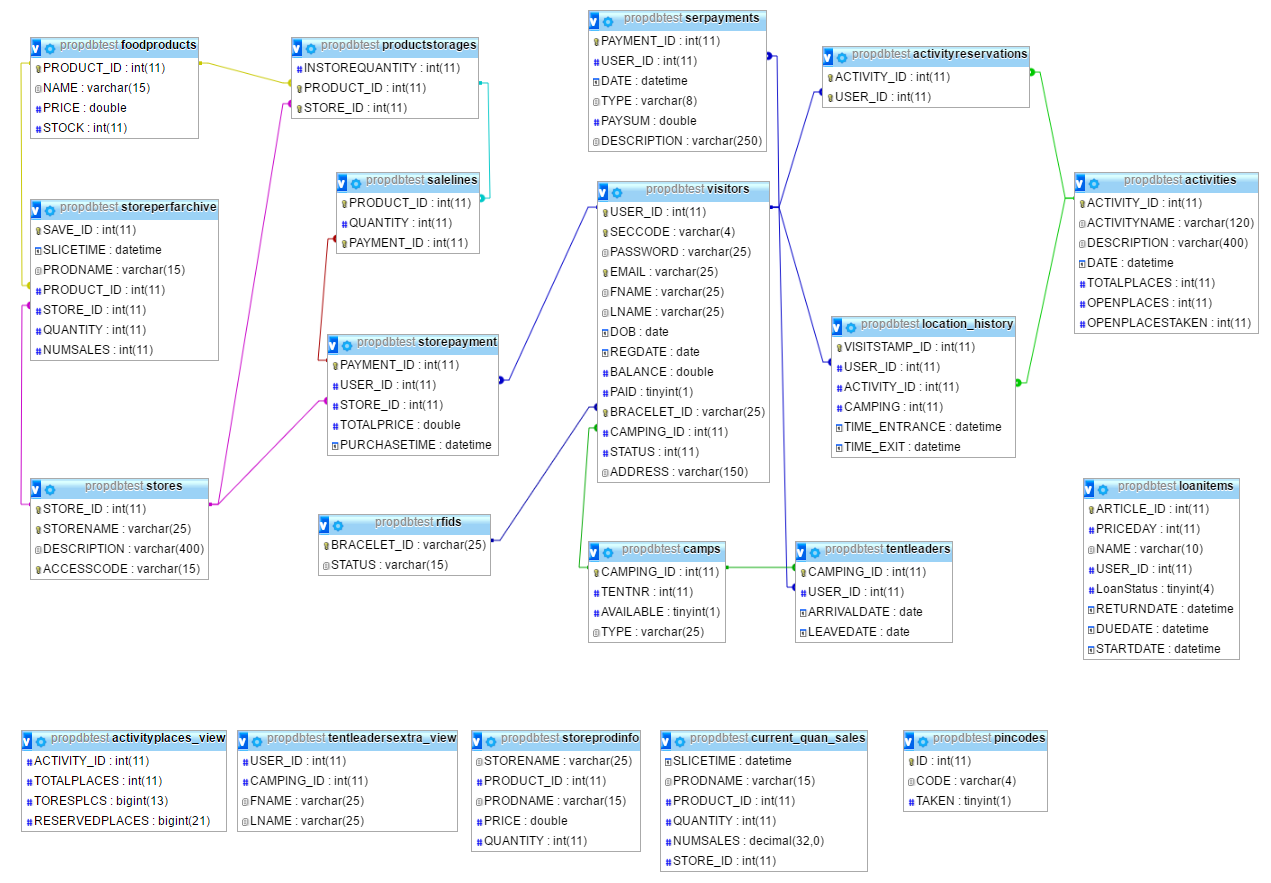
At this page the camping spots can be reserved for a group of up to 6 people. One user is required to pay the whole sum for the camping from his balance and also will be the responsible person for the camping spot. He has to enter his credentials once again to ensure the secure user authentication.   
  
The codes that are used to generate the input fields are written using JQuery and pure Javascript. The server interaction is done via the AJAX and PHP.

Tickets page   
This page is used to purchase the tickets for the events – this is a required condition for obtaining the bracelet and reserving a tent at the entrance. The user is redirected to this page right after the registration.

There are three types of tickets, which generally differ only in price and have do not affect the general user experience in any way (Our team supposes that is how all the festivals work). To purchase a ticket the visitor has to enter his credentials to ensure his validity. For the purchase a new window is loaded in the center of the page.

\*For further details refer to the code and comments.  
\*\*The website visual design, sitemap, and processes are discussed in the according appendix document. This is the initial concept so the real user experience can differ.

## C:\Users\Dmitry\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Screenshot (137).pngDATABASE



Design

The ERD diagram above depicts a complete design of the database.

It features 16 tables, 5 views, and 1 event. Those are required for different interactions with the database via the applications and the website. The 5 views were created in order to facilitate the data retrieval from the database. 1 event that is not present in the ERD diagram is called SalesSlicerEvent and is executed every hour in order to save the ‘slices’ of the quantity and sales of the products per each shop.

### Tables

* Activities.  
  The table to keep the data about the events. Includes the name, id, date, description, number of places (of three types – open/take/total)
* ActivityReservations.  
  This is an intersection table to store the data about the visitor’s reserved activities.
* Camps.  
  This is a table to store the data about the camps. It has the attributes to store the availability of a camp, its ID, number and type (which is not used for anything now).
* Foodproducts.  
  This is a table to store the data about the food products that are sold at the stores. It serves as a general warehouse – from there all the products are divided between the shops.
* Loanitems.  
  This is a table to have all the data about the items that can be loaned during the event. This table is not connected with a relation in the ERD, but in fact the USER\_ID attribute references to the USER\_ID of the visitors. If the item is loaned the USER\_ID is retrieved from the visitor and if not it is set to 0. The dates signify the dates when the item was returned, when it is due to be returned and when it has been taken. – when a new visitor starts using the item the data is overwritten.
* Location\_History.   
  This table is used to store the location of the visitors – it was designed to secure the entrances and exits to the events and camps from fraud. It has the attributes: activity\_id and camping, which are set former to the activity\_id of the one the visitor is entering and camping is set to 1 if the visitor is entering the camping. Potentially this table can also be used to review the statistical information about the popularity of the activities and density of the camping.
* Pincodes.   
  This table was created to store the unique 4 char that were supposed to be used as an additional mean to identify the visitor. However, at the current moment this table is not used anyhow.
* ProductStorages.  
  This association table is used to see the quantity of the products present in the stores.
* RFIDS.  
  This data keeps the data about the bracelets – IDs and statuses. A range constraint applies to the statuses: they can be any of only three following values: STAND\_BY, ACTIVE, DEACTIVATED.
* SaleLines.  
  This is an association table used to keep track of the products that were bought at one payment. Thus the foreign key Payment\_ID references to the StorePayment table. The data about the products is obtained from the reference to the FoodProducts table.  
  This follows a concept of a following data design:

A person could have made several payments in different stores for different sets of products

A sale-line stores a single product and its quantity purchased that was included in the payment

A payment can have many products in one check

* SerPayments.  
  This table stores the information about all “service” payments performed by the users. The payments for the tickets and tents are considered to be service payments.
* StorePayment.  
  \*This table is the only table that does not follow the general naming convention due to a typo on an early stage and afterwards too many code was written using the faulty name.  
  This association table stores the payments in the stores of the visitors. It has a Payment\_ID, PurchaseTime – which shows the time when the purchase was made –, a Store\_ID to store the reference to a store where the payment was made, a TotalPrice – to store the complete amount that was paid – and a User\_ID to store the reference to which Visitor has performed this payment.
* StorePerfArchive.  
  \*This table has a different meaning and thus doesn’t follow the naming convention.  
  This table is populated via the SalesSlicerEvent event every hour. It stores the number of sales of a certain product, its name, id, current (at the time of insert) quantity in a store for every store, and also stores a time when the save (‘slice’) was done and a store\_id to keep track of the stores.
* Stores.  
  This table keeps the store data, with its name, ID, description and accesscode. The former two attributes are not used in any applications, but are valuable for further development in case of continuing production. The access code can be used to restrict the use of certain applications, and the description - to display the information on the website for the visitors.
* Tentleaders.  
  This table keeps the data about the camping reservations – the name is slightly updated and doesn’t comply correctly with the current purpose. It stores the data about the user who had paid for the tent group – thus the responsible person (chief), camping\_id, leave date and arrival date – which signify for which dates the camp is going to be reserved.
* Visitors.  
  *“The table of all tables”*  
  This table stores all the data about a certain visitor – all his/her general information provided at the registration, the camping and bracelet associated with the visitor. The status can be either 1/0 (Tinybit or a Boolean) and represents whether a visitor has entered the event or not. The paid attribute shows whether the user has purchased a ticket. Passwords now are stored in an open way, which is a very bad practice, but we decided to leave it this way for the time being.

### Views

* ActivityPlaces\_view.  
  This view is used to calculate the places for the events – the total number of places/2 is the number of places available to be reserved. Reserved places – the number of visitors registered for an event via the website. The number of open places at the moment of the event start = totalplaces – reservedplaces.  
  This view is defined as:   
    
  SELECT `a`.`ACTIVITY\_ID` AS `ACTIVITY\_ID`, `a`.`TOTALPLACES` AS `TOTALPLACES`, floor((`a`.`TOTALPLACES` / 2)) AS `TORESPLCS`, count(`ar`.`USER\_ID`) AS `RESERVEDPLACES`   
  FROM (`propdbtest`.`activities` `a`   
   JOIN `propdbtest`.`activityreservations` `ar`   
   on((`a`.`ACTIVITY\_ID` = `ar`.`ACTIVITY\_ID`)))   
  GROUP BY `a`.`ACTIVITY\_ID`
* Current\_quan\_view.  
  This view is used in pair with a storeperfarchive and follows the same structure but instead of being populate each hour, stores the data of a current moment. This is used in the statistics to compare historic data and the current.  
  It is defined as:   
    
  SELECT now() AS `SLICETIME`,`fp`.`NAME` AS `PRODNAME`,`ps`.`PRODUCT\_ID` AS `PRODUCT\_ID`,`ps`.`INSTOREQUANTITY` AS `QUANTITY`,sum(`sl`.`QUANTITY`) AS `NUMSALES`,`ps`.`STORE\_ID` AS `STORE\_ID`   
  FROM (((`propdbtest`.`foodproducts` `fp`   
   join `propdbtest`.`productstorages` `ps`   
   on((`fp`.`PRODUCT\_ID` = `ps`.`PRODUCT\_ID`)))   
   left join `propdbtest`.`storepayment` `sp`   
   on((`ps`.`STORE\_ID` = `sp`.`STORE\_ID`)))   
   join `propdbtest`.`salelines` `sl`   
   on(((`sp`.`PAYMENT\_ID` = `sl`.`PAYMENT\_ID`)   
   and (`sl`.`PRODUCT\_ID` = `ps`.`PRODUCT\_ID`))))   
  group by `ps`.`PRODUCT\_ID`,`ps`.`STORE\_ID`;
* Storeprodinfo.  
  This view is used to facilitate the access to the information about the products currently located in the store.  
  It is defined as:  
    
  SELECT `st`.`STORENAME` AS `STORENAME`,`ps`.`PRODUCT\_ID` AS `PRODUCT\_ID`,`fp`.`NAME` AS `PRODNAME`,`fp`.`PRICE` AS `PRICE`,`ps`.`INSTOREQUANTITY` AS `QUANTITY`   
  FROM ((`propdbtest`.`productstorages` `ps`   
   join `propdbtest`.`foodproducts` `fp`   
   on((`ps`.`PRODUCT\_ID` = `fp`.`PRODUCT\_ID`)))   
   join `propdbtest`.`stores` `st`   
   on((`st`.`STORE\_ID` = `ps`.`STORE\_ID`)))   
  order by `fp`.`NAME`
* Tentleadersextra\_view.  
  This view is used to facilitate the access to the information about the tent leaders. It retrieves the visitor data based on the user\_id of a leader.  
  It is defined as:   
    
  SELECT `t`.`USER\_ID` AS `USER\_ID`,`t`.`CAMPING\_ID` AS `CAMPING\_ID`,`v`.`FNAME` AS `FNAME`,`v`.`LNAME` AS `LNAME`   
  FROM (`propdbtest`.`tentleaders` `t`   
   join `propdbtest`.`visitors` `v`   
   on((`t`.`USER\_ID` = `v`.`USER\_ID`)));

### Events

* SalesSlicerEvent.  
  This event is used to make hourly ‘saves’ or ‘snapshots’ or ‘slices’ of the product statistical data per shop. It inserts the data into the StorePerfArchive table on every interval. This table is described above.  
  The event’s function is defined as:   
    
  INSERT INTO storeperfarchive (SLICETIME, PRODNAME, PRODUCT\_ID, QUANTITY, NUMSALES, STORE\_ID) SELECT NOW(), fp.NAME, ps.PRODUCT\_ID, ps.INSTOREQUANTITY, SUM(sl.QUANTITY) AS NUMSOLD, ps.STORE\_ID   
   FROM foodproducts fp  
   join productstorages ps   
   on fp.PRODUCT\_ID = ps.PRODUCT\_ID  
   left outer join storepayment sp   
   on ps.STORE\_ID=sp.STORE\_ID   
   join salelines sl   
   on sp.PAYMENT\_ID = sl.PAYMENT\_ID and sl.PRODUCT\_ID = ps.PRODUCT\_ID   
   GROUP BY ps.PRODUCT\_ID, ps.STORE\_ID  
    
  \*The event can not be run in the Athena due to the lack of rights to set the event scheduler on.

This database design proved itself usable and informative enough despite its drawbacks and imperfections. The final design differs a lot from the one declared initially. It does not implement some concepts of the initial idea, but improved a lot in other spheres.

## Applications

This section describes the application covering their details and functionality overviews. It covers it in a more or less general way pointing out the most relevant features  
  
The following applies to all the applications:

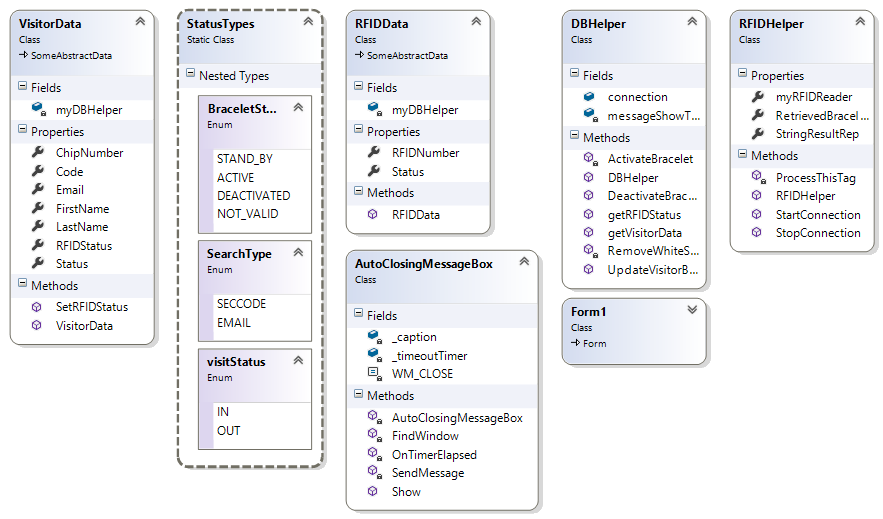
* The deployment is not performed
* The database connection settings are stored in a respective DLL – DBConnectionDLL – and are shared by the applications.
* For the additional information and details see the source codes and comments.

### Bracelet management

#### Overview

This application is to be used in the support/bracelet management booths. It's main functionality is to manage the bracelets: assign, issue, deactivate and change the bracelets for certain visitors. It is built upon two communication channels: PC-MySQL database and PC-RFID reader.   
This application should be used to issue the bracelets for the new arriving visitors. Example testing data: email – [abc@abc.abc](mailto:abc@abc.abc) / [jf@jf.jf](mailto:jf@jf.jf) / seccode - KARL

Class diagram



* Visitor Info:   
  Visitor information retrieved for the purposes of this application complies with the database visitor table and uses the following attributes: EMAIL, FNAME, LNAME, SECCODE, BRACELET\_ID, STATUS.  
  The information is obtained by searching for a visitor via email/secret code input. The following method is used to retrieve the respective data:   
    
   public VisitorData getVisitorData(StatusTypes.SearchType searchAttribute, string whereClauseValue)   
  In the body of this method the following query is executed:   
   String sql = "SELECT EMAIL, FNAME, LNAME, SECCODE, BRACELET\_ID, STATUS FROM VISITORS WHERE UPPER(" + whereClauseAttribute + ") =" + " UPPER(\"" + whereClauseValue + "\")";
* RFID data:   
  Complies with the database structure as well. Obtained from the database via the visitor information or from the RFID reader.
* To interact with the RFID reader and the Database the according RFID helper and DB helper classes were used.

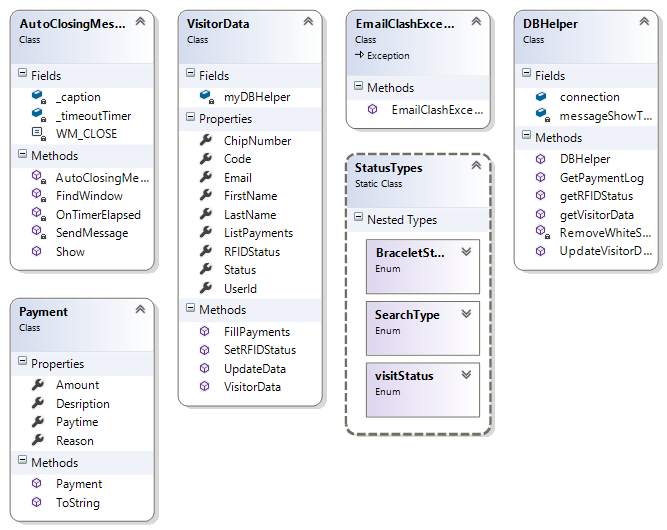
### Visitor support

#### Overview

This application is based on the bracelet management application discussed earlier and is a modified version of the latter. It has less functions than the Bracelet Management, but does implement some of the same methods and also additional functionality used in this application.

This application offers the methods to update the visitor information – email, first name and last name; also it makes it possible to see the overview of all the payments a certain visitor had done (service and store including).

Class diagram



* Visitor Info:   
  Same as the one in the Bracelet Management.
* DBhelper:  
  This class has a method to update the visitor data – it is performed on a click of a button ‘Update data’ and writes the newly input data from the textboxes back to the database. However, before the changes are done it checks for the ‘clashing’ email. Below is the code snipped for the email clash checking:  
    
  “...   
   command.CommandText = "SELECT USER\_ID FROM VISITORS WHERE EMAIL = '" + email + "';";

MySqlDataReader reader = command.ExecuteReader();

bool emailClash = false;

while (reader.Read())

{

if (userid != (int)reader["USER\_ID"])

{

emailClash = true;

}  
 }...”  
  
The whole method   
public bool UpdateVisitorData(int userid, string email, string fn, string ln)  
uses transactions to accomplish the task of not writing the incorrect data and preventing other faults.  
  
The payment overview is retrieved by the following query:

string sql = "SELECT DATE as PAYTIME, TYPE as REASON, PAYSUM as AMOUNT, DESCRIPTION" +

" from serpayments WHERE USER\_ID = " + userId +

" UNION " +

"SELECT stp.PURCHASETIME as PAYTIME, s.STORENAME as REASON, stp.TOTALPRICE as SPENT,'Payment in the store' " +

" from storepayment stp JOIN " +

" stores s " +

" on stp.STORE\_ID = s.store\_id " +

" WHERE USER\_ID = " + userId +

" order by PAYTIME ;";  
It uses the union in order to merge the data from two different tables into one.

* File export:   
  This app has a functionality to save the log of payments generated for a visitor in a separate .txt file. This file then can be sent via email on request etc.

### Statistics app *“StatsApp”*

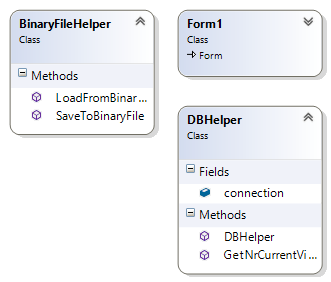
#### Overview

This application is used to keep track of different statistical information that can be retrieved from the database. The different data is managed according to certain patterns. Generally, it has three modules: Visitors, Products and Finances, which present the overviews of the data relevant to the respective modules.

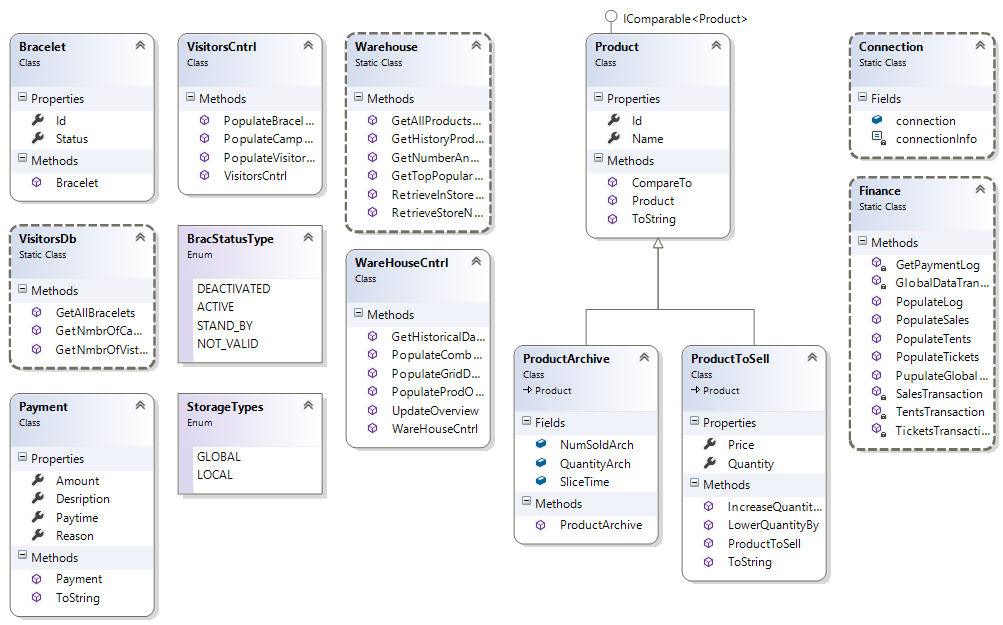
Each module is presented by a separate tab inside of one form. To gain the high level of logic separation inside of the same solution a separate project was created called Modules, which was meant to store the Controllers of the respective modules.   
  
In this application our team had tried to reach the closest to the MVC software architecture concept.

As it is seen below the class diagram of the classes in StatsApp.csproj is relatively small and counts only three class handling files: BinaryFileHelper, Form1, DBHelper. Some of the methods, handling the data, that are present inside of these classes were used at an early stage of the development and thus stayed untouched.

Class diagram for the ‘VIEW’ project.



In contrast to the class diagram above the class diagram of the classes of Modules.csproj is considerably larger.

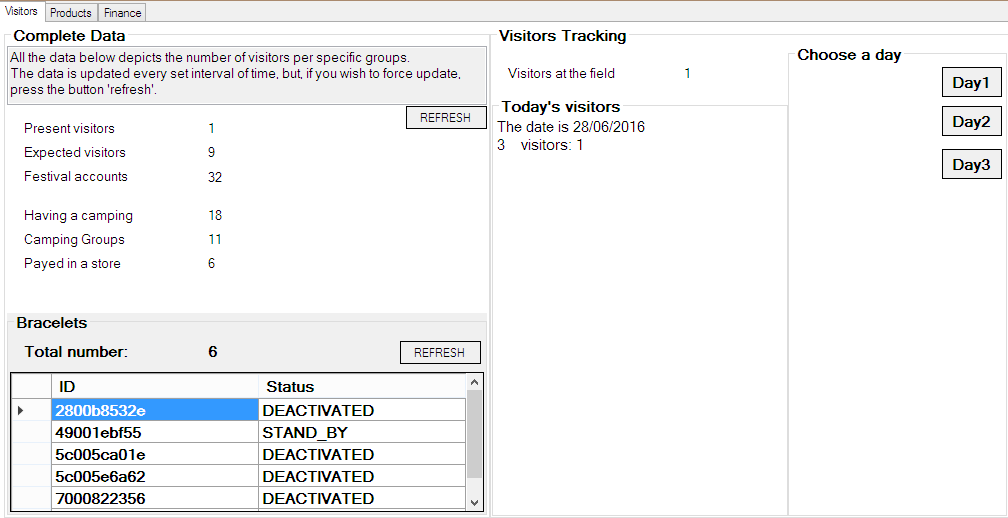


The methods of these classes mostly follow the following assumptions and concepts:

1. Separate all the data manipulation from the form
2. Populate the data to the elements inside of the modules’ methods – thus passing the components as parameters
3. Make the methods to be mostly static

#### Visitors module

This module was the initial module and the first developments were done a long time before the latest updates and even before the whole ‘division of labors’ concept was established. Therefore it partially relies on the methods featured in the StatsApp.csproj and partially – on the final concept. Below is the screenshot of an application at runtime.



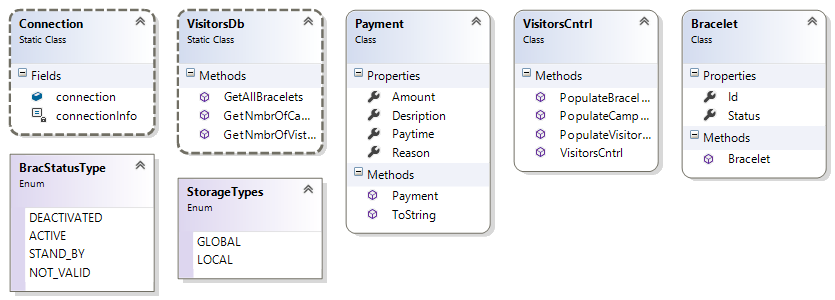
The functionality from the main project involves the two right groupboxes and is based on the class diagram of the StastApp.csproj classes.

The methods rely on the two classes: DBhelper and BinaryFileHelper. The former one provides the methods to retrieve the data from the database: GetNrCurrentVisitors, which returns the number of users who had entered at the event. This method is executed every 100 ms via timer1 of the form1. On every interval of the timer the new data is displayed in the label in the right side of ‘Visitors at the field’ and in the listbox todays’ visitors.

The listbox with 3 buttons on the rightmost side of the window is used to show the data saved for the 1st, 2nd or 3rd day of the event accordingly. On click of a certain day the data from an according binary file is loaded via the BinaryFileHelper’s LoadFromBinaryFile method.

The data is saved as a binary file using the BinaryFileHelper’s SaveToBinaryFile method. The saving is executed every 5 seconds according to the timer2 of the Form 1.

Modules part:



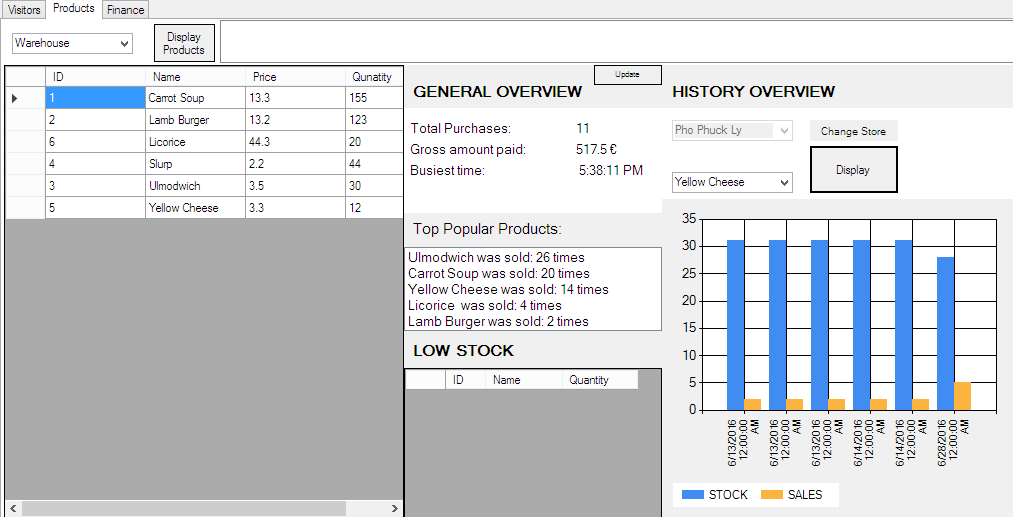
Above is the class diagram of the classes used in the Modules.csproj for the ‘Visitors’ modules. The VisitorsCnrl is used to populate the data by using the methods that take the according components as the parameters and then, by retrieving the data from the database using the methods of VisitorsDB, populates the controls. Below is an example of a pair of methods that populates the numbers for the three following labels. “Present Visitors”(the difference from the current users – is that this number counts only the visitors who have payed and are at the event and the former one counts all – thus it is used to make sure that all the current visitors are the ones who have payed (even though the opposite is very improbable)), “Expected Visitors”, “Festival Accounts”.  
In the left section is the method of VisitorsCntrl and in the right one – of VisitorsDB.

|  |  |
| --- | --- |
| static public void PopulateVisitorGroupData(Label[] lbls)  {  int nmbrTotal, nmbrExp, nmbrPres;  VisitorsDb.GetNmbrOfVistrPerStatus(out nmbrTotal, out nmbrExp, out nmbrPres);  try  {  lbls[0].Text = nmbrPres.ToString();  lbls[1].Text = nmbrExp.ToString();  lbls[2].Text = nmbrTotal.ToString();  }  catch  {  Console.WriteLine("okay, didn't work");  }  } | static public void GetNmbrOfVistrPerStatus(out int nmbrTotal, out int nmbrExp, out int nmbrPres)  {  nmbrTotal = 0;  nmbrExp = 0;  nmbrPres = 0;  ….  DBConnectionDll.Connection.connection.Open();  MySqlDataReader reader = command.ExecuteReader();  while (reader.Read())  {  int.TryParse(reader["TotalVis"].ToString(), out nmbrTotal);  int.TryParse(reader["ExpVis"].ToString(), out nmbrExp);  int.TryParse(reader["PresVis"].ToString(), out nmbrPres);  } |

The data in the left panel is updated on every timer tick and the information about the bracelets can be only force refreshed.

#### Products/Warehouse module

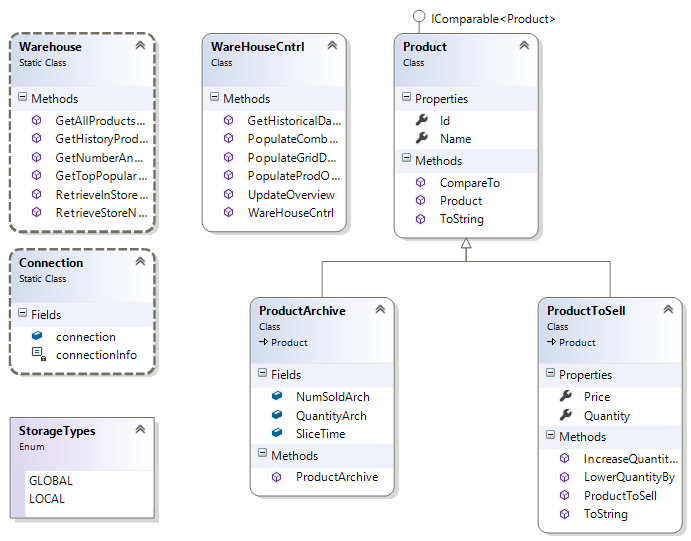
This module is used to keep track of the sales and the product storage conditions – it covers the quantity, number of sales and other statistical data.



The main concept of the separation of tasks between the database interacting and the view populating methods is the same as in the second part of the Visitors Module. This tab makes use of such controls as dataGridView and chart, which makes the tab load a little bit slower.

The left part – the largest data grid view is meant for checking the overview of the products distributed per store or stored in the warehouse globally. The latter one retrieves the data of the foodproducts table and the former – of the storeprodinfo view.

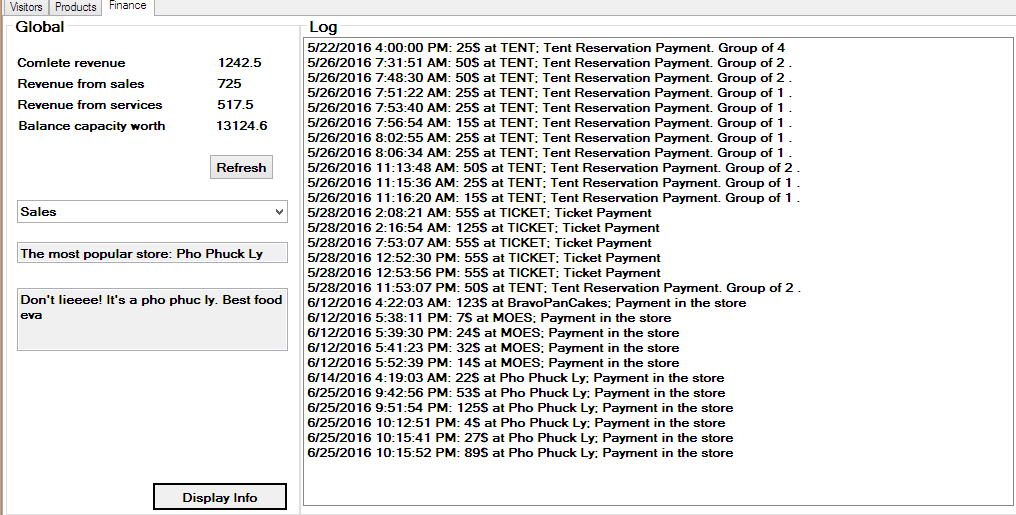
The lower grid is populated when the stock of a certain product in the foodproducts table gets too low.



The chart on the right is built upon the data of the saleslices stored in the storeperfarchive and the data from the current\_quan\_view. For this purpose two types of the product objects are present – two classes: ProductArchive and ProductTotSell are used. They both inherit from the main class Product that inherits from the IComparable interface.

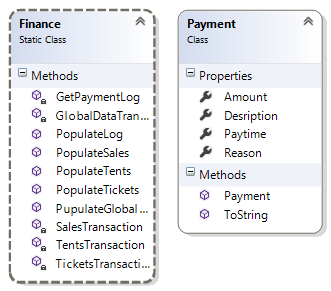
#### Finance module

This module is devoted to keeping the track of the financial data. It presents the overview of the total money earned from the events, shows the data relevant to each type of payment – store and service (divided into both tents and tickets).



The left part is refreshed every time a finance tab is opened or the button refresh is pressed.  
The lower panel featuring the combobox with three options: Sales, Tickets and Tents is used to show the data about the chosen type of payments in the textboxes below. The log on the right is used to present the data about all the payments ever performed at the event (of all kind). The method used is deprived from the Visitor Support application as well as the payment class.

This module is more concise in terms of classes used grace to using the MySQL transactions to retrieve most of the grouped data at once. This facilitates the work with the data retrieved from the database and the way it is retrieved.



Below is the example of the codes that are used to retrieve the data about the tents from the database:

private static List<string> TentsTransaction()

{

List<string> financialTentsValues = new List<string>();

using (MySqlConnection connection = DBConnectionDll.Connection.connection)

{

connection.Open();

MySqlCommand command = connection.CreateCommand();

MySqlTransaction transaction;

// Start a local transaction.

transaction = connection.BeginTransaction();

// Must assign both transaction object and connection

// to Command object for a pending local transaction

command.Connection = connection;

command.Transaction = transaction;

try

{

command.CommandText = @"SELECT SUM(PAYSUM) as tent FROM serpayments WHERE TYPE = 'TENT'; ";

MySqlDataReader reader = command.ExecuteReader();

if (reader.Read())

{

string tentsTotalSum = "Total money sum of tents: " + reader["tent"].ToString();

financialTentsValues.Add(tentsTotalSum);

}

reader.Close();

/////add number of updates

// Attempt to commit the transaction.

transaction.Commit();

Console.WriteLine("Both records are written to database.");

}

catch (Exception ex)

{

Console.WriteLine("Commit Exception Type: {0}", ex.GetType());

Console.WriteLine(" Message: {0}", ex.Message);

// Attempt to roll back the transaction.

try

{

transaction.Rollback();

}

catch (Exception ex2)

{

// This catch block will handle any errors that may have occurred

// on the server that would cause the rollback to fail, such as

// a closed connection.

Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());

Console.WriteLine(" Message: {0}", ex2.Message);

}

}

}

return financialTentsValues;

}

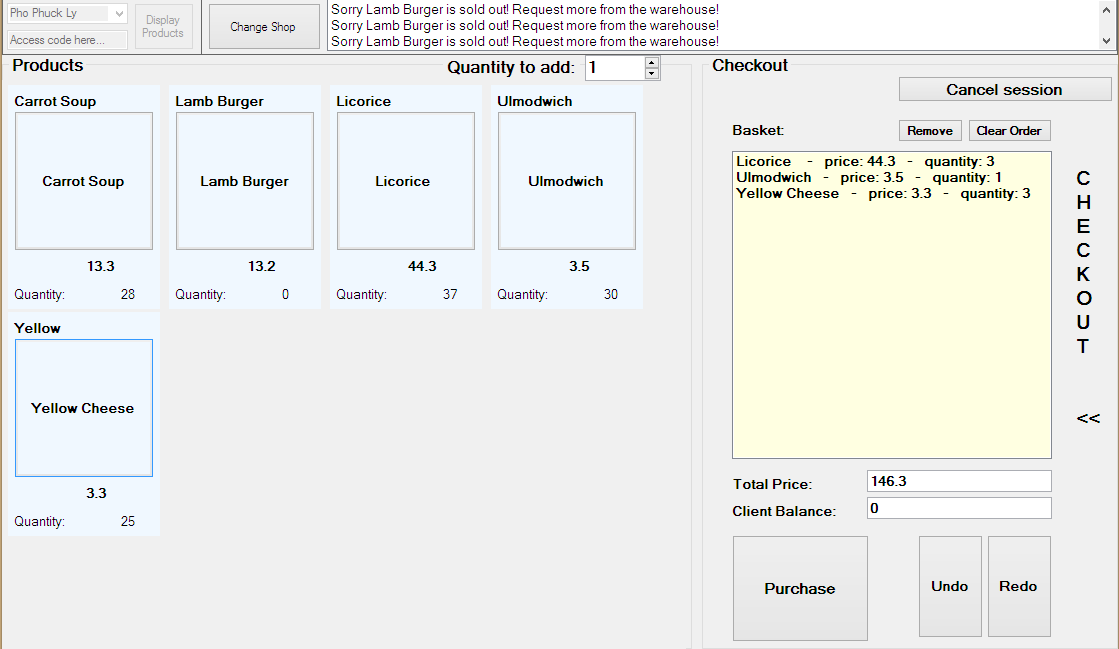
The method above returns the values to be assigned to the labels in one of the populate data methods. Some of the transactions could be replaced by the simple queries as of the time being, but it was done in regards to the continuous development so that more queries could be added.

\*The application is not yet in its potentially perfect state but has some room for easy development in the future.  
\*\*For the additional details and information refer to the source code and comments.

### Shopping application

#### Overview

This application is meant to be used at the shops and the other points of sales. It offers an easy way to sell products to the visitors. No cash is used during the payment – only the visitors’ balance is sufficient and required condition.

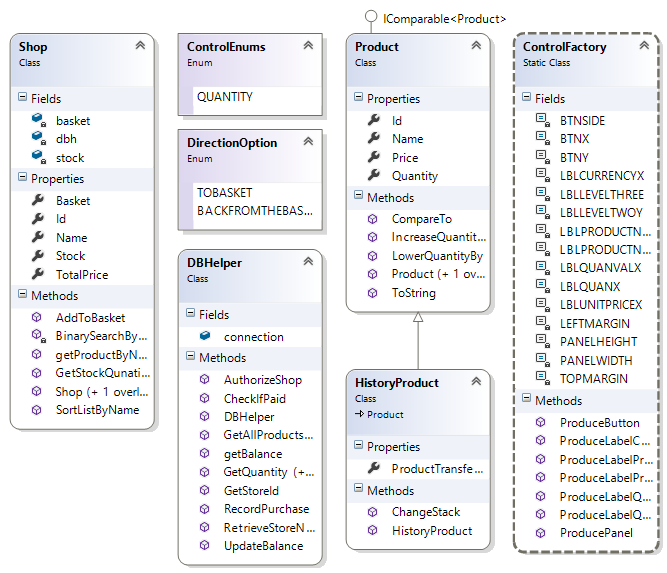


Above is the screenshot of the application at the runtime. As it can be seen the data about the project is retrieved from the database for the according shop chosen in the upper left corner. The upper right corner is used by the log listbox that logs all interactions with the application and the products. Current messages inform that: the “Lamb Burger” is out of stock, and it could not be added to the purchase.

The right panel “Checkout” is used to manage the payment session. The visitor scans the bracelet and his balance is displayed in the current balance text box. A new tag can be scanned only upon pressing “Cancel session” button. Button purchase is clearly used to proceed with a purchase and writes all the changes to the database. Buttons Undo and Redo are used to manage the additions and removals of products to and from the shopping cart. “Clear oder” empties the cart and button “remove” is used to remove a selected item from the basket.

The quantity of the products in the left panel “Products” is managed dynamically and responds to the actions during the session. Until the Purchase button is pressed no changes are written to the database. When it is pressed all the tables that were used in the payment are updated: storepayments, salelines, visitors (balance), productstorages.

#### Class diagram



Above is presented the UML class diagram of the classes used in this application.

* Of a particular interest is the class Control factory that is used for creating the forms of products and populating them inside of the Products panel.  
  The constant properties of this class are used for adjusting the properties of the created controls.   
  Below is an example of code used to create a panel for a product  
    
  public static Panel ProducePanel(ref int X,ref int Y)

{

Panel myPanel = new Panel();

myPanel.Location = new System.Drawing.Point(X, Y);

myPanel.Size = new System.Drawing.Size(PANELWIDTH, PANELHEIGHT);

//myPanel.Name = "pnlTest";

myPanel.BackColor = Color.AliceBlue;

myPanel.Font = new Font(new FontFamily("Microsoft Sans Serif"), 9.75F, FontStyle.Bold);

// moves the starting point of a next panel to the right on an accodring distance

X = X + PANELWIDTH + LEFTMARGIN;

return myPanel;

}

* The difference between a normal product class and a history product is done using the inheritance. The history products are used for redo and undo buttons.   
  For instance when the product was first added and then ‘undone’, when the button redo was pressed it would be restored and added back to the basket – thus the original product has a constructor that creates a copy based on the history product: public Product(HistoryProduct hp)
* In the searching and sorting of products the binary search methods are utilized.