**COS 491 Senior Project I**

**Car Meets Application**

**Gabriel Stoyanov, 100096332**

**Author:**   **, Date:**

*signature*

**Supervisor:**   **, Date:**

*signature*

**Blagoevgrad, 2017**

**Title**: Car Meets Application

**Author**: Gabriel Stoyanov

**Abstract**:

Mobile devices have improved greatly over the years and their capabilities and processing power are comparable with that of PCs that were considered average just a few years ago. These improvements allowed mobile devices replace a lot of items that we use in our lives, such as books, mp3 players and cameras. Their capabilities allowed the creations of many useful applications.

Another factor, which inspired the creation of the application is the fact that, even though autonomous cars are receiving great appreciation among people, leading to less and less people being enthusiastic about driving and cars in general, there are still a lot of people who share a common hobby – cars. People of such kind tend to meet regularly on parking lots or gas stations and talk about their cars, which is most cases is something they are proud of. The problem is that other than following certain social media sources or being in contact with people that organize such meetings, there is really no way to quickly check when and where they are such a meeting.

Therefore the goal of “Car Meets Application” is to connect people with common interest in cars and allow them to easily and quickly find or create car meets. While such meetings are not very common in Bulgaria, especially outside Sofia, they are extremely popular in countries like the United Kingdom and the United States, for example.

Also the application does not require anything special from the user. The usage of the application is relatively simple – a user registers into the system and by doing that created his/her own profile. From that point onwards, the user can create events or view events, including suggested events, which are picked based on an information provided during the registration.

**Declaration of authorship:**

“The Bulgarian Diploma Thesis presented here is the work of the author solely, without any external help, under the supervision of prof. Volin Karagiozov. All sources, used in development, are cited in the text and in the Reference section.”

Author:

**Table of Contents**

Introduction…………………………………………………………………………… page 3

Specification and Analysis of the Software Requirements……………………… page 9

Design of the Software Solution…………………………………………………... page 13

Implementation……………………………………………………………………… page 50

Results and Conclusion……………………………………………………………. page 58

References………………………………………………………………………….. page 60

**Introduction**

The main purpose of “Car Meets Application” is to bring the car community closer together. In order to achieve that goal, the application is allowing its users to easily find or create various meeting events. These events can have very specific requirements to whom can attend them or just a general meeting where everyone is invited to join. There is also a “Suggest Event” feature, which takes into the account the user’s preferences, availability and location and shows him events that meets these specification, while hiding the others. The user is also given the ability to sign up for an event, letting people know that he/she is planning to participate in a given meeting. The list of people attending is visible to everyone else, meaning the rough number of attending people can be planed ahead.

I chose to do develop this application, because I myself am very interested in such events. Unfortunately for me, the official ones are extremely rare in Bulgaria (roughly two times per year). On several occasions I have seen people gathering with friends and their cars on a given place with the idea of just talking about cars, eventually accumulating enough people for this gathering to be called a car meet. This led to me believe that such casual meeting happen randomly and there is not really organization in these groups. With this application I aim to achieve a certain level of organization and making the forming of such meeting easily achievable. After doing some casual research over the years, I have noticed that in countries, such as the United Kingdom and the United States, where the car community is relatively big, people use the social to inform themselves about events, specifically Facebook group. But in order to enter such a group, one must learn about its existence first, which makes it extremely hard for people outside of these closed communities to learn about them.

Essentially, the software that was used for developing this application was Android Studio and PHP. The Android Studio was used for the development of the android application, including its user interface and application logic. The PHP was mostly used as a tool of obtaining the data from the database. The database is hosted on an external hosting service provider (host.bg). The purchased service is with a duration of one year, meaning the application will be active for that time only, in case there is no decision to continue with its development and maintenance. The domain address for the hosting is <http://car-meets.eu/>. The files that are hosted there are only the PHP that are used for manipulating the database. They can be accessed by appending their name to the URL. The names of the files can be seen in their explanation later on the documentation.

The SQL database where the user and event information is kept is relatively simple. There are only two tables (entities), keeping track of numerous attributes – Users and Event. The Users table hold data about the user, such as user ID, username, password, location, age, preferences, availability, car\_info, gender, register\_date and email. The username and password fields hold data about the user’s credentials, which he/she can use to log into the system. The age, ca\_info, gender and register\_date are just general information fields about the user. The location, preferences and availability hold information about the user’s preferences when the user decides to make use of the events suggestion functionality of the application. The email is used to recover a forgotten password. The Event table hold various data about the events themselves, such as event ID, event name, event location, event restrictions, event time, event duration, creator of the event, time of creation and the users attending the event. When it comes to suggesting events, the event time (event starting time) is matched with the user’s availability value, the event location is matched with the user’s location and the event’s restrictions is matched with the user’s preferences. The users attending the event, the time of creation of the event and the creation is simply providing more background information about a given event.

The Android application consists of ten activities. Upon opening the application, the user is introduced into the application through the login activity (LoginActivity.java). This activity consists of two text boxes and three buttons, one of which is styled as a hyperlink. The two text boxes are responsible for the username and the password of the user, respectively. Upon entering valid credentials and clicking Login, the user is redirected to the Profile activity.

In case the user does not have a registration, he/she can click the Register button and this action will redirect the user to the Register activity. In case the user has forgotten his or her password, he or she can click the Forgotten Password button, which is styled as a hypertext, so that it can be more familiar to the user who has even slight experience with websites.

In case the user has decided to register, he or she is presented with the Register activity. It has eight textboxes, along with a spinner and a button. The text boxes are initialized with a hint inside them, to let the user know what he or she is supposed to enter in each one of them. They are responsible for asking the user to enter username, password, location, age, preferences, availability, email and car info. The spinner is letting the user select his or her gender. There was also an idea to replace the car info text box with two additional spinner, which would be populated in a similar way to the one in which most car sale websites use to let their users select the make and model of a car. During attempts to develop this functionality, an additional entity was added to the database (Cars). It contains almost all car makes and models currently in existence. The entity was deleted when the attempt to develop the functionality was abandoned, because it might come in use in future development of the application and repeating the attempts to develop that functionality. Upon clicking the Complete Registration button and if all the fields are filled in correctly, the user will be notified that his or her registration has been completed and he or she can use the credentials in order to log into the system.

If the user has forgotten his or her password, he or she can click the Forgotten Password button and will be redirected into the ForgottenPassword activity. It contains one textbox and one button. The textbox prompts the user to enter his or her email address, and if the format of the email address is valid and the email address is present in the database, the user will be redirected to the ChangePassword activity.

The ChangePassword activity contains a label, holding data about the current user, for whom the password will be changed, two textboxes and a button. The two textboxes require from the user to enter a password and then repeat the same password, in order to ensure that he or she has not typed the password incorrectly the first time. If there are no errors and the two passwords are identical and the user click the button, the password is hashed and updated into the database. Afterwards, the user is automatically transferred to the login activity, where he or she can use the username along with the new password in order to log into the system.

When the user successfully logs into the system, he or she is presented with the Profile activity. It has sixteen labels, eight of which are used to indicate what kind of information is going to be display right after the given label. They have their values initialized right after the activity is started, while the other eight labels hold the database data about the various user information. The last eight labels have empty texts – they are populated after the connection with the database is established and response is split into pieces into an array. When this is done, each array element is assigned as text property to each of the eight labels. These labels hold information about username, location, age, preferences, car info, gender, member since and availability. The data that is extracted from the database is colored in red. There are also three buttons – Edit Profile, Create Event and View Events.

If the user selects the Edit Profile button, he or she will be redirected to the ProfileEdit activity. It has nine labels, seven textboxes and one button. The first two labels are about the username. The first one has a value “Username” and the second one is initiated with the username that was taken from the sharedPreferences, which was updated with the user’s username when he or she signed into the system. The other seven labels are responsible for noting what kind of data their corresponding textboxes are supposed to have – location, age, preferences, car info, gender, email and availability. The button has a text value Submit and upon clicking it, if there are not errors with the user entered data, the information about the user will be updated in the database and the user will be redirected to the Profile activity again and will be shown the corresponding updated profile data.

If the user selects the Create Event button, he or she will be redirected to the CreateEvent activity. This activity has five labels, five textboxes and two buttons. The labels, as in previous activities, are used to notify the user what kind of data must be entered in the neighbor textbox. The labels are for event name, event location, specific information, event starting time and event duration. The user is supposed to enter correct data into each textbox, based on what the label requires. One of the button has a text value Clear and what it does is set the text value of all textboxes to empty (“”), clearing the data entered. The other button has a text value Create Event and if all the textboxes are filled in correctly by the user a new event entry will be added into the database. Upon completion, the user is notified that the event was created successfully and he or she is being transferred back to the profile activity.

If the user selects View Events button, he or she will be redirected to the EventDisplayType activity. This activity lets the user choose whether he or she wants to view all events or only suggested events. It has only two button with text values View All Events and View Suggested Events. Based on the user’s choice, a string is saved in sharedPreferences, which will be later used to determine which events to show, whether all or the suggested ones only. Upon making a decision and clicking a button, the user is transferred to ViewEvents activity.

The ViewEvents activity implements a ScrollView. The reason for using ScrollView is that the number of events might be more than what a single phone screen can fit in, so it created a scroll control, so that the user can view all events, even the one that could not fit on the screen. Before the activity is fully initialized, it is empty. The reason for that is that the activity is populated with data dynamically, based on the response from the database. The code behind the activity created a button for each event that was returned as an entry from the database and places each one below the previous button in a linear layout. The button text contains all the data about specific event. If the user wants to see all the information about specific event and eventually sign up for one, he or she can simply click the button. By clicking the button, an event name is saved is sharedPreferences and is later used to display data about this particular event.

When the user clicks a button about a specific event, he or she is redirected to the EventInformation activity. This activity has fourteen labels. Seven of them are used to display information about what kind of information is displayed by its neighbor label. The labels are for event name, event location, additional information, starting time, event duration, event creator and time of creation. There is also a button with text value Sign Up, which lets the user sign up for an event. If the user is not signed up for an event already, he or she is notified that the sign up was successful. If the user is already signed up for that event, he or she is notified that the sign up has already been made.

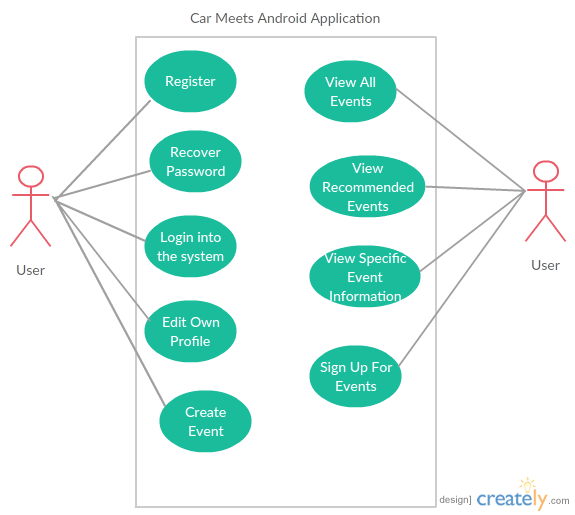
Several PHP files have been used, in order to ensure the proper connection to the database and extract the requested data or perform the requested action on the database. The application deals with the database by invoking a particular link to the appropriate PHP file in the server and appending the required data, if any. Those files do not implement any kind of user interface. If the user navigates to a PHP link manually, without appending the required data or if the file is not designed to return any data, he or she will encounter an empty blank page without absolutely anything in it. It is done that way, because the user interface is not needed for the application, what is important is to receive the requested data or perform the required action. These files are not meant to be opened manually via the browser.

There are thirteen PHP files. The first few lines in each file is absolutely identical, because it is responsible for establishing the connection with the database, which would be the same of each communication request with the database. The connection is established by creating a variable that is an object to the connection method – mysqli\_connect. In order for the method to work properly, there is some specific information passed as parameters to it, such as database location, which in this case is localhost, because the PHP files are hosted on the server where the database is hosted. Afterwards, the username and password are password, in order to provide authentication and gain access to the database. The last parameter is the name of the database, which in this case is “carmeets\_cm”. After the connection is established, a check is performed to check whether the connection has failed or was successful. In case the connection has failed, an error message will be displayed on the webpage, stating that the connection has failed, otherwise there will be no such message and the execution of the code will continue. In the end of each file, there is a piece of code that ensured that the connection is closed by calling the mysqli\_close() method along with the connection object that was used to store the connection call. Below is a brief explanation of what each file is responsible for:

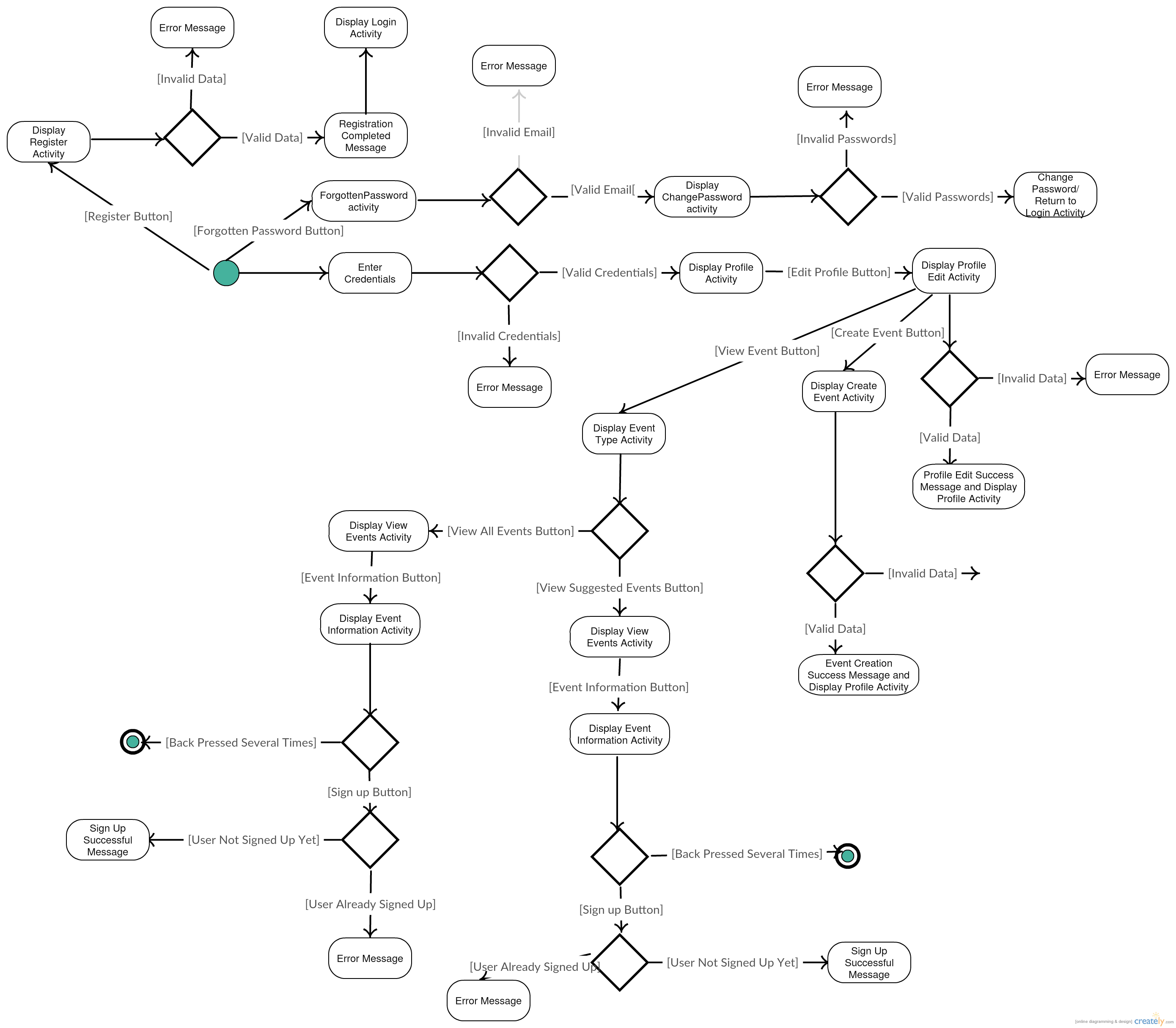
* **Reg.php** – takes as parameters through the POST method username, password, location, age, preferences, availability, car info, gender and email and inserts that information into the database. It is used to register a new user into the database.
* **Mail\_check.php** – takes as parameter through the POST method email and check whether it is present in the database. If there is a record in the database that corresponds to this email, a username will be printed, otherwise an “f” will be printed, which stand for failed.
* **Password\_change.php** – takes as parameters through the POST method username and password. Afterwards it changed the value in the password field with the passed value where the username matches the passed username. Afterwards, a “success” message is displayed.
* **Login.php** – takes as parameter through the POST method username and password. It uses the values that were passed to compare with the corresponding username and password. In case both are identical, the username is printed, otherwise “f” is printed, which stand for failed.
* **Profile.php** – takes as parameter through the POST method username. It uses that username to find the corresponding and prints all the information about that user, such as location, age, preferences, car info, gender, register date and availability.
* **Get\_profile\_data.php** – takes as parameter through the POST method username. It is very similar to the profile.php, but it returns a bit more information. It returns location, age, preferences, availability, car info, gender and email. It is used to populate the textboxes when the user clicks the Edit Profile button.
* **Update\_profile.php** – takes as parameter through the POST method username, location, age, preferences, availability, car info, gender and email and updates the existing values in the database with the new ones, based on the username match. When the query is executed successfully, “s” is printed, which stand for success.
* **Create\_event.php** – takes as parameter through the POST method event name, event location, event restrictions, event starting time, event duration and event creator and inserts them into the database, i.e. creating new entry in the Event table. Upon completion, “Query Successful!” message is displayed, notifying that the insertion was successful.
* **Event\_sign\_up.php** – takes as parameter event name and users attending and uses them to update an existing record’s user\_attending field with the new value. It recognizes which event to update based on the event name that was passed. Upon completion, a “success” message is printed, notifying that the query was successful.
* **View\_all\_events.php** – does not take anything as parameter through the POST method. Instead it selects all records from the Event table and print each one with all available details, such as event name, event location, event restrictions, event starting time, event duration, event creator and event creation time, separating each entry by “&” and each event entry with “#”.
* **View\_specific\_event.php** – similar to the view\_all\_event.php, but this one takes event name as parameter through the POST method and prints the available data about particular event, such as event name, event location, event restriction, event starting time, event duration, event creator, event creation time and users attending. It uses “&” as a separator between each entry.
* **Car\_data\_make.php** – used during the attempts to create the spinners, which were supposed to offer the user with makes and models options, like the big car selling websites are doing. The file is kept there, although it is not currently used, because it will be used in future development.
* **Car\_data\_model.php** - used during the attempts to create the spinners, which were supposed to offer the user with makes and models options, like the big car selling websites are doing. The file is kept there, although it is not currently used, because it will be used in future development.

**Specification and Analysis of the Software Requirements**

1. Functional Requirements
   1. The application is limited to registered users only – non-registered users will not be able to use the application, since they will not be able to proceed past the login screen.
   2. Data must be entered correctly before submitting any requests towards the PHP’s queries – no blank boxes and incorrect data allowed.
   3. The application should provide the user with the ability to recover his or her password through the ForgottenPassword activity.
   4. The application should correctly authenticate the user.
   5. The application should correctly encrypt the user password when sending a login request to the database.
   6. The application should correctly decrypt the password when receiving a login response from the database.
   7. The application should correctly compare the password that was entered by the user and the password that was passed as a response by the database.
   8. The application should correctly populate the textboxes in the ProfileEdit activity.
   9. The application should display the appropriate events when requested by the user.
   10. The application should correctly update the database with the new event that the user specified.
   11. The application should correctly display the profile information after signing in.
   12. The application should correctly sign up a user for a specific event that he or she requested.
   13. Requires internet connection in order to modify and extract data from the database.
2. Non-functional requirements
   1. (Performance) Query executions themselves should not take more than two seconds to be executed.
   2. (Performance) The transitions between the activities should not take more than two seconds.
   3. (Portability) The application should run on all android devices that were produced no later than two years ago and have the latest software updates (Tested on Android 6.0.1).
3. Constraints
   1. The application will not run on devices that are not using android operating system



Use Case Diagram



Activity Diagram

**Design of the software solution**

The application implements the MVC (Model-View-Controller) pattern, which is in a way applied by default by the Android Studio environment.

The code, responsible for the display of the data to the client, is stored in xml files, generated by the environment. There is such file for each activity that stores the code generating the design of the specific activity. It also contains information about each element, such as text boxes, buttons etc. like IDs, names, texts, width, height, for example.

The applications has several database connection methods that are used for either manipulating the database in certain ways, or extracting output from the database. In order to achieve this, the application uses the java.net library to invoke the corresponding PHP for the given operation. It passes the data to the PHP by using the POST method, instead of the GET method. In order to achieve that, a link to the PHP is provided and extended with the required variables following the link. The extensions are formatted in UTF8 format. The communication with the database is done asynchronously, performing the requests in the background by invoking the function doInBackground(). Since this communication requires certain time to complete all interactions with the database, the application displays a loading animation throughout the whole process. While the animation is present, the user cannot interact with the application, but instead he or she can make sure that everything is complete once the loading animation is gone from the screen.

For the database interactions that need to return data, the application uses the java.io library, and more specifically the BufferedReader, InputStreamReader and OutputStreamWriter. The OutputStreamWriter servers as a bridge from character stream to byte stream. The BufferedReader stores the output from InputStreamReader. The InputStreamReader gathers the output from the database. Afterwards, a StringBuilder object is created, appending the data and afterwards assigns the result to a string in the corresponding class that will need to work with that data.

The database interaction classes have a constructor, used to assign values to the previously initialized variables. Such classes are called from the various activities with the corresponding variables, in order to have the constructor properly assign them in the class itself.

Most of the code, related to the database connection is very similar. On various attempts to combine most of these into one class with related functions, the application was constantly throwing null pointer exceptions. Hence, I made the decision to keep each database operation in a separate class.

The code for the initial activity, when the application starts, is located in the LoginActivity class. It is coupled with the Login\_Input\_Checks and Password\_Hashing class. For the database connection, it is also coupled with the DB\_Login class, which is responsible for the connection. The Password\_Hashing object is created, so that the activity can gain access to the md5 method, which is used to hash the value entered by the user in the password field, so that the hashed value can be passed to the PHP, so that the credentials can be checked if they are valid or not. It has three global variables, which are made static, so that the DB\_Login can insert the data from the database into them. The “log” screen is responsible for storing the returned data, while the “log1” and “log1l” are used for performing various check for the data. The onCreate method contains several initialization statements about the UI elements present in the activity, such as buttons, Edit Texts and TextViews. There is also a string array with one element, that is used to store the hashed the password. It is done in an array, so that it can be manipulated by other method that are not inside the onCreate method. There is also an onClickListener for the login button. This method is used to define the functionality of the Login button when it is clicked. It stores the hashed password into the variable by calling the md5 method from the Password\_Hashing class, along with text as parameter, taken from the password edit text. Afterwards, two checks are called, in order to ensure that the username and password fields are correctly filled and not empty. Lastly, both check methods are called again and if they both return true, since they are Boolean, the login method is called, along with username and password as parameters. It instantiates a SharedPreferences objects, in order to store the user’s usernamet, which will be used in other activities. SharedPreference seemed like a better way to store such information between activities instead of passing the data through the intents, therefore it was chosen for this situation. Later on the function calls the execute() method for the class DB\_Login, along with the link leading to the corresponding PHP and the data for login – username and hashed password. Afterwards a ProgressDialog is created, which is essentially the loading animation, notifying the user when the program is busy communicating with the database. Once the returned string from the PHP’s response is empty, the ProgressDialog is closed, along with a timer which runs while the communication is present. If the credentials were correct, the method notifies the user with a Toast message that he or she has successfully logged in. Then the user’s username is added into the SharedPreferences and the startAcivity() method is called in order to transition from the current activity to the next – the activity responsible for the selection of the tables. Upon successful login, the user is greeted with a Toast message containing his or her name, which is being extracted from the database. In case the credentials were not correct, a Toast message is displayed notifying the user that the login has failed. There are also two more onClickListeners, responsible for the Forgotten Password button and the Register button. Both of them simply use the startActivty() towards the corresponding activity – ForgottenPassword.class and Register.class.

The ForgottenPassword activity is called when the user presses the Forgotten Password button. In this activity, the user is expected to enter his or her email. The function mail() works in a similar way to the login function from LoginActivity. It also instantiates a SharedPreferences objects, which in this case is used to store the email that the user entered. Afterwards it uses the execute() method on the class DB\_EmailCheck, along with the PHP link and email to be processed. Similarly to login method, the next step is creating a ProgressDialog to notify the user that the application is currently communicating with the database. After that a timer is created, making sure that the ProgressDialog will stay on display as long as the application is done communicating with the database – until the output string from the PHP is empty. If the initial output from the PHP was with different length thatn 1, the email is added to the SharedPreferences. Afterwards, the user is redirected to ChangePassword activity. In case the response from the PHP was with length 1, meaning the email address entered was wrong, a Toast message is displayed notifying the user that he or she entered a wrong email address.

Upon entering a correct email and clicking the Submit button, the user is redirected to the ChangePassword activity. It contains two textboxes, prompting the user to enter a new password and reenter the password in the second textbox. If the two passwords do not match, the user is notified via Toast, alerting him that the password does not match. Otherwise, the password is encrypted into MD5 format and the passwrd() method is called. It uses the execute() method on class DB\_PasswordChange, along with the link, email and password as parameters to be used to be used by the class’ constructor. It then initiates a ProgressDialog, to notify the user when the application is done with its communication with the database. If the operation has been successful, the user is notified through a Toast, saying that his or her password has been changed. If it fails, the Toast’s message is: “Could not change password.” There is also a label notifying the user of the username for whom the password is being changed.

When the user clicks the Register button from the login screen he or she is transferred to the Register activity. This activity contains several global variables that are meant to be populated by the database connection class’ code. There are also some other global variables that are currently not used, but they will be used in future development. They are meant to be used for the spinners for make and model, which is planned to be developed in future. The onCreate method initializes several UI elements, which are present in the activity, such as Button, Text Views, Edit Texts and one Spinner, that is used for gender selection. There is also an instance of class Register\_Input\_Checks, which provides access to methods located in Register\_Input\_Checks class. There are also two string array, used to store data about the user’s gender and hashed password. There is also a Boolean array, used to store a value, indicating whether there was an error in the user entered data or not. The reg() method is responsible for the database communication about registration. It uses the execute() method on class DB\_Resgister, along with the link, username, password, location, age, preferences, availability, car info, gender and email to be used by the class’ constructor. It then initiates a ProgressDialog to notify the user when the application is done with its communication with the database. If the communication is successful, the user is notified that the registration has been completed. There is not statement notifying the user if the registration was not complete due to data error, because this is handled in the onClickListener for the Register button, before calling the reg() function. The onClickListener for the Register buttons first checks the data in each of the provided text boxes to make sure that the user has entered all the data correctly and did not leave a field blank. In case there is an error with a given field, an error label is displayed and isCorrect (the Boolean variable responsible for making sure whether the data that was entered was correct) is set to false. After all the checks are done, a check is performed to see whether the isCorrect’s value is true or false. If it is false, there are not additional actions taken and the user is left with the chance of editing the information so that it can be acceptable. If isCorrect is true, the reg() function is called along with the needed parameters, such as username, password, location, age, preferences, availability, car info, gender and email. After the method completed its work, a startActivity() method is used to transfer the user from the current activity to the Login activity again, so that he or she can use his or her credentials.

When the user enters valid credentials in the Login activity, he or she is transferred to the Profile activity. This activity contains various information about the logged user, such as username, location, age, preferences, car info, gender, time of registration and availability. There is also a profile picture, which at this point is a default one and it cannot be changed. This functionality will be developed in the future. This activity contains three global variables, which are used by the database connection, in order to store data about the returned data from the database. Only the first one – “data” variable is used by the external class, while the other two – “data1” and data1l” are used to perform various check about the data and manipulate it in different ways. There are also global strings userLocation, userPreferences, userCar, userGender, userRegisterDate, userAge and userAvailability which are being populated after the database response is received and split appropriately. These strings are used to hold data that will later populate the labels, responsible for storing the various user information. Some of them are used to also populate sharedPreferences fields, in order to make this information accessible for other activities without having the need to establish another communication with the database. The activity also has a few statement to initialize the elements in the user interface, such as TextViews (username, location, interests (preferences), car\_info, gender, member\_since, age and availability), three buttons (Create Event, View Events and Edit Profile). There are also statement to create object for various sharedPreferences instances, such as username, location, preferences and availability. There are also editors initialized for each sharedPreferences instance, except for the username. The reason for this is that this activity adds data to all other instances other than username. The username sharedPreferences is already populated from previous activity and it is used simply to extract data from it. When the activity is instantiated and the user interface controls are loaded and initialized, the activity starts establishing database communication. It is done by calling the DB\_ProfileData execute() method, along with the required link and data. It is then creating a ProgressDialog and a Timer, in order to keep the loading animation active while the communication is established and completed. After the communication is completed and the response from the database is saved in the global variable “data”, the ProgressDialog is cancelled. Afterwards, the data is processed and manipulated in an appropriate way, so that it can be easily be used to populate the various text views. In order to do that, the data is split using “&” as delimiter and saved in a string array (data\_array). Then the username field is populated based on what value was extracted from the username sharedPreferences. The location is taken from the first element in the array and is being set as value for the location textView. After this value is set, the location is added to the location instance of sharedPreferences and the apply() command is used for the location instance of sharedPreferences, so that the changed can be saved. The following information, which is being filled in the textViews, is the preferences data. It is extracted from the third element in the response array. The value is assigned to the interests (preferences) textView, respectively. It is then added to the sharedPreferences by using the appropriate object and afterwards executing the apply() method to save changes. The user’s car information is extracted from the fourth element in the array and is assigned to the car\_info textView. This information is not saved to sharedPreferences, because it will not be used in any following activities, so there is no need to save it. The user’s gender information is extracted from the fifth element in the array and is assigned as text value to the gender textView. The registration time is extracted from the sixth element of the array. Since the returned data from the database is not in the appropriate format, a function was created and called to construct an appropriate format by manipulating the response information. The returned value is assigned to member\_since textView. The data about the user’s age is extracted from the seventh element in the array. It is then assigned to the age textView. The user’s availability is extracted from the eight and last element of the array. It is set as a value for the availability textView’s text. The availability value is also saved in sharedPreferences by using the corresponding object for it and calling the apply() method to save the data. Afterwards, the timer is canceled. There is an “if” statement, checking whether the response length was 0. If it is 0, the user will be notified that the extraction of data from the database has failed and the abovementioned process will not be executed. It will be performed only if there was data returned.

The convertDate method is used to transform the response data from the database, the one about user’s register data. Since the database saves this as CurrentTimeStamp, it saves the data as time of registration along with date. What his method essentially does is to get rid of the time and save the date only in an appropriate format. In order to do that, it takes as parameter the raw date. It then splits this data using a single space (“ “) as delimiter and saves it to an array called dateArray. Since the text field to display the date of registration is only interested in the date itself, only the first element of the array is saved to a variable (date1). The second element of that array contains data about the time of the registration. Since the data format is YYYY-MM-DD (year-month-day), this method is also responsible for reversing it into DD-MM-YYYY format (day-month-year). In order to do that, the first part of the array, which was saved into date1 string variable is again split, this time using “-“ as a delimiter and is saved into dateArray1. Then the correct format is built into the newString variable by using the append method. Since the format has to be reversed, the items from dateArray1 are appended in a reverse order – third element is appended first, followed by the second element in the middle and lastly the first element in the array. Then the function returns the newly built string newString, which hold the correct format.

There are also three onClickListeners in this activity, responsible for specifying the functionality of each button. All three of them are similar, because these buttons are only responsible for transferring the user to various other activities.

If the user decided he or she wants to edit his or her profile and click the Edit Profile button, he or she is transferred to the ProfileEdit activity. This activity has two sets of global variables used to hold data about the database response. The reason for this is because there are two database connection methods in this activity. The first one is executed in the onCreate method, right after the user interface has be loaded and the controls are being initialized. This connection is established in order to populate the EditTexts with the user’s information, so he or she will not have to enter all the data from scratch, but instead edit only the data he or she wants to, while preserving the rest. There is also an isCorrect global Boolean variable, which serves the same purpose as the one in the Register activity – it keeps track of whether there was an error in the data entered by the user. It is also used to perform a check and decide whether to give the user another try at editing the data with correct information or execute the second database connection, responsible for updating the database with the new information. This activity makes use of an instance of class Register\_Input\_Checks, in order to validate the data entered by the user. It is not using a separate class, because the data that the user is identical to the data that he or she is asked to enter upon registration.

There is also a single button in the activity (Submit), which is supposed to be clicked when the user is done editing his or her information and wants to submit it into the database, in order to apply the changes made to the profile.

The activity contains various user interface controls, which are the first thing that is being initialized in the onCreate method after the instance of the Register\_Input\_Checks class. There controls are a button (Submit), TextViews used for labels (username, location error, age error, preferences error, car info error, email error, availability error), along with various Edit Texts (location, age, preferences, car info, email, availability) and a spinner for gender selection. Once all controls are initialized, an instance of sharedPreferences for username is created. It is used to obtain the current user’s username. After the creation of the instance, connection to the database is established. It is used to obtain the user’s register information, such as location, age, preferences, car info, email, availability and gender. In order to do this, the execute() method of DB\_GetProfileInfo is called, along with a link to the corresponding php (get\_profile\_data.php) with the addition of the username, since the php file looks for the user based on his or her username, in order to return the data for the correct user. Upon calling the execute method, a ProgressDialog is called, along with a Timer. They are both used to keep the loading animation active while the communication is completed. Once the execution of the DB\_GetProfileInfo is completed, the output from the database response is checked. If it different than an empty string (“”), the ProgressDialog is cancelled. Then the response is split into an array (data\_array) using “&” as delimiter. The username text field is populated from the data that was gathered from the username object of sharedPreferences. The text for location is extracted from the array’s first element and inserted as text value of txt\_location, the name of the text field which holds information about the user’s location data. The user’s age is extracted from the second element of the array and inserted as text value of txt\_age, the name of the text field which holds information about the user’s age data. The user’s preferences are extracted from the third element of the array and inserted as text value of txt\_preferences, the name of the field which holds information about the user’s preferences data. The user’s availability is extracted from the fourth element of the array and inserted as text value of txt\_availability, the name of the field which holds information about the user’s availability information. The user’s car info is extracted from the fifth element of the array and inserted as text value of txt\_carInfo, the name of the field which hold information about the user’s car information. The user’s gender is extracted from the sixth element of the array and saved in gender string variable (which in this case is an array, because it has to be accessed from outside the method it is being defined. In order to avoid using the “static” modifier, it had to be an array). The user’s email is extracted from the seventh element of the array and inserted as text value of txt\_email, the name of the field which holds information about the user’s email information. Afterwards, based on the value of the gender string array’s first and only element, the spinner’s value is selected. If the value of the array is “Male”, the first selection option is programmatically selected, while if the value is something other than “Male” (in this case it can be only “Female”), the second selection option is programmatically selected. There is an if statement performed, in case the response string’s length is 0 and if it is 0, this means the database information extraction has failed and the user is notified of it. In case of 0 length, the above explained code is not being executed – it is being executed only if the length of the response is string is more than 0, i.e. it is not empty.

The onClickListener for the Submit button in this activity is very similar to the one in the Register activity. It sets the isCorrect value to true first, which is the default value. Then various checks are performed to make sure the user entered a valid value in the corresponding text fields. First the location check is performed, followed by an if statement, calling the same location check function. If the value entered by the user in the location field is incorrect, the isCorrect value will be set to false. Afterwards, a check is performed for age value that was entered by the user, followed by an if statement, calling the same age check function. If the value entered by the user in the age field is incorrect, the isCorrect value will be set to false. The following check that is being called is for the preferences field, followed by an if statement, calling the same preferences check function. If the value entered by the user in the preferences field is incorrect, the isCorrect value will be set to false. Afterwards, the availability check is performed, followed by an if statement, calling the same availability check function. If the value entered by the user in the availability field is incorrect, the isCorrect value will be set to false. Following the availability check, car information check is performed, followed by an if statement, calling the same car information check function. If the value entered by the user in the car information field is incorrect, the isCorrect value will be set to false. Lastly, the email check is performed, followed by an if statement, calling the same email check function. If the value entered by the user in the emal field is incorrect, the isCorrect value will be set to false. The last part of the onClickListener is to check whether the isCorrect is set to false or true. In case the value is false, the user will be notified that there was an error in the data that he or she entered and the isCorrect will be set to true (reset to default value). No further actions will be taken. In case the isCorrect value is set to true, the reg method is called, along with the data it needs, such as username, location, age, preferences, availability, car information, gender and email. After the reg() method is done executing code, the user is transferred back to the Profile activity.

The reg() method in the ProfileEdit activity is responsible for the database communication. It takes as parameters username, location, age, preferences, availability, car information, gender and email. Afterwards, the DB\_ProfileUpdate class’ execute() method is called, along with the required data needed by its constructor, such as link (<http://carmeets.eu/update_profile.php?>), username, location, age, preferences, availability, car information, gender and email. Once it is done, a ProgressDialog is created, along with a Timer, in order to make sure a loading animation is displayed to the user until the communication with the database is complete. If the “res” string, which keeps data about the response from the database is not an empty string, i.e. something other than “”, the user will be notified that the profile update process has been completed successfully. After this notification, the ProgressDialog is being cancelled, along with the timer. Afterwards, the user is automatically transferred back to the Profile activity, with a line of code, specified in the onClickListener of the Submit button.

Other than the Profile editing option, the user is presented with two more options in the Profile activity. These two options are to Create an event and to View (an) Event(s).

In case the user decides to create an event, he or she is transferred to the CreateEvent activity. The layout of this activity is similar to the registration layout and the profile editing layout, but here it is about creating an event other than creating a new user.

The first initialization in the activity is for the global variables, which will be used for storing information about the database response. The string variable “create” is the one storing the initial response value, while the other two are used for manipulating that value in various ways, while preserving the original value. The onCreate method takes cares of initializing all the user interface controls, such as buttons, edit texts and text views (used as error labels), before specifying the onClickListeners for the buttons. The activity contains two buttons – Create and Clear. It has five edit texts, providing the user with the ability to input the required data, in order to create an event. As mentioned above, the labels are used for notifying the user about any errors that he or she might have made while inputting the event details. In order for them to function properly and not show when the activity is first initialized, all five labels are set to invisible. The edit texts provide the user with the ability to input data about event name, event location, event specific information, such as restrictions, event starting time and event duration. there is a label for each edit text, located directly below it, which is being displayed if there is an error in the data entered in the corresponding field. There is also an instance of Register\_Input\_Checks created, in order to make use of three of the method, located in the class. The three methods are emptyField\_Check, starting\_Time and duration\_Check. Those methods are used to validate the user input data. There is also an instance of sharedPreferences for the username. The value from that object is set to the string variable user, which will be used to upload information into the database about which user created the event. The last initialization of a variable is the Boolean variable isCorrect, which tracks whether there was an error in the user data.

The Create button functions in a similar way to the registration of a new user and editing the profile. This is due to the fact that all these function require validation of the data before submitting any information to the database, and that is why their syntax is very similar. First the event name field is checked if it is empty. In case the field is empty, the corresponding error label for the event name will be show below the text box by setting its visibility to VISIBLE. After the execution of the method, the same method is used in an if statement. In case the returned value from the function is false, the isCorrect value is also set to false. After the event name, the event location field is checked if it is filled with data by calling the same method emptyField\_Check on the event location edit text. In case the data entered in this field is wrong, i.e. the field is empty, the error label for the location field will be displayed. Afterwards, the same function is called in an if statement, checking whether the field is not empty again, but this time the isCorrect value is set to false, in case there is an error with the data entered by the user. Similar to the previous two fields, the event specificInformation field is also checked if it is empty. First the method emptyField\_Check is called, along with the specificInformation edit text field. In case the field was left empty, an error message will be displayed below the specificInformation text box, notifying the user that the field should not be left empty. Afterwards, the same function emptyField\_Check is called upon the specificInformation text box and in case the returned value from the function is false, the isCorrect value is also set to false. Similar to the above three text boxes, the validity of the event starting time is also checked. This time, instead of only checking for empty field, the format of the starting time is also checked for validity. The function is start\_Time is called with eventStartingTime text box as parameter, along with the corresponding label. In case the field was empty or the user has entered invalid format, an appropriate message will be displayed. In case the field was left empty, the user will be notified that the field should not empty and in case the format was wrong, the user will be notified and will be informed what the correct format is, so he or she may know what is expected to be written in that field. Afterwards, the same function start\_Time is called in an if statement, along with the eventStartingTime text box and the corresponding error label. In case the function returns false, the isCorrect value is also set to false. The last of the text boxes input checks concerns the user input for the event duration. Just like with the starting time, there is a separate function that checks the validity of the field along with a check whether the field is empty. If any of these two conditions turns out be true, the user will be notified that either the text box is empty, while it should not be, or that the input was wrong and the user will be notified about what value is expected to be written there by providing an example. These errors will be displayed below the corresponding text box, in the error label. After all the input checks are performed, an if statement is created, checking the value of isCorrect variable. In case the value of isCorrect is false, its value is set to true and the user if given the chance to correct his or her errors in the text boxes without submitting anything to the database. In case the value of isCorrect is true, meaning all user input is correct and there are no error, therefore it is safe to submit a request to the database, the createEvent function is called, along with the needed information needed to create this request, such as event name, event location, specific information, starting time, duration and event creator.

The createEvent function is the function that is responsible for invoking a request to execute the code, located in the class responsible for the database connection. This functions also takes care of handling the response data. The function first calls the DB\_CreateEvent’s execute() method, along with the required data by the constructor of the class. The data needed for the proper execution of the class is the link to the php (<http://carmeets.eu/create_event.php?>), event name, event location, event restriction, event starting time, event duration and event creator. After calling the execute() method, a ProgressDialog is created, along with a Timer. The reason for this is to notify the user that the application is currently communicating with the database and the response is being processed. The response value, which is stored in the “create” string variable. If the value of “create” is different than an empty string (“”), the ProgressDialog is cancelled. Then the user is notified that the event creation process has been successful and the timer is cancelled. In case the length of the response was 0, i.e. an empty string, the user is notified that there was an error creating the event.

The second button in the activity is the Clear button. It is a simple button responsible for clearing all the data entered by the user in the text boxes with one click. In order to this, the text property of each text box (event name, event location, event specific information, event starting time and event duration) is set to an empty string (“”) and after such statement the corresponding error label’s visibility property is set to INVISIBLE.

In case the user wants to view the events from the Profile activity, he or she can achieve that by clicking the View Events button. Upon clicking the button, the user is transferred to the EventDisplayType activity. This activity is used to provide the user with the choice whether he or she wants to view all events or the suggested events. It has two buttons – View All Events and View Suggested Events. The activity also creates an object of sharedPreferences about display, storing information of the user’s choice. If the user chooses to see all events, the value in the sharedPreferences will be “All”, while if the user wants to see the suggested events only, the value will be “Suggested”. Since no matter what the user chooses, he or she will be transferred to the ViewEvents activity, instead of using two separate activities for each choice, the sharedPreferences objects can be used by the ViewEvents activity to properly display the events that user chose.

Upon clicking the View All Events button, the user is transferred to the ViewEvents activity and the value of the “display” object in sharedPreferences is set to “All”. Afterwards the apply() method is called, in order to save this value for future use, i.e. in the ViewEvents activity.

Upon clicking the View Suggested Events button, the is transferred to the ViewEvents activity and the value of the “display” object in sharedPRefernces is set to “Suggested”. Afterwards the apply() method is called, in order to save this value for future use, i.e. in the ViewEvents activity.

The ViewEvents activity is slightly longer when it comes to lines of code, simply because it is essentially two activities combined in one. In order to achieve it, instead of creating the user interface controls by hand through the Design mode of the activity, they are created dynamically.

The activity beings in a similar way to the other activities – by initializing three global variables to hold and manipulate data obtained through database response. The “res” string variable is responsible for storing the initial database response, while the “res1l” integer variable stores data about the length of the response. There is another integer variable, which is used for keeping track of the buttons’ IDs when they are being created.

Since this activity does not have any user interface controls added by hand, there are also none to be initialized in the onCreate method. Instead there are various shardPreferences objects initialized in this method. These objects are responsible for accessing sharedPreferences data about event display type, location, preferences and availability. After each objects initialized there is a string variable created that stores the data, located in each sharedPreferences objects, respectively. Those variables are called displayType, location, preferences, availability.

This activity contains the functions, used to specify the logic for events suggestion. There are three functions responsible for making sure that the event suggestion is working properly and functioning the way it is supposed to work. Those three functions are called “location\_check”, “preferences\_check” and “availability\_check”. What they does essentially is comparing the event value for location, restrictions or starting time with the user’s value for location, preferences or availability, respectively. In case those value are identical the functions will return true, otherwise they will return false.

There is also a createButton function, which is responsible for the dynamical creation of the buttons in the activity.

In order to differentiate between the two options of displaying events, which the user had in the previous activity, there is a switch statement. The switch check is performed over the displayType string variable, which is obtained from the “display” value of the sharedPreferences object. The code for both cases is relatively similar, because both attempt communication with the database to extract various events, but the difference is which events they try to extract.

In case the value of displayType is “All”, the database request is to extract all events currently present in the database. The next step is invoking the execute() method of DB\_GetAllEvents, along with the corresponding link to the PHP (<http://carmeets.eu/view_all_events.php?>). Afterwards, a ProgressDialog is created, along with a Timer, in order to create a loading animation, until the communication process with the database is completed, so that the user can be notified when the process is ongoing and when it is completed. In case the “res” string variable is not an empty string, i.e. not “”, the ProgressDialog is cancelled. Then the response from the database is being separated into different pieces using “#” as delimiter, and stored in the data\_array array. Then a “for” loop is created, iterating through all records in the data\_array array and calling the createButton function for each record in the array, being passed as a parameter for the function. When the loop is completed, the timer is also cancelled. In case the response message from the database is an empty string, i.e. its length is 0, the user is notified that there was an error extracting the data.

In case the value of displayType is “Suggested”, the database request is also to request all events currently present in the database, but they are to be filtered later. The next step is invoking the execute() method of DB\_GetAllEvents, along with the corresponding link to the PHP (<http://carmeets.eu/view_all_events.php?>). Afterwards, a ProgressDialog is created, along with a Timer, in order to create a loading animation, until the communication process with the database is completed, so that the user can be notified when the process is ongoing and when it is completed. In case the “res” string variable is not an empty string, i.e. not “”, the ProgressDialog is cancelled. Then the response from the database is being separated into different pieces using “#” as delimiter, and stored in the data\_array string array. Then a “for” loop is created, iterating through all record in the data\_array array one by one. Afterwards, a string array is created, called “separated\_event”, which hold the data about each event, after it was split using “&” as delimiter, i.e. separating each piece of data, belonging to one event. Then there is an “if” check performed, evaluation whether the location, preferences/restrictions or availability/starting time match. In order to do this, the three suggestion function are called – location\_check, preferences\_check and availability\_check. Since the “if” statement is constructed with “||” (OR), other than && (AND), only one of the conditions has to be true in order to display a specific event. This reason to do it this way is because the database currently does not have a huge amount of events added and for demonstration purposes this kind of evaluation will have a lower chance failing to display events at all. While the correct to do it is to have the || replaced with &&, in order to provide the user with the perfect events for his or her preferences, location and availability, this version also provides the suggestion functionality, although not providing the perfect matches. In case the “if” statement evaluates to true, the createButton function is called for the corresponding event, meaning this event will be displayed to the user. Afterwards, the timer is also cancelled. In case the response from the database was an empty string, i.e. its length was 0, the user is notified that there was an error extracting the data.

The location\_check Boolean function is responsible for evaluating the event location and the user’s location. It takes two strings as parameters, event location and user location. Then a simple “if” statement is created, evaluating the value of both string. The statement was designed with the idea that either the user or the event creator might have written down the location with lower case or upper case, so in order to avoid this issue, both the user location and the event location are transformed to lower cases, so that they might be identical and the issue of having identical location, but one with lowercase letters and one with upper case letter(s), could be resolved.

The availability\_check Boolean function is slightly more complex than the location\_check function. It is responsible for checking whether the event starting time matches with the user’s availability. While the check performed is very similar to the location\_check function, the problem is the different formats of the user’s availability and the event’s starting time. In order to make them possible to be evaluated, both the availability value has to be manipulated a little. The function takes as parameters the event starting time and the user’s availability as strings. The user’s availability value is being split twice. First time it being split using “-“ as delimiter, since the format is HH:MM – HH:MM (hour:minutes – hour:mintes). The second time it is being split is, the “:” is being used as delimiter upon the first part of the availability, i.e. the starting time, in order to obtain the starting time of the availability of the user. The event time is being split once, in order to separate it into hour and minutes. Then there are two integer variables for each time (one for event and one for user), storing data about the hour and minutes. Since the split values are in String format, while assigning the value to the integer variables, are parse into Integer using Integer.parseInt. The “if” statement makes sure that the event starting time is in the range of the user’s availability. In case the evaluation is true, the function returns true and if it is not true, the function returns false.

The preferences\_check Boolean function checks whether the preferences of the user matches with the restrictions of the event. The function takes two string parameters – event restrictions and user preferences. Both the user preferences and the event restrictions are being trimmed, in order to remove the spaces and afterwards split using “,” as delimiter. Afterwards, there is a “for” nested in another “for” loop, in order to compare each event restrictions with each user’s preference. For each event restriction, all separate user preferences are evaluated and if there is a match in an event restriction and user preference, the function returns true, otherwise the function returns false.

The createButton function is responsible for the dynamic creation of button. It takes one string as a parameter, which is supposed to contain the whole event information. Then the parameter is split into str\_array string array, using “&” as delimiter, in order make every different event information into a different entry in the array. Then there are seven String variables, initialized with the array’s elements. There Strings store data about event name, event location, event starting time, event duration, event creator and event creation time. Then there is an instance of Button created, following by setting its text value, which appends all those Strings, along with the required “label text” to the button’s text value. The end result is the button itself displaying the event information inside. Afterwards, an ID for the button is being set. In order to do it, a use of the global integer variable “num” is performed. It is first initialized to 0 when the activity is created and with each button creation, the value of this variable is incremented by one, so that there will not be an identical button ID. After the ID is set, an instance of the layout is created, in this case LinearLayout. Then this instance is used to add the button to the layout and essentially create this button. While the button will be visible from this point onward, it will not be functional, since it does not have an onClickListener. Therefore, an onClickListener is created. What it does is creating an instance of sharedPreferences for the event name, along with an editor. Afterwards, the String value for event name is added to that sharedPreferences object and the apply() method is called, in order to save it. After everything is done, the user is transferred to the EventInformation activity.

The EventInformation activity is the last activity in the application. It is invoked when the user chooses and event from the dynamically generated buttons, corresponding to specific events in the ViewEvents activity. The goal of this activity is to provide the user will all information about a specific event in a well-structured way. It is also allowing the user to sign up for an event by clicking the only the button in the activity – the Sign Up button.

The activity invokes database connection at least once in the onCreate method, but there is also option to communicate second time with the database when the user wants to sign up for a particular event. The onCreate database connection is used to extract the event information and populate the text views with the appropriate information.

Since there are possible two times when a database connection has to invoked and the response saved, there are two sets of global variables, responsible for storing data about the database response. The first global String variable is called “res”, along with “res1” and “res1l”. The “res1” is responsible for storing the “res” variable’s length. The second global String variable is called “data”, along with “data1” and “data1l”. The “data1l” is responsible for storing the “data” variable’s length.

The onCreate method starts with a declaration of a String array called “users”, which will be storing data about users attending this particular event. This data will be extracted from the database at certain point.

There are also eight labels in the activity, responsible for displaying information about the event name, event location, event restrictions, event starting time, event duration, event creator, event’s date of creation and the users that are attending the event. There is also a button, which allows the user to sign up for the particular event.

There are two instances of sharedPreferences – the first one is responsible for providing the event name, which was added by clicking a certain button about specific event the user wants to see from the ViewEvents activity and the other instance is responsible for providing information about the user’s username. The username will later be used to perform a check whether the user has already signed up for the particular event.

The next instruction in the onCreate method, after creating the sharedPreferences instances is accessing the database information about the particular event. In order to achieve this, execute() method is called on DB\_EventInformation class, among with the required parameters needed, in order for it to execute properly. Those parameters are the link to the PHP, located on the web server, which is <http://carmeet.eu/view_specific_event.php>? and the event name, obtained through the sharedPreferences object. Afterwards, a ProgressDialog is created, along with a timer. This is needed in order to notify the user that the application is currently communicating with the database. While the communication is ongoing, the user will see a loading animation on top of the activity. When the communication with the database is done, this animation will disappear and the required fields will be populated with the response data from the database. If the “res” variable, responsible for storing the response from the database is not an empty string, i.e. not “”, it is being split using “&” as a delimiter and populated the newly created data\_array string array. Afterwards, the data stored in the array is assigned to the appropriate labels by appending them to a hardcoded text, letting the user know what the information is about. Also, the users string array is being populated with the eighth element in the data\_array, which is the users attending the event. Afterwards, the timer is also cancelled.

In case the response from the server was an empty string, i.e. its length is 0, the user is notified that there was an error extracting the data.

There is also a function checking whether the user has attended an event or not. This is a Boolean function and is called checkAttending. It takes two parameter – one for all the users attending and one for the username of the user. A new array is initialized, called user\_array, which takes the attending users string and splits it using “,” as delimiter, storing the username of each individual user that is attending. Then there is a relatively simple “for” loop, iterating through the whole array, i.e. going over all possible users that are attending, and comparing this value to user’s username value. In case they match, the function will return false, meaning the user’s username matches a username that is already present in the attending users. In case there is not match, the function will return true, meaning the user’s username was not located in the already attending users.

The function signup is responsible to establishing connection with the database and updating the attending users with the new list of attending users, where the user’s username is added to the list. The function takes two string parameters, one for event name and one for a list of users attending. The first thing that the function does is to call the execute() method on DB\_EventSignUp class, passing the link to the PHP – <http://carmeet.eu/event_sign_up.php>?. Afterwards, a ProgressDialog is created along with a Timer, in order to notify the user when the application is communication with the database. It is essentially playing a loading animation while communication is ongoing and when it is done, the animation disappears and the user can continue interacting with the application. If the “data” string, responsible for storing the data returned from the database is not empty, i.e. not “”, the Progress Dialog is cancelled. There is also a check, making sure that the response string’s length is not equal to 0, and if that check returns true, the user is notified that he or she has successfully signed up for the event. If the response string’s length is 0, the user is notified that the event sign up as failed.

The onClickListener for the Sign Up button is responsible for making the only button in the activity functional. It creates a new string called newAttending. Afterwards, there is an “if” statement, calling the checkAttending function along with the two parameters it takes – the users attending string, obtained earlier through the PHP response for the event information, and the user’s username. In case the checkAttending function return true, the newAttending variable is given the value of the users attending list, appended with the user’s username. Afterwards, the singup function is called, giving it the values of the event’s name and the new list of attending users. If the function checkAttending returns false, the user is notified that he or she has already been signed up.

There are also eleven classes, responsible for establishing the connection with the database and extracting specific data. All of them deal with different requests towards the database. Technically, there are thirteen such classes, but two of them are the DB\_CarMake and DB\_CarModel, which are currently not used, but they will be used in future development. Their purpose was to populate two spinners with all possible car makes and models, so the user can use the Spinner to select his or her car make and model instead of typing it by himself or herself, similar to what a lot of car sale websites are using.

The DB\_CreateEvent class is responsible for making a request to the database, in order to create a new event. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are seven strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php, the event name, the event location, the event restrictions, the event time, the event duration and the event creator. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “create” variable in the CreateEvent activity.

The DB\_EmailCheck class is responsible for making a request to the database, in order to check if the user entered email is present in the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are two strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php and the email value. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “res” variable in the ForgottenPassword activity.

The DB\_EventInformation class is responsible for making a request to the database, in order to extract information about an event from the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are two strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php and the event name. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “res” variable in the EventInformation activity.

The DB\_EventSignUp class is responsible for making a request to the database, in order to sign up a user to an event from the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are three strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php, the event name and the user’s username. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “data” variable in the EventInformation activity.

The DB\_GetAllEvents class is responsible for making a request to the database, in order to extract information about all events from the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There is one string, designated to store the data that the PHP file will require, in order to operate properly. This string is storing data about the link to the php. Initially, this string are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “res” variable in the ViewEvents activity.

The DB\_GetProfileInfo class is responsible for making a request to the database, in order to obtain the user’s profile information from the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are two strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php and the user’s username. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “data” variable in the ProfileEdit activity.

The DB\_Login class is responsible for making a request to the database, in order to verify the user’s credentials with the credentials, corresponding to this user from the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are three strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php, the user’s username and the user’s password. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “log” variable in the Login activity.

The DB\_Password class is responsible for making a request to the database, in order to change the existing password in the database for the corresponding user with the new one that he or she has chosen. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are three strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php, the user’s username and the user’s new password. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “res” variable in the ChangePassword activity.

The DB\_ProfileData class is responsible for making a request to the database, in order to the user’s profile data from the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are two strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php, and the user’s username. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “data” variable in the Profile activity.

The DB\_ProfileUpdate class is responsible for making a request to the database, in order update the user’s profile info, located in the database, with the new, edited information that he or she has chosen. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are nine strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php, user’s username, user’s location, user’s age, user’s preferences, user’s availability, user’s car info, user’s gender and the user’s email. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “res” variable in the ProfileEdit activity.

The DB\_Register class is responsible for making a request to the database, in order to insert (register) a new user into the database. It makes use of the Asynchronous Task class, therefore it extends AsyncTask. There are ten strings, designated to store the data that the PHP file will require, in order to operate properly. Those strings are storing data about the link to the php,the user’s username, the user’s password, the user’s location, the user’s preferences, the user’s availability, the user’s car info, the user’s gender, the user’s age and the user’s email. Initially, those strings are being initialized as empty (“”). They are being assigned with values in the class’ constructor. The constructor requires values for each string when it is called. The doInBackground function is a mandatory function, in case a class extends ASyncTask. It is taking care of the processes that are being executed in the background, while in the case of this application the foreground is busy displaying the loading application. These processes are not visible by the user. The first thing that this function does is to construct the “data” string variable. When it is completely constructed, this variable stores data about what must be passed through the POST method to the PHP file. The string is encoded in the UTF8 standard. Afterwards, an URL variable is created, which is essentially the link and the data appended together. There is also an instance of URLConnection class, which opens the connection with the PHP using the URL variable. An OutputStreamWriter is used to append the data variable to the request’s body, so that the POST method can obtain the information it needs. A bufferedReader object is also created, so that it can obtain the information that the PHP file returns. A string variable “line” is created and is initially assigned to null, i.e. empty. This variable is then assigned the value of the PHP’s response in the “while” loop and appending its value, to the “sb” string variable, using the append method. This action is performed while there is still data in the “line” variable. Once this process is complete, the “sb” variable already stored all data from the PHP response and is then assigned to the “reg” variable in the Register activity.

There are also two classes used for user input data validation and one more class, responsible for hashing the password, which the user has entered.

The Login\_Input\_Check contains two function, dealing only with the login process of the user. It has two Boolean functions – username\_Check and password\_Check. Both of them take as parameter one Edit Text and one TextView. the Edit Text is the text box where the user has entered his or her username, while the Text View is the error label, in case there was an error with the data he or she has inputted. The function basically checks the length of the data, entered in the Edit Text and if it is 0, i.e. the field is empty, the error label’s text property is set to “Username field is empty!” and the error label’s visibility property is set to VISIBLE. Afterwards, the function returns false. The password\_Check function works the same way. It takes two parameters – one Edit Text and one Text View. The Edit Text is the text box where the user is supposed to enter his or her password, while the Edit Text is the error label for the password field. The function checks the length of the data entered in the text box and if it is 0, the error label’s text property is set to “Password field is empty!” and its visibility property is set to VISIBLE. Afterwards, the function return false. For both functions, if the entered data’s length of both corresponding Edit Texts is not 0, they will simply return true. There is nothing else they need to do, because if the user has inputted some data in the fields, this data is sent to the corresponding PHP to check in the database whether the records match.

The Register\_Input\_Checks contains several function, checking the information that the user has entered in during the registration process. It also contains a constant, which is assigned the value of a regular expression, in order to check if the email format, entered by the user, is valid.

The username\_Check function takes one Edit Text and one Text View as parameters. The Edit Text is the text box where the user is supposed to enter his or her username. A string value is initialized with the value of the Edit Text’s text value, so it can be used easier and make its usage more understandable. Since the username has to be at least four characters long and, of course, not empty, there are two check that are being performed. The first check is to make sure that the username field was not left empty. There is an “if” statement, checking whether the length of the string variable’s value is equal to zero. If it is zero, the error label’s text value is set to “Empty username field!” and its visibility value is set to VISIBLE. Afterwards, the function will return false. After making sure that the username field is not empty, there is an “if” statement whether the length of the username, that was entered by the user is more than four. If the entered username is less than four characters long, the error label’s text value will be set to “Username must be at least 4 symbols long!” and its visibility status will be set to VISIBLE. Afterwards, the function will return false. In case the user has entered a username, which is longer than four characters, the function will return true, since the “if” statements’ conditions will not be true, therefore they will not return false.

There is also a general function, checking for an empty field – it is called emptyField\_Check. It takes two parameters, one Edit Text and one Text View. The Edit Text is the text box, where the user is supposed to enter information, while the Text View is the error label, corresponding to the text box. The value from the text box is saved in a String variable, called str\_field. An “if” statement is performed, checking whether the length of the str\_field variable’s value is equal to zero. In case this conditions is true, i.e. the text box is left empty, the error label’s text value will be set to “Empty Field!” and its visibility value will be set to VISIBLE. Afterwards, the function will return false. In case the “if” condition is not true, the error label’s visibility value is set to INVISIBLE, in case it was set to visible in a check before, and the function returns true, meaning here are no errors with the user input for that text field.

Another checking function is the password\_Check. It takes two parameters, one Edit Text and one Text View. The Edit Text is the text box, where the user is supposed to enter his or her password information, while the Text View is the error label, corresponding to the password text box. The value from the text box is saved in a String variable, called pass. There is an “if” statement, checking whether the length of the pass variable’s length is zero. In case it is zero, i.e. the password field was left empty, the error label’s text value is set to “Password field is empty!” and its visibility status is set to VISIBLE. Afterwards, the function return false. In case the password field has some entered data, the function will not return false, because it will skip the execution of the first “if” statement. Instead, the second “if” statement will be checked. The condition in this “if” statement is checking whether the length of the pass variable is more than four. In case the value entered in the password field is between and one and four, the error label’s text value will be set to “Password should be at least 4 symbols!” and its visibility is set to VISIBLE. Afterwards the function return false. In case this conditions is not met either, i.e. both conditions are false – the data entered in the password field is more than four symbols, the function will return true, meaning there was no problem with the data, and the error label’s visibility status will be set to INVISIBLE, in case it was set to VISIBLE before.

The location\_Check function makes sure that the field where the user is supposed to enter location is not empty. It takes two parameters – one Edit Text and one Text View. The Edit Text is the text box where the user is supposed to enter the location and the Text View is the error label, corresponding to the text box. A String variable is created, called loc, which is assigned the text value of the Edit Text. The creation of the variable is followed by an “if” statement, checking whether the loc variable length is zero, i.e. the location text box is left empty. In case it is left empty, the error label’s text value is set to “Location field is empty!” and its visibility status is set to VISIBLE. Afterwards, the function returns false. In case the length of the location is more than zero, i.e. the user has entered a value, the “if” statement condition is false, therefore the function will set the error label’s visibility value to INVISIBLE (in case it was set to VISIBLE before) and will return true, meaning there were no errors with the location text box.

The age\_Check function is responsible for making sure that the user has entered a valid integer value in the age field. The function takes two parameter – one Edit Text, which is the text box where the user is supposed to enter the age value, and one Text View, which is the error label for the corresponding text box. First off, the function checks the length of the string value that was entered in the text box through an “if” statement. In case the length is 0, the error label’s text value is set to “Age field is empty” and its visibility value is set to VISIBILE. Afterwards, the function return false. In case the length is more than 0, the function performs a few other checks, in order to ensure that the value is valid. First, the String value is assigned to an integer variable. Everything is enclosed in a try catch statement. In case the assigned to the integer, the catch statement will be executed, since the assignment will fail, i.e. the user has entered a string value, not an integer one. When the catch statement is executed, the error label’s text value is set to “Age must be a number!” and its visibility value is set to VISIBLE. Afterwards, the function return false. In case the assignment has succeeded, i.e. the entered value was in fact a number, an “if” statement is created in order to check if the value is more than 0 and less than 120. If this conditions turns out to be true, the error label’s visibility value is set to INVISIBLE, in case it was set to VISIBLE before, and the function true. If the entered value is not within this range (1-120), the error label’s text value will be set to “Age must be between 1 and 120!” and its visibility value will be set to VISIBLE. Afterwards, the function will return false.

The pref\_Check function is responsible for making sure that the user has entered a value in the preferences text box. It takes two parameters – one Edit Text and one Text View. The Edit Text is the text box, where is supposed to enter preferences information, and the Text View is the error label for the corresponding text box (the preferences text box). If the length of the value, which the user has entered is more than 0, i.e. the text box was not left empty, the function will set the error label’s visibility value to INVISIBLE, in case it was set to VISIBLE before, and the function with return true. Otherwise, if the length is 0, i.e. the preferences field is left empty, the error label’s text value will be set to “Preferences field is empty!” and its visibility value will be set to VISIBLE. Afterwards, the function will return false.

The avail\_Check function is responsible for making sure that the user entered a valid value in the availability text box. It takes two parameters – one Edit Text and one Text View. The Edit Text is the text box, where the user is supposed to enter availability information, and the Text View is the error label for the corresponding availability text box. First, the function check whether the field is empty or not by evaluation whether the length of the availability text box’s value is equal to 0. In case it is 0, the error label’s text value will be set to “Availability field is empty!” and its visibility property will be set to VISIBLE. Afterwards, the function will return false. In case the availability field is not empty, i.e. the user has entered some value, an “if” statement is called to check the format of the user inputted value. In order to make sure that the value format is correct, the “if” statement’s condition checks whether the value matches the regular expression - ^([0-9]|0[0-9]|1?[0-9]|2[0-3]):[0-5][0-9]-([0-9]|0[0-9]|1?[0-9]|2[0-3]):[0-5][0-9]$ . This regular expression matches the format HH:MM-HH:MM. For example, a valid format would be 19:30-23:30, while an invalid format would be 10-23:30. The conditions has “!” before the evaluation, in order to make sure that if this regular expression does not match the user input value, the code should be executed. In case there is a mismatch, the error label’s text value will be set to “Invalid time format (21:00-23:30)” and its visibility status will be set to VISIBILE. Afterwards, the function will return false. In case both of the “if” statement are not executed, i.e. the user input is correct, the error label’s visibility status will be set to INVISIBLE, in case it was set to visible before, and the function will return, meaning there were no problems with the user input.

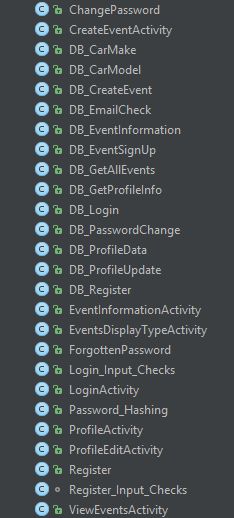
The car\_Check function checks whether the car info text field was not left empty. The function takes two parameters – one Edit Text and one Text View. The Edit Text is the text field where the user is supposed to write down information about his or her car and the Text View is the error label for the corresponding car info text field. An “if” statement takes the length of the user input in the text field and if it is 0, the error label’s text value is set to “Car info field is empty!” and its visibility status is set to VISIBLE. Afterwards, the function return false. In case the car info was not left empty, i.e. the length of the text field’s value is more than 0, the error label’s visibility value is set to INVISIBLE, in case it was set to VISIBLE before and the function returns true, meaning there were no errors in the this text field.

The duration\_Check function checks whether the event duration value that was entered by the user is valid or that there is actually some value entered by the user. The function takes two parameters – one Edit Text and one Text View. The Edit Text is the text field where the user is supposed to enter value about the event’s duration and the Text View is the error label for the corresponding event duration text field. The function first checks the length of the value entered in the text field using an “if” statement. In case the length is 0, i.e. the user has not entered anything in the field, the error label’s text value will set to “Duration field is empty!” and its visibility will be set to VISIBLE. Afterwards, the function will return false. In case the user has entered some value, a try-catch statement is invoked, in order to make sure that the value entered by the user is an integer. The text field’s text value is assigned to an integer in the “try” part. In case there was an error parsing the value to an integer variable, the “catch” part is executed. It sets the error label’s text value to “Enter integer number!” and its visibility value is set to VISIBLE. Afterwards, the function returns false. In case both checks are passed successfully, i.e. there was no error with the data in the text field, the function will set error label’s visibility value to INVISIBLE, in case it was set to VISIBLE before, and return true.

The startTime\_Check functions checks whether the value entered in the event starting time text field is valid, if there is any. It takes two parameters – one Edit Text and one Text View. The Edit Text is the starting time text field, where the user is supposed to enter starting time value. The Text View is the error label, corresponding for the starting time text field. First of all, the function checks whether there is any value at all in the text field. In order to that, an “if” statement is invoked checking whether the length of the text field’s text value is 0. In case it is 0, the error label’s text value is set to “Starting time field is empty!” and its visibility value is set to VISIBLE. Afterwards, the function return false. In order to check whether the format is correct, a regular expression is used. It is similar to the one for the availability check, but slightly modified to work with starting time format - ^([0-9]|0[0-9]|1?[0-9]|2[0-3]):[0-5][0-9]$. In case the value entered by the user does not match this regular expression, the error label’s text value is set to “Invalid time format (21:00)” and its visibility value is set to VISIBLE. Afterwards, the function returns false. In case both “if” statement are not executed, meaning there was no error with the data entered, the error label’s visibility value will be set to INVISIBLE, in case it was set to VISIBLE before, and the function will return true.

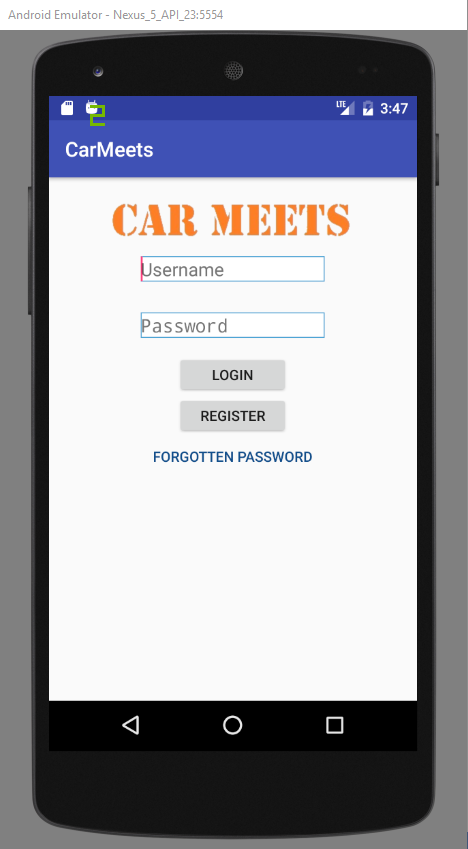
The checkEmail function is responsible for checking whether the format of the string that the user has inputted in the email field is valid, if any. The function takes two parameters – one Edit Text and one Text View. The Edit Text is the field where the user is supposed to enter email data and the Text View is the error label, corresponding to the email text field. The function first checks whether there is any value entered in the field at all by evaluating whether the length of the text value is equal to 0. If it is 0, i.e. the user has not entered any data, the error label’s text value is set to “Email field is empty!” and its visibility status is set to VISIBLE. Afterwards, the function return false. In case there is data entered in the text field, the function checks whether the format of the inputted data is valid for an email. In order to do this, the function evaluated whether the data matches the regular expression for an email - [a-zA-Z0-9\+\.\\_\%\-\+]{1,256}" + "\\@" + "[a-zA-Z0-9][a-zA-Z0-9\\-]{0,64}" + "(" + "\\." + "[a-zA-Z0-9][a-zA-Z0-9\\-]{0,25}" + ")+. In case it does not match, the error label’s text value is set to “Wrong email format!” and its visibility status is set to VISIBLE. Afterwards, the function returns false. In case there were no errors in the data, i.e. the data is entered correctly and both “if” statements’ code was not executed, the function sets the error label’s visibility value to INVISIBLE, in case it was set to VISIBLE before and returns true.

The HashPassword class is responsible for the password encryption in md5 format. It contains only one method inside, called md5, which returns the encrypted password as a string. The method requires one parameter to be passed to it – toEncrypt (the password that needs encryption). Then a MessageDigest object is initialized along with the string to encrypt. The MessageDigest is used to provide a secure one-way hashing. The update() function takes the string and iterates over it until it goes through the whole string and encrypts it in a byte format. Afterwards, a byte array is created, which is used to hold the data with the encrypted password. After this process, a StringBuilder is going through the array and adding the data to a single string, which is afterwards transformed to lowercase letters using the toLowerCase() method.



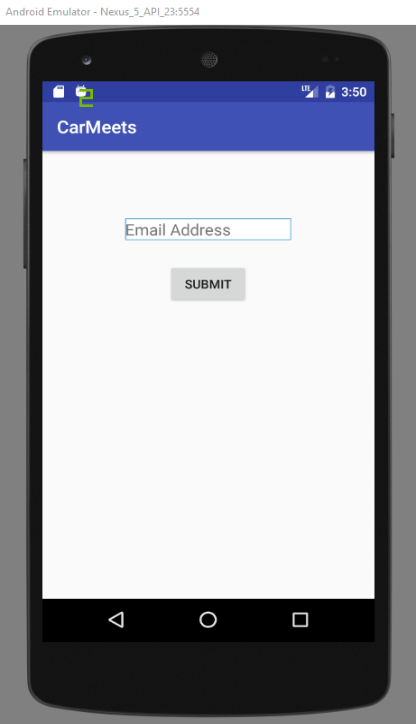
Android Application Classes

The user interface of the application is relatively simple and intuitive. Upon starting the application, the user is faced with two textboxes with three buttons and one Image, holding the application’s logo, which can be seen in various activities, located in the Login Activity. The textboxes do not have labels next to them, notifying the user what he or she is expected to type, but instead have the “Hint” attribute assigned the appropriate value. The hint attribute fills the textbox with text, which has slightly lighter font color than the normal input font color. The text persists in the textbox until the user enters a symbol. After the entry, the “hint” text disappears and reappears if the user clears the textbox of input. The values of the “hint” attribute are “Username” and “Password”, respectively, notifying the user what he or she is expected to write there. The buttons are responsible for logging into the system or transitioning to the ForgottenPassword activity. The Login button and the Register button hav a standard android button styling and having the text attribute to “Log In” and “Register”, respectively. The ForgottenPassword button, on the other hand, does not like a button at all. Its border is set to transparent by the android:background attribute for this object, located in activity\_login.xml. The XML code for the user interface of the activity can be found in activity\_login.xml. The end result is having the button look like a hyperlink, which is familiar to the majority of the people using the internet.



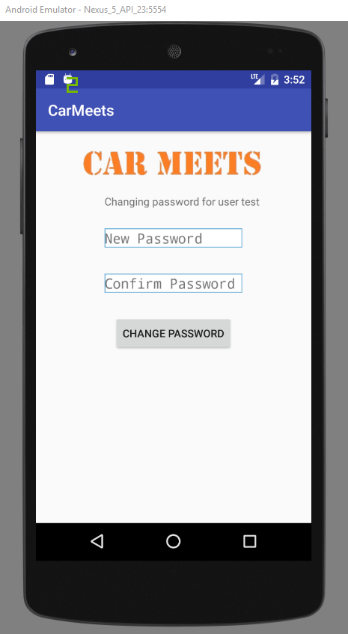
Login activity user interface

Upon clicking the Forgotten Password button, the user is redirected to ForgottenPassword activity. The user interface in this activity is similar to the one in the Login activity’s interface. The difference is that this activity contains only one textbox and one button. The layout code is located in activity\_forgotten\_password.xml. The textbox also makes use of the “hint” attribute instead of using labels. The “hint” value is “Email Address”, prompting the user to enter his or her email address there. The button has a text value “Submit”. It checks whether the email entered is present in the database and if it is, the user will be sent to the ChangePassword activity.



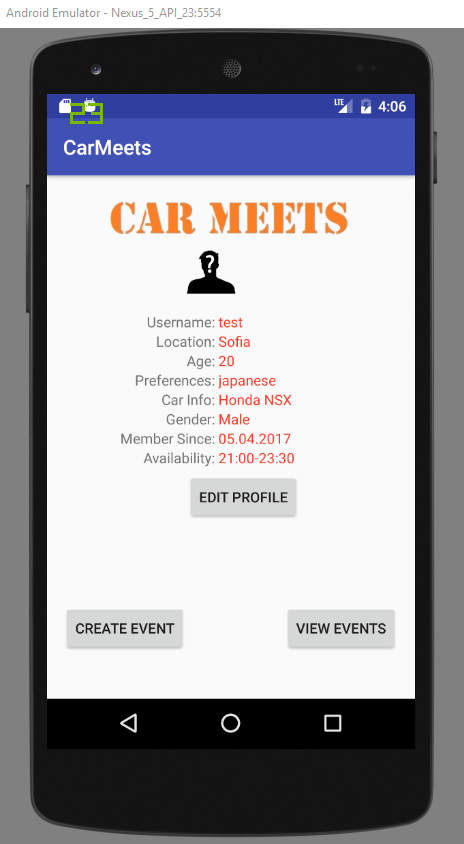
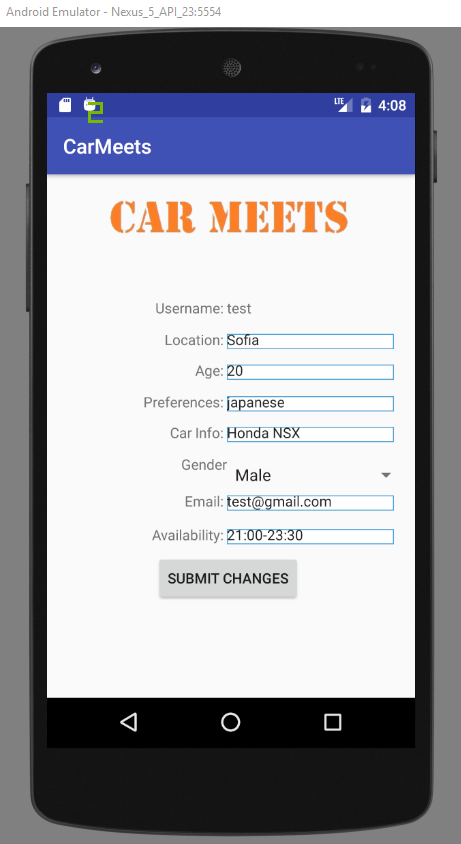
ForgottenPassword activity user interface

The ChangePassword activity contains two textboxes and one button. Both textboxes make use of the “hint” attribute with values “New Password” and “Confirm Password”. The button has a text value “Change Password” and if the user entered correct output, by clicking this button he or she can change his or her password. It also has a label displaying the username, for which the password will be changed.

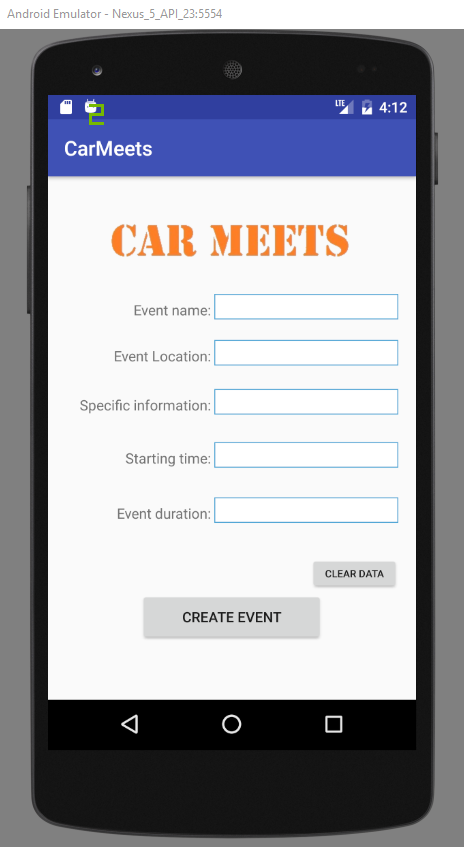


ChangePassword activity user interface

Upon entering valid credentials and clicking the Login button, the user is transferred to the Profile activity. This activity displays all the information about the current user, along with three button – Edit Profile, Create Event and View Events. It also has a profile picture field, but it is currently not changeable. Upon clicking the Edit Profile button, the user is redirected to the ProfileEdit activity. This activity allows the user to change details about his or her profile. The information is automatically populated with the previously entered details by the users, extracted from the database. There is also a button to submit changes, with a text value Submit Changes. Upon clicking this button and if there are no errors with the data, the user is transferred back to the Profile activity.

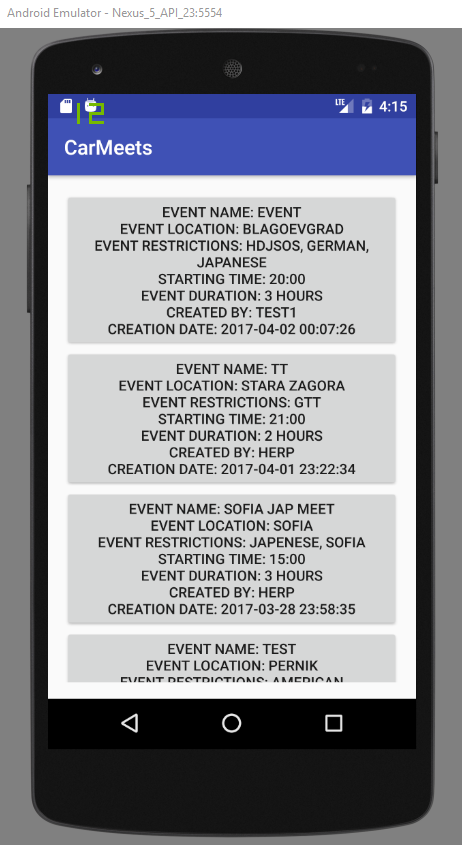
 

Profile activity user interface ProfileEdit activity user interface

Upon clicking the Create Event button, the user is transferred to the CreateEvent activity. This activities has various text fields and labels, in order to allow the user to create an event and let him or her know what is he or she expected to enter in each field. There are also two button – Clear Data and Create Event. Upon clicking the Clear Data button, everything written down in the fields will be erased. If the Create Event button is clicked and there are no errors with the data, the user will be transferred back to the Profile activity. 

CreateEvent activity user interface

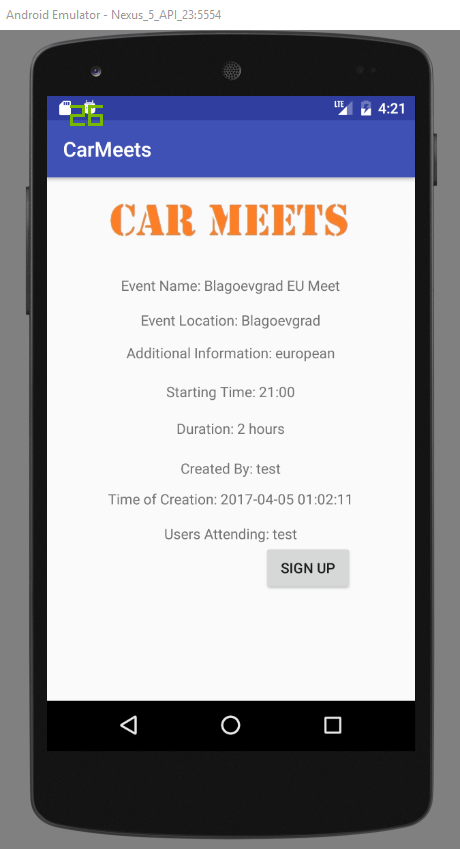
Upon clicking the View Events button, the user is faced with the choice to either view All events or the Suggested events only. This choice is present in the EventDisplayType activity. This activity has only two button – View All Events and View Recommended Events. Upon choosing an option, the user is redirected to ViewEvents activity. Essentially, this activity is blank, it does not have any buttons whatsoever. Instead they are generate dynamically, based on how many records are found to be display. The generated controls are button with text value set to display information about the events.

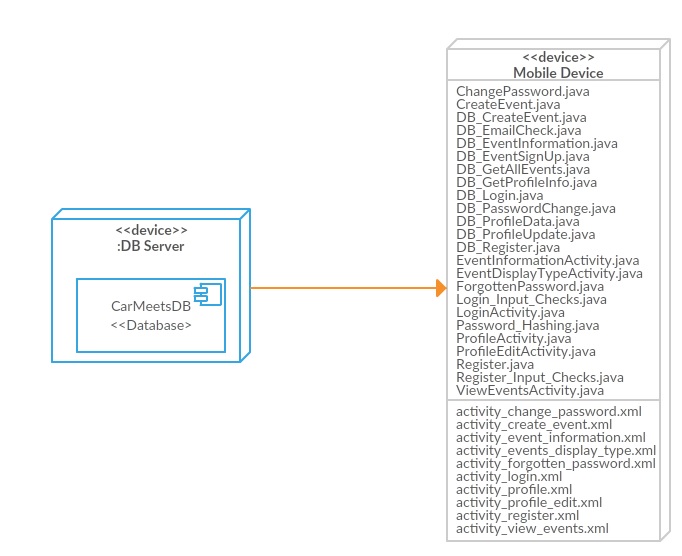
EventDispalyType activity ViewEvents activity user interface

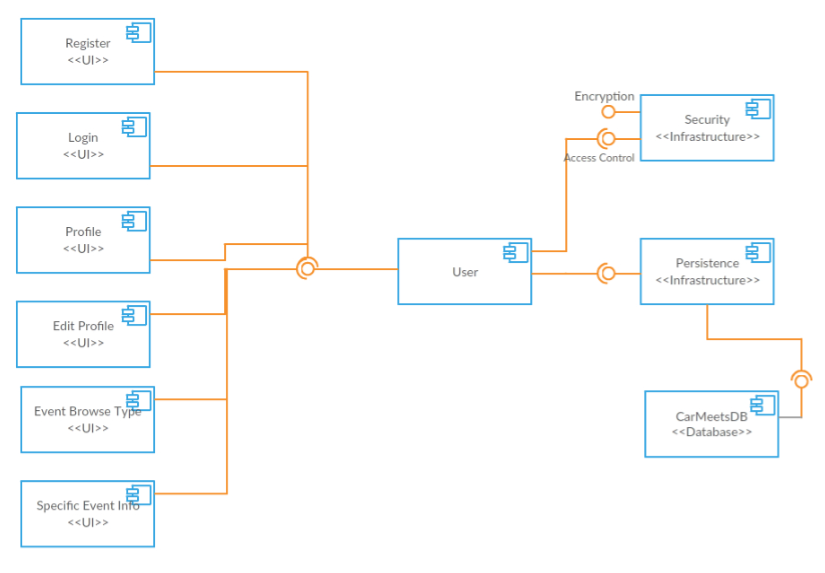
user interface

Upon clicking on a specific events, the user is redirected to EventInformation activity. This activity contains all the information present in the database about a specific event and is displayed to the user. It also has a button, allowing the user to sign up for an event, having a text value “Sign Up”.



EventInformation activity user interface

CarMeets Application Deployment Diagram



CarMeets Application Component Diagram

**PHP Files Database Queries**

**create\_event.php** - "INSERT INTO Event (e\_name, e\_location, e\_restrictions, e\_time, e\_duration, e\_createdBy, users\_attending)

VALUES ('$e\_name', '$e\_location', '$e\_restrictions', '$e\_time', '$e\_duration', '$e\_createdBy', '$e\_createdBy')"

**event\_sign\_up.php** - "UPDATE Event SET users\_attending='$users\_attending'

WHERE e\_name='$eventName'"

**get\_profile\_data.php** - "SELECT \* FROM Users where

username='$username'"

**login.php** - "SELECT \* FROM Users where

username='$username' and password='$password'"

**mail\_check.php** - "SELECT \* FROM Users where

email = '$email'"

**password\_change.php** - "UPDATE Users SET password='$password'

WHERE username='$username'"

**profile.php** - "SELECT \* FROM Users WHERE username = '$username'"

**reg.php** - "INSERT INTO Users (username, password, location, age, preferences, availability, car\_info, gender, email)

VALUES ('$username', '$password', '$location', '$age', '$preferences', '$availability', '$car\_info', '$gender', '$email')"

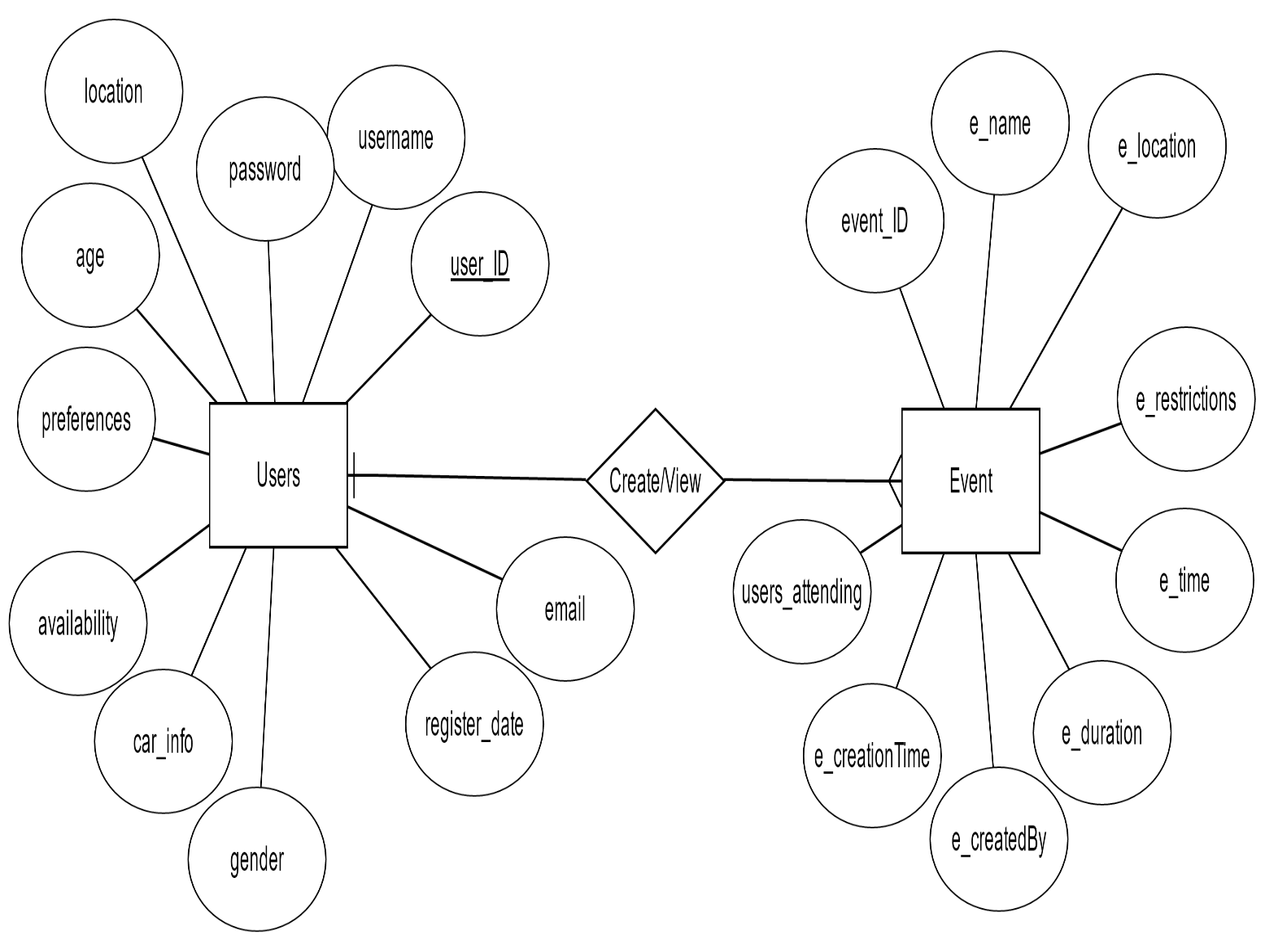
**update\_profile.php** - "UPDATE Users

SET location='$location', age='$age', preferences='$preferences', availability='$availability', car\_info='$car\_info', gender='$gender', email='$email'

WHERE username='$username'"

**view\_all\_event.php** - "SELECT \* FROM Event"

**view\_specific\_event.php** - "SELECT \* FROM Event WHERE e\_name='$eventName'"



CarMeets Application Database ER Diagram

**Implementation**

The application is designed to be useable on any mobile device, running Android operating system, such as mobile phones or tablets. This allows the user to be able to use the full functionality of the application, as long as he or she has internet connection or his or her device. In the future, a web application might be developer, to allow the user to register, edit profile, view events and create events, but at this point only the mobile application is present.

The application has been developed using only Android Studio. The programming language is Java.

External libraries and frameworks have not been used. However, a few build-in libraries have been used in both android and desktop applications.

Specific libraries used:

* App.ProgressDialog – provides the methods for creating a progress dialog, which is used as a loading screen in the android application.
* Content.SharedPreferences – provides methods to save some data in the phone’s internal database, which can be used later.
* Util.Timer – provides methods for using a timer, which can be used for scheduling tasks for future use or, as in the case with the application, keeping the progress dialog active for as long as needed.
* Util.TimerTask – provides methods for scheduling a new timer task.
* Text.SimpleDateFormat – provides methods for formatting and parsing dates in various ways.
* Util.Calendar – provides methods for converting between a specific instant in time and a set of calendar fields (year, month, day, hour)
* Os.AsyncTask – provides methods for executing certain tasks in a background thread while the foreground is free to execute other methods.
* Io.BufferedReader – provides methods for reading characters from input stream, buffering the characters in order to provide an efficient way of reading characters, arrays and lines.
* Io.InputStreamReader – provides methods that serve as a bridge between byte streams and character streams.
* Io.OutputStreamWriter – provides methods that are identical to the InputStreamReader’s (bridge between byte streams and character streams), but for writing data.
* Net.URL – provides methods for the use of the Uniform Resource Locator, which is a pointed to a resource on the World Wide Web.
* Net.URLConnection – provides methods for communication between the application and an URL.
* Net.URLEncoder – provides methods for representing a string into an html-recognizable format.
* Security.MessageDigest – provides methods for secure one-way data hashing.
* Content.DialogInterface – provides methods responsible for the interaction between the user and the dialog messages.
* App.AlertDialog – provides methods for displaying and handling alert dialogs. Alert dialogs can be used in numerous forms, depending on the buttons required.

In order to install the Android application on a mobile device, the user must install the “app-debug.apk” file, located in the “Android install” folder.

Code fragments:

btn\_create.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 //the Create button first invokes the methods from Register\_Input\_Checks class  
 //such as emptyField\_Check, starting\_time, duration\_Check, in order to validate  
 //the user input, afterwards, if there are not errors in the input  
 //the createEvent function is called to update the database and  
 //the user is transfered to the Profile activity  
 check.emptyField\_Check(txt\_eventName, lbl\_eventNameError);  
 if (!check.emptyField\_Check(txt\_eventName, lbl\_eventNameError))  
 isCorrect[0] = false;  
 check.emptyField\_Check(txt\_eventLocation, lbl\_eventLocationError);  
 if (!check.emptyField\_Check(txt\_eventName, lbl\_eventNameError))  
 isCorrect[0] = false;  
 check.emptyField\_Check(txt\_specificInformation, lbl\_specificInformationError);  
 if (!check.emptyField\_Check(txt\_specificInformation, lbl\_specificInformationError))  
 isCorrect[0] = false;  
 check.starting\_Time(txt\_eventStartingTime, lbl\_eventStartingTimeError);  
 if (!check.starting\_Time(txt\_eventStartingTime, lbl\_eventStartingTimeError))  
 isCorrect[0] = false;  
 check.duration\_Check(txt\_eventDuration, lbl\_eventDurationError);  
 if (!check.duration\_Check(txt\_eventDuration, lbl\_eventDurationError))  
 isCorrect[0] = false;  
  
 //if there are no errors with the user input, the create button will update the database, otherwise it will only set the  
 //isCorrect to true, in order to properly check the input during the second click of the button  
 if (isCorrect[0]) {  
 createEvent(txt\_eventName.getText().toString(), txt\_eventLocation.getText().toString(), txt\_specificInformation.getText().toString(), txt\_eventStartingTime.getText().toString(), txt\_eventDuration.getText().toString() + " hours", user);  
 startActivity(new Intent(CreateEventActivity.this, ProfileActivity.class));  
 } else {  
 isCorrect[0] = true;  
 }  
 }  
});

//method to check whether a users is already signed up for an event  
boolean checkAttending(String users1, String username){  
 //splitting the signed up users using "," as a delimiter into an array  
 String[] users\_array = users1.split(",");  
 //looping through the array, if the user  
 //is found in the signed up users  
 //return false, otherwise return true  
 for(int i=0;i<users\_array.length;i++){  
 if(Objects.*equals*(users\_array[i], username)) {  
 return false; }  
 }  
 return true;  
}

//function that is hashing the passed password as parameter  
public static final String md5(final String toEncrypt) {  
 try {  
 final MessageDigest digest = MessageDigest.*getInstance*("md5");  
 digest.update(toEncrypt.getBytes());  
 final byte[] bytes = digest.digest();  
 final StringBuilder sb = new StringBuilder();  
 for (int i = 0; i < bytes.length; i++) {  
 sb.append(String.*format*("%02X", bytes[i]));  
 }  
 return sb.toString().toLowerCase();  
 } catch (Exception exc) {  
 return "";  
 }  
}

//the login function, responsible for calling the database connection  
//class and working on the data  
public void login(final String username, final String password) {  
 //initiating the SharedPreferences, in order to store the name of the user  
 final SharedPreferences sp\_username = this.getSharedPreferences("username", 0);  
 final SharedPreferences.Editor editor = sp\_username.edit();  
 //creating an instance of DB\_Login class with constructor data  
 new DB\_Login("http://carmeets.eu/login.php?", username, password).execute();  
 //creating a progress dialog (the circling animation while the connection with the database is established  
 //and all the work with the data is being completed  
 final ProgressDialog pd = new ProgressDialog(LoginActivity.this);  
 pd.setMessage("Please Wait.");  
 pd.show();  
 //Setting the timer for the loading screen while the connection is established and the credentials checked  
 final Timer tm = new Timer();  
 tm.scheduleAtFixedRate(new TimerTask() {  
 public void run() {  
 runOnUiThread(new Runnable() {  
 public void run() {  
 //checking if there is a response from the server  
 //no response means the connection hsa failed  
 if (!*log*.equals("")) {  
 //if there is a response from the server, cancel the progress dialog  
 pd.cancel();  
 log1 = *log*;  
 *log1l* = log1.trim().length();  
 *log* = "";  
 //cancelling the timer  
 tm.cancel();  
 //if the length of the response is different from 1  
 //notify the user that the login has been successful  
 //and update the username field in sharedpreferences with  
 //the user's username and transfer the user tot he ProfileActivity  
 if (*log1l* != 1) {  
 Toast.*makeText*(getApplicationContext(), "You have successfully logged in, " + log1 + "!", Toast.*LENGTH\_SHORT*).show();  
 //saving the user's name in shared preferences  
 editor.putString("username", log1);  
 editor.apply();  
 startActivity(new Intent(LoginActivity.this, ProfileActivity.class));  
 }  
 //if the response length is equal to 1  
 //notify the user that the login failed  
 if (*log1l* == 1) {  
 Toast.*makeText*(getApplicationContext(), "Login failed!", Toast.*LENGTH\_LONG*).show();  
 }  
 log1 = "";  
 }}  
 });}  
 }, 1, 1000);}

//function to convert the data into an acceptable string format  
String convertDate(String rawDate){  
 String[] dateArray;  
 String[] dateArray1;  
 StringBuilder newString = new StringBuilder();  
 String date1;  
 dateArray=rawDate.split(" ");  
 date1=dateArray[0];  
 dateArray1=date1.split("-");  
 newString.append(dateArray1[2]);  
 newString.append(".");  
 newString.append(dateArray1[1]);  
 newString.append(".");  
 newString.append(dateArray1[0]);  
 return newString.toString();  
}

//input check for availability field  
//if the field is empty or the data entered is in the wrong format  
//an error will be shown  
boolean avail\_Check(EditText avail, TextView avail\_error) {  
 if (avail.getText().toString().length() == 0) {  
 avail\_error.setText("Availability field is empty!");  
 avail\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 if(!avail.getText().toString().matches("^([0-9]|0[0-9]|1?[0-9]|2[0-3]):[0-5][0-9]-([0-9]|0[0-9]|1?[0-9]|2[0-3]):[0-5][0-9]$")){  
 avail\_error.setText("Invalid time format (21:00-23:00)");  
 avail\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 avail\_error.setVisibility(View.*INVISIBLE*);  
 return true;  
}

//function checking whether the event duration is valid  
boolean duration\_Check(EditText duration, TextView duration\_error){  
 if(duration.getText().toString().length()==0){  
 duration\_error.setText("Duration field is empty!");  
 duration\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 String check = duration.getText().toString();  
 //checking whether the user input is an integer number  
 try {  
 int num = Integer.*parseInt*(check);  
 } catch (NumberFormatException e) {  
 duration\_error.setText("Enter integer number!");  
 duration\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 duration\_error.setVisibility(View.*INVISIBLE*);  
 return true;  
}

//checking whether the event starting time  
//entered by the user is correct  
boolean starting\_Time(EditText avail, TextView avail\_error) {  
 if (avail.getText().toString().length() == 0) {  
 avail\_error.setText("Starting time field is empty!");  
 avail\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 if(!avail.getText().toString().matches("^([0-9]|0[0-9]|1?[0-9]|2[0-3]):[0-5][0-9]$")){  
 avail\_error.setText("Invalid time format (21:00)");  
 avail\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 avail\_error.setVisibility(View.*INVISIBLE*);  
 return true;  
}

//checking whether the email entered is in the correct format  
 boolean checkEmail(EditText email, TextView email\_error) {  
 if (email.getText().toString().length() == 0) {  
 email\_error.setText("Email field is empty!");  
 email\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 if(!*EMAIL\_ADDRESS\_PATTERN*.matcher(email.getText().toString()).matches()){  
 email\_error.setText("Wrong email format!");  
 email\_error.setVisibility(View.*VISIBLE*);  
 return false;  
 }  
 email\_error.setVisibility(View.*INVISIBLE*);  
 return true;  
}

//function checking whether the availabilities are fit  
boolean availability\_check(String e\_time, String u\_time){  
 String[] u\_timeSplitFirst = u\_time.split("-");  
 String[] e\_timeSplit = e\_time.split(":");  
 int e\_time1 = Integer.*parseInt*(e\_timeSplit[0]);  
 int e\_time2 = Integer.*parseInt*(e\_timeSplit[1]);  
 String[] u\_timeSplit1 = u\_timeSplitFirst[0].split(":");  
 int u\_time1 = Integer.*parseInt*(u\_timeSplit1[0]);  
 int u\_time2 = Integer.*parseInt*(u\_timeSplit1[1]);  
 if((e\_time1 > u\_time1) || ((e\_time1 == u\_time1) && (e\_time2 > u\_time2)) || ((e\_time1 == u\_time1) && (e\_time2 == u\_time2))){  
 return true;  
 }  
 return false;  
}

//function to create buttons dynamically  
void createButton(String str){  
 String[] str\_array = str.split("&");  
 final String e\_name = str\_array[0];  
 String e\_location = str\_array[1];  
 String e\_restriction = str\_array[2];  
 String e\_time = str\_array[3];  
 String e\_duration = str\_array[4];  
 String e\_createBy = str\_array[5];  
 String e\_creationTime = str\_array[6];  
 Button myButton = new Button(this);  
 myButton.setText("Event Name: " + e\_name + "\nEvent Location: " + e\_location + "\nEvent Restrictions: " + e\_restriction  
 + "\nStarting Time: " + e\_time + "\nEvent Duration: " + e\_duration + "\nCreated By: " + e\_createBy + "\nCreation Date: " + e\_creationTime);  
 myButton.setId(num);  
 LinearLayout layout = (LinearLayout) findViewById(R.id.*linearLayoutID*);  
 layout.addView(myButton);  
  
 myButton.setOnClickListener(new View.OnClickListener() {  
 public void onClick(View view) {  
 //initiating the SharedPreferences, in order to store the name of the user  
 final SharedPreferences sp\_eventName = ViewEventsActivity.this.getSharedPreferences("e\_name", 0);  
 final SharedPreferences.Editor editor = sp\_eventName.edit();  
 editor.putString("e\_name", e\_name);  
 editor.apply();  
 startActivity(new Intent(ViewEventsActivity.this, EventInformationActivity.class));  
  
 }  
 });  
 num++;  
}

//function checking whether preferences match  
boolean preferences\_check(String event\_restrictions, String user\_pref){  
 String[] event = event\_restrictions.trim().split(",");  
 String[] user = user\_pref.trim().split(",");  
 for(int i=0; i<event.length; i++){  
 for(int j=0; j<user.length; j++){  
 if(event[i].trim().equalsIgnoreCase(user[j].trim())){  
 return true;  
 }  
 }  
 }  
 return false;  
}

//function checking whether the locations are identical  
boolean location\_check(String event\_location, String user\_location){  
 if(event\_location.toLowerCase().equals(user\_location.toLowerCase())){  
 return true;  
 }  
 return false;  
}

//Defining the doInBackground function of ASyncTask  
@Override  
protected String doInBackground(Object[] params) {  
 try {  
  
 //Initializing the request to the database  
 String data = URLEncoder.*encode*("username", "UTF8") + "=" + URLEncoder.*encode*(Username, "UTF8");  
 data += "&" + URLEncoder.*encode*("password", "UTF8") + "=" + URLEncoder.*encode*(Password, "UTF8");  
  
 //appending the data to the link  
 URL mylink = new URL(Link + data);  
 //opening connection  
 URLConnection connect = mylink.openConnection();  
  
 connect.setDoOutput(true);  
 OutputStreamWriter wr = new OutputStreamWriter(connect.getOutputStream());  
 wr.write(data);  
 wr.flush();  
  
 BufferedReader reader = new BufferedReader(new InputStreamReader(connect.getInputStream()));  
 //creating a stringbuilder to construct the output from the response  
 StringBuilder sb = new StringBuilder();  
  
 String line = null;  
 while ((line = reader.readLine()) != null) {  
 sb.append(line);  
 }  
 //assigning the output from the php to a  
 //variable in LoginActivity activity  
 LoginActivity.*log* = sb.toString();  
 } catch (Exception e) {}  
 return "";  
}

**Results and Conclusion**

The result from developing the CarMeets applications is a fully functional application for organizing and attending car meetings. The system is able to record new users and new events into the database, as well as allow the users to edit their profiles. The application also properly differentiates when it needs to show all events or to show only the suggested events.

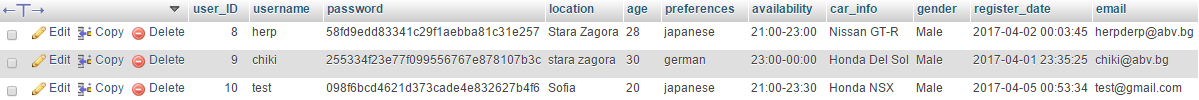
The applications turned out well. It has all the functionalities that were specified in the beginning, before the development started – all the features were implemented.

There were not really any serious difficulties during the development process. The reason for this is the fact that I have already created one Android application before, along with the database, database connection and so on. The only slight difficulty was coming up with a proper event suggestion algorithm and implementing properly afterwards.

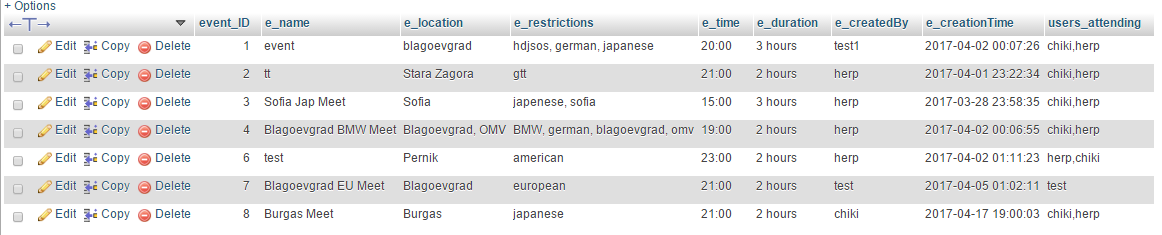
What I think I could have done better is the user interface, but it is an aspect that normally requires a lot of time to be created properly and I did not have that time. Therefore, I decided to implement a simpler design.

By finishing the project, I feel like I have improved my Android programming skills. I also wanted to implement various other capabilities of the application, such as implementing various instances of Google Maps, but these features will be left for future development. Also a web version of the application will be left for future development.

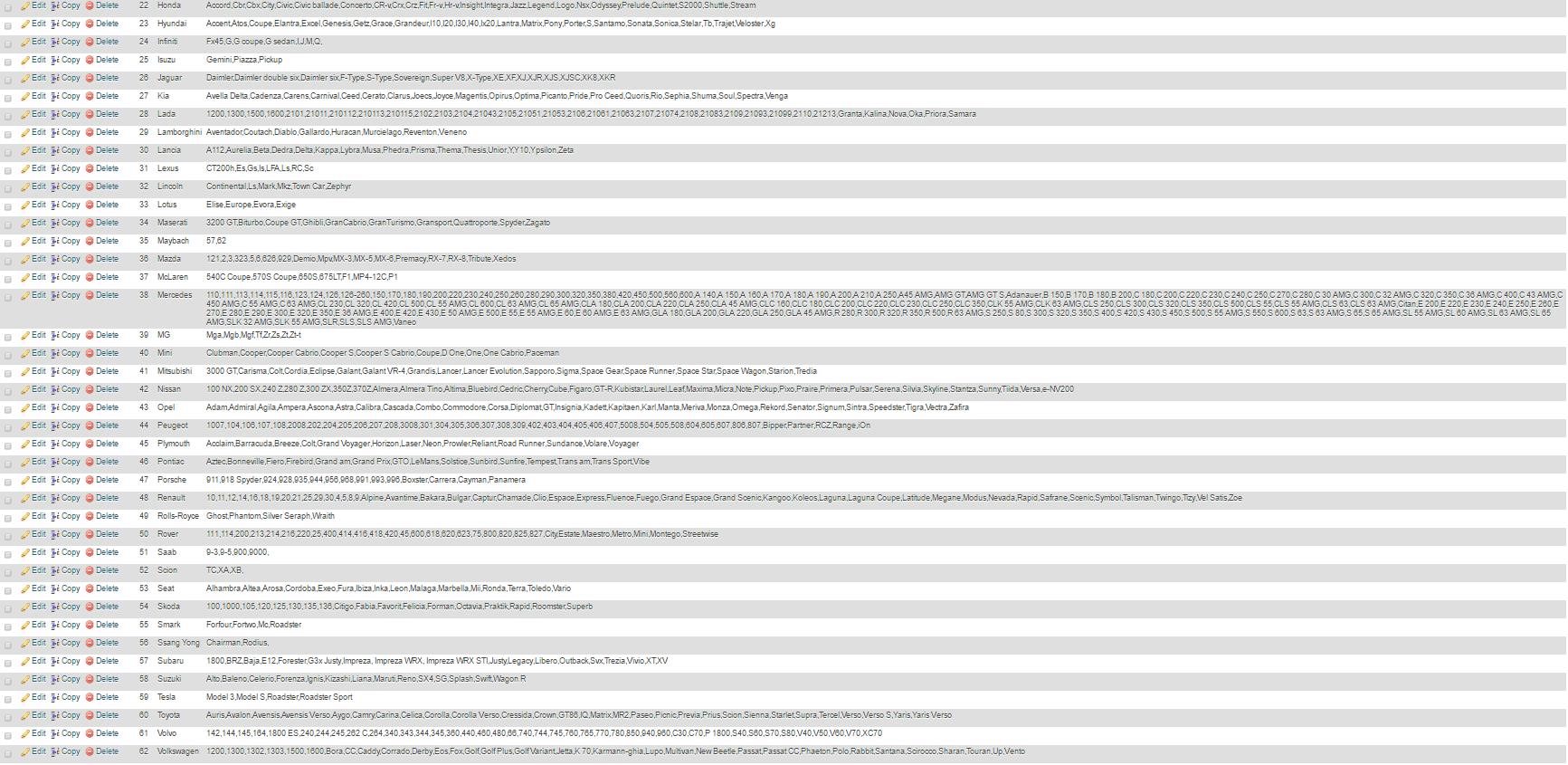
Database screenshots:



“Users” Table



“Event” Table



Part of the “Cars” database. While it is not very visible, it holds data about 62 car makes along with all their models.

**References**

<https://msdn.microsoft.com/en-us/library/ms123401>

<https://developer.android.com/reference/packages.html>

<https://stackoverflow.com/>