CMPE 332 – Database Management Systems

**K-Town Car Share**

Phase 3 – Final Report

April 6th, 2017

Andrew McClelland – 10150229

Rony Besprozvanny – 10137022

Ryan Fredrickson - 10130487

Table of Contents

[Assumptions 3](#_Toc479200833)

[ER Schema 4](#_Toc479200834)

[ER Explanation 5](#_Toc479200835)

[Relational Schema 6](#_Toc479200836)

[Relational Schema Explanation 7](#_Toc479200837)

[State Machine – User Diagram 8](#_Toc479200838)

[State Machine – Admin Diagram 9](#_Toc479200839)

[State Machine Diagram SQL Interactions & Output 10](#_Toc479200840)

[Login 10](#_Toc479200841)

[User Related Forms 11](#_Toc479200842)

[Homepage Form 11](#_Toc479200843)

[Homepage Handle Form 12](#_Toc479200844)

[Reservation Handle Form 14](#_Toc479200845)

[Homepage Query Locations 15](#_Toc479200846)

[Member Rental History 16](#_Toc479200847)

[Pickup/Dropoff Form – pdform php file 16](#_Toc479200848)

[Pickup Dropoff Form – pdhandle php file 18](#_Toc479200849)

[Admin 21](#_Toc479200850)

[Admin Homepage Form 21](#_Toc479200851)

[Monthly Invoice Handle Form 24](#_Toc479200852)

[Add Car Form 25](#_Toc479200853)

[Add Car Handle Form 25](#_Toc479200854)

[Car Rental History Handle Form 27](#_Toc479200855)

[Available Cars at Location Form 27](#_Toc479200856)

[Damaged or Need to be Repaired Cars Handle 28](#_Toc479200857)

[Rental Comment Handle Form 28](#_Toc479200858)

[Rental Comment Obj Form 29](#_Toc479200859)

[Insert Reply Comment Form 29](#_Toc479200860)

[Cars Distance Since Maintenance Handle 30](#_Toc479200861)

[Rentals on Day Handle Form 31](#_Toc479200862)

[Register Form 32](#_Toc479200863)

[Logout 32](#_Toc479200864)

[Discussion 33](#_Toc479200865)

[Encountered Problems & Solutions 33](#_Toc479200866)

[Design & Implementation Decisions 34](#_Toc479200867)

[Technologies & Tools Used in Development 34](#_Toc479200868)

[Project Reflection 34](#_Toc479200869)

[User Manual 36](#_Toc479200870)

[Regular User 36](#_Toc479200871)

[Register as New User 36](#_Toc479200872)

[Login 37](#_Toc479200873)

[Find all cars to rent on a day 37](#_Toc479200874)

[Reserve a Car 38](#_Toc479200875)

[Find all Locations 40](#_Toc479200876)

[Pick up a Car 40](#_Toc479200877)

[Drop off Car and Provide Feedback 41](#_Toc479200878)

[Rental History 41](#_Toc479200879)

[Admin 42](#_Toc479200880)

[Generate Monthly Invoice for User 42](#_Toc479200881)

[Add a Car to Fleet 42](#_Toc479200882)

[Show Rental History for Car 43](#_Toc479200883)

[Show Cars at Location 43](#_Toc479200884)

[Show Cars with Distance Since Maintenance 44](#_Toc479200885)

[View Cars with the most and least number of rentals 44](#_Toc479200886)

[Generate Monthly Invoice for User 45](#_Toc479200887)

[Show all Reservations on a Day 46](#_Toc479200888)

[Respond to a Members Comment 46](#_Toc479200889)

[Appendix 47](#_Toc479200890)

[SQL Dumps of All Tables Preloaded with the KTCS\_DB script 47](#_Toc479200891)

[Admin Table 47](#_Toc479200892)

[Car Table 47](#_Toc479200893)

[Car\_Maintenance\_History Table 48](#_Toc479200894)

[Car\_Rental\_History Table 48](#_Toc479200895)

[KTCS\_Member Table 48](#_Toc479200896)

[Parking\_Location Table 49](#_Toc479200897)

[Payment\_History Table 49](#_Toc479200898)

[Rental\_Comment Table 49](#_Toc479200899)

[Reservations Table 50](#_Toc479200900)

# Assumptions

* Each member will pay for the entire month during which they sign up, regardless of signup day (so Jan 1 and Jan 29 signups are charged for all of January).
* Each member will pay for car / reservation as soon as they book the car.
* There are no cancellations / refunds.
* A car is available for the day unless it has been reserved.
* A car reserved for a day begins at 12:00 am on that day and ends 11:59 pm on the end reserved day.
* Every car is in its specified location at the start of each rental.
* Parking location is never over capacity → a location starts under or at max capacity, and is never exceeded because all cars returned to the same location.
* Replies to comments are limited to one reply and only an admin can reply.
* Car rental history is only created when a car gets rented.
* Car maintenance history is only created when a car gets sent in for maintenance.
* Assume only one picture per car, one phone number per member, one email per member, one Credit Card per member.
* Each time maintenance is performed, only one type of maintenance can be displayed.
* No Username is required – email will be used instead.
* Reservations tuples are removed from the Reservation relation when the reservation end date is less than the current date (information is still stored in Car\_Rental\_History relation).
* At login, normal users will provide an email and password while administrators will provide an email, password, and unique admin code.

# ER Schema

## ER Explanation

To optimize the database system, the following decisions were made. These include the use of weak-entity sets and various cardinality and participation modes. The purpose of using weak-entity sets and relationships is due to the close association seen between the objects. The weak entity set object (i.e. Car\_Rental\_History) is dependent and relies on the strong entity, as it cannot exist without the Car entity set. As can be seen in the Relational Schema section, in the weak-entity relationships, the primary key of the strong entity becomes a primary key of the weak-entity set.

The ER schema above uses a variety of cardinality modes such as one-to-many/many-to-one relationships. As an example, the Reservations table has a many to one relationship with the Car and KTCS\_Member objects. This is because you can have many Reservation objects for one car and member. The many to one relationship is represented in the Relational Schema seen below by adding the primary key of the one side (i.e. The Car and KTCS Member) as attributes to the many side (i.e. Reservations). Finally, total participation is used solely for the Payment\_History to KTCS Member relationship. This is because one of the assumptions the team made is that as soon as KTCS Member is created there will be at least one Payment\_History object created which contains the monthly fee to be charged. Payment\_History objects will also be created whenever a user reserves a car.

Within the Admin table, it was decided to make it a dedicated table that is not in a relationship with anything else. This is because the Admin will only contain 3 attributes: the user name, password and admin code that is generated for the admin. Admins will sign into the website by providing all 3 fields. Admins do not have a name or any personal information associated with them and because of this the Admin table does not need to have an ISA relationship with the KTCS\_Member.

# Relational Schema

## Relational Schema Explanation

Based on the ER diagram, the relational schema was created (as seen above). In terms of handling the one to many/many to one relationships, the primary key of the one side was added as an attribute to the many side. This created a foreign key relationship. As an example, the Parking\_Location shares a one to many relationship with Car so Parking\_Location’s primary key of LocationID was added as an attribute in Car.

Another note with the relational schema is the various optimizations the team made in order to make the schema more efficient. In the ER diagram the dates for the various objects (ie. Car\_Rental\_History) were a composite value comprised of day, month and year. In SQL there is a predefined date type that takes care of the various aspects of the date. Thus, in the relational schema there is no need to have day, month and year as it is just replaced by a date object.

In terms of constraints of various variables, this is not seen in the Relational Schema but it is present in the SQL table create statements. Constraints were added for Rating, Maintenance and Status. The ‘Rating’ attribute constraint ensured that Rating would only be an integer from 1-4. The ‘Maintenance’ attribute constraints are that it can only take on the values of either ‘scheduled’, ‘repair’ or ‘body work’. Finally, the ‘Status’ attribute constraints are that it can only take on the values of either ‘normal’, ‘damaged’, or ‘not running’.

# State Machine – User Diagram

Please note that the state machine diagram is one full diagram, however we split it into two diagrams so it would be easier to view in this report. The *Login*, *Register*, and *Logout* tables are common nodes between the User and Admin diagrams.

# State Machine – Admin Diagram

# State Machine Diagram SQL Interactions & Output

## Login

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *login* form has internal *update* and *submit* events that causes internal transitions. These *updates* include changing the *email*, *password*, and *accessCode* fields, while *submit* queries the *KTCS\_Member* and *Admin* tables to validate login credentials.

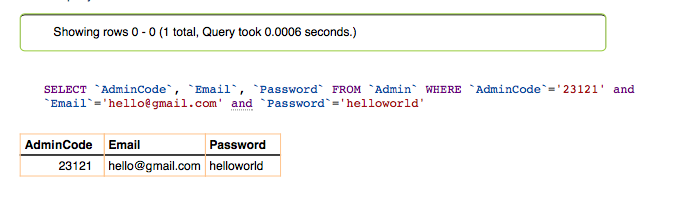
#### SQL Interactions, Logic, & Sample Output

Please note that in the php code files created the needed values for the queries are dynamic and are variables. When running all the following SQL queries the value for the given variable at that point in time was set into the SQL query.

The following queries try to validate the user email ‘[testuser@gmail.com](mailto:testuser@gmail.com)’ with the password ‘testing’.

When verifying the user, the users text password entered in the form is hashed and is compared to the hash password in the database seen above. If both email address and the hashed password match the values stored in the database, the user will successfully be logged in.

In terms of verifying an admin the following SQL query was used. The values seen in SQL statement below were the values the user entered into the Login form.



## User Related Forms

### Homepage Form

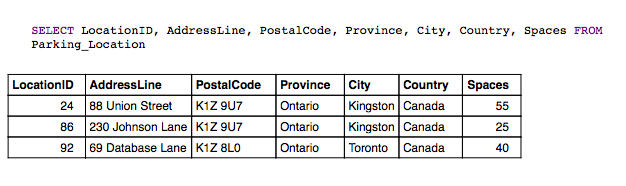
#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *Homepageform* form has internal *update*, *locationDropdown*, and *resCheck* events that causes internal transitions. These *updates* include changing the *startDate*, *endDate*, and *parkingLocation* fields while trying to reserve a car. *locationDropdown* queries the *Parking\_Locations* table to return all locations, while *resCheck* queries the *Reservations* relation to delete any tuples that have an end date smaller than the current timestamp.

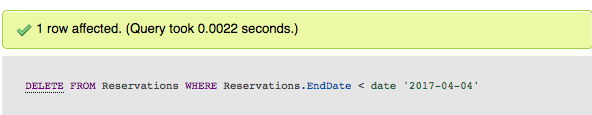
#### SQL Interactions, Logic, & Sample Output

The user homepage form has the following SQL queries executed within it.

Firstly, within the homepage users will be able to reserve a car. In order to begin the reservation process users, specify a start and end date for the reservation and choose from one of the Parking location spots from a dropdown list. This parking location dropdown list is dynamic and retrieves the parking locations from the database with the following query:



In addition, the user homepage executes another query every time the user accesses the homepage/reloads it. This query is responsible for consistently checking and deleting reservations that have already happened ie. Reservation object end date is less than today’s date. Reservation objects should be deleted as they complete as the reservation object is converted to a car rental history object upon its completion. The following SQL query will delete any and all old reservation objects:



The DELETE query does not return any results, but as seen above it did indeed delete one row whose reservation end date was less than today’s date (April 4th 2017). Doing a current dump of the Reservations table (seen below) it is evident that there is no reservation whose end date is less than today’s date.



### Homepage Handle Form

#### State Machine Diagram Explanation

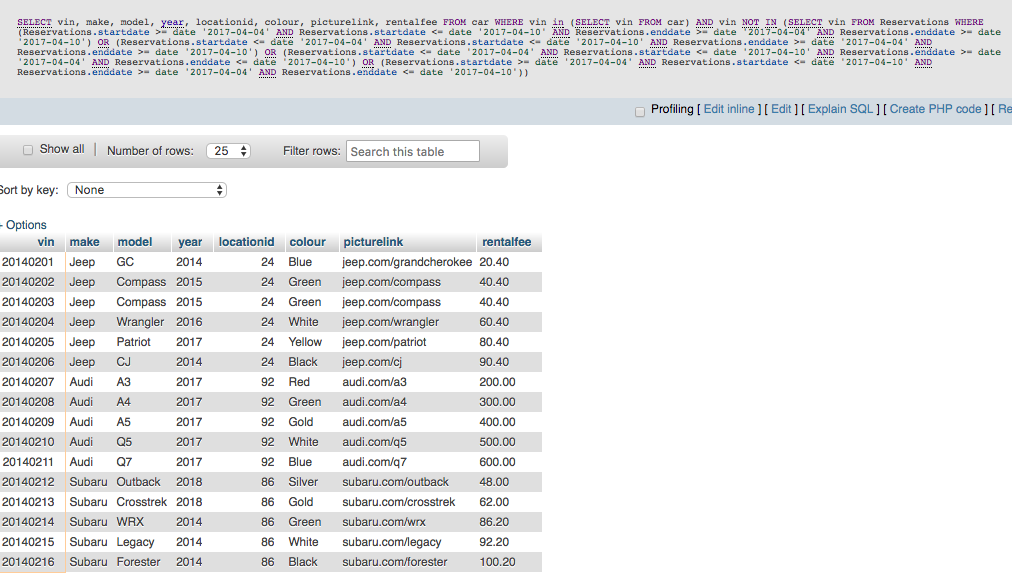
In addition to the entry and exit conditions, the *homepageHandle* form has an internal *availableCars* event that causes an internal transition. *availableCars* queries the *Car* and *Reservations* relations to return any available cars and their information to the user.

#### SQL Interactions, Logic, & Sample Output

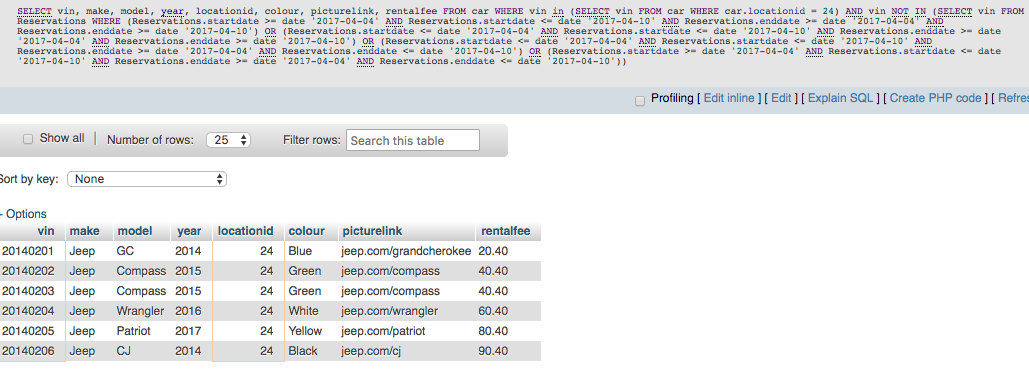
The homepagehandle php file is called when the user clicks submit in the homepageform.

The homepagehandle form displays all the cars that are available to reserve for the specified dates. The user specified dates are passed from the homepageform (where they are entered by the user) to this form to be used for the following queries:

**Please note the following queries are for the user entered start date of 2017-04-04 and end date of 2017-04-10.**

If the user selected the ‘All’ locations in the homepage the following query will be performed and the following cars will be displayed:

If the user selected a specific location such as ’88 union street’ which has a location ID of 24, the following query will be performed:



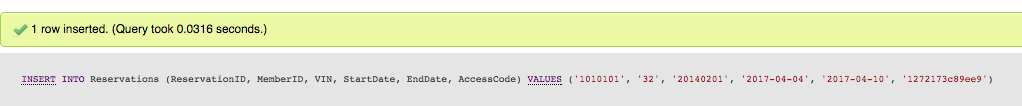
Overall, the difference between choosing the ‘All’ locations or a specific location is whether the query looks at the location ID or if it instead selects all location ID’s (in the case of ‘All’).

### Reservation Handle Form

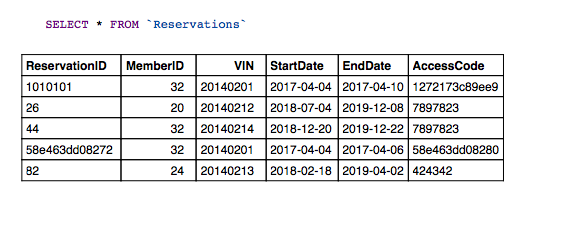
#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *reservationHandle* form has an internal *createRes* event that causes an internal transition. *createRes* queries the *Reservations* and *Payment\_History* relations to insert the reservation and the new *Payment\_History* object.

#### SQL Interactions, Logic, & Sample Output

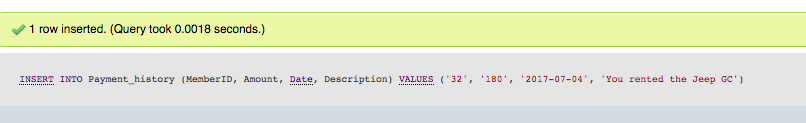
The reservationhandle php file will get the given car info (such as VIN, make, model, year, etc..) from the homepagehandle form as every car shown in the homepagehandle form has an associated ‘Reserve a Car’ button. Upon retrieving this information, the following queries are performed to reserve the car:

The above query will insert the new Reservation into the Reservation table. To confirm that the query is correct the dump of the Reservation table can be seen below.

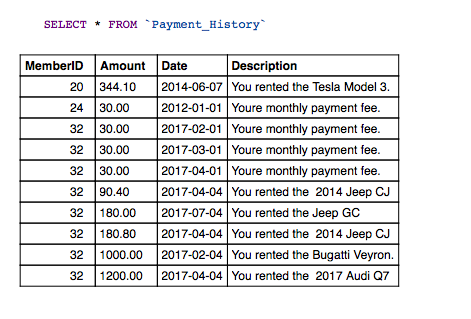


From the above dump, it is evident that the ‘20140201’ VIN reservation object was inserted.

In addition, the reservationhandle form will also create a payment object with the following query:



Performing a dump on the Payment\_History table gives the following results:



It is evident that the new Payment object was inserted successfully.

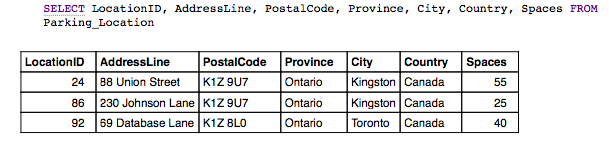
### Homepage Query Locations

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *homepageQueryLocations* form has an internal *locCheck* event that causes internal transition. *locCheck* queries and shows KTCS\_Locations.

#### SQL Interactions, Logic, & Sample Output

The homepageQueryLocations php file displays all the Parking\_Location objects and their information to the user. The SQL query performed is seen below.



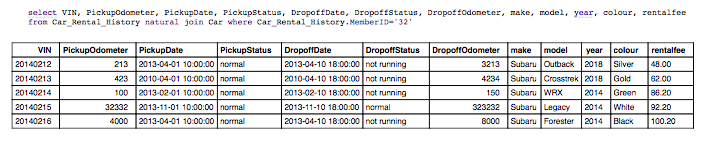
### Member Rental History

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *memberRentalHistory* form has an internal *rentalCheck* event that causes internal transition. *rentalCheck* queries and shows Car\_Rental\_History for the logged in *MemberID*.

#### SQL Interactions, Logic, & Sample Output

The following form presents all of the users previously completed rentals and the information about each of them. The form performs the following SQL query:



### Pickup/Dropoff Form – pdform php file

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *pdform* form has internal *pickup, dropoff, resCheck, noReservation,* and *completeReservation* events that cause internal transitions. *pickup* allows the user to update the odometer and pickup status upon filling out the pickup form, while *dropoff* allows the user to update the odometer, dropoff status, rating, and comment upon filling out the dropoff form. *resCheck* queries the *Reservations* relation to check if there are any reservations for the logged in *MemberID*, *noReservation* indicates to the user that they have no current reservation, while *completeReservation* indicates to the user that their reservation forms have already been completed.

#### SQL Interactions, Logic, & Sample Output

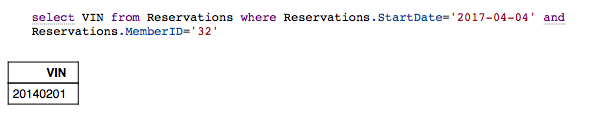
The following form is an extremely dynamic form. It can take on 4 different internal states. The first internal state it can take on is if there are no reservations whose start date is today’s date, it will simply display text informing the user there is no pickup form to complete. To check if any reservations become active today, the SQL statement used is:

select VIN

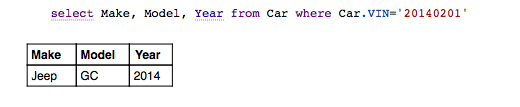
from Reservations

where Reservations.StartDate='$currentDate' and Reservations.MemberID='$memberID'

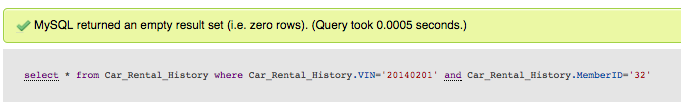
Please note that $currentDate and $memberID are defined php variables in pdform.php. Running this query with current date of 2017-04-04 and memberID of 32 gives the following results:



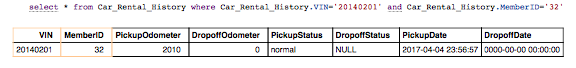
If the result returns at least 1 row or more (in this case it does) then there is a reservation object with the following VIN seen above starting today. If this is the case the following query will be ran to collect the car information for the VIN from the previous query.



Now the following query is executed to check whether a Car\_Rental\_History object already exists for the given VIN or not. The purpose of this is to determine if the form should dynamically change to its other internal states which is show the pickup form values or show the dropoff form values.



As can be seen above, if no rows are returned then this means the Car\_Rental\_History object does not exist for the current car, so the pickup state form will be shown asking the user to enter the current odometer and status of car values. Upon submitting the pickup values and these values being handled by the pdhandle php file a Car\_Rental\_History object will be created for the given car. It will only contain the pickupOdometer and pickupStatus values the user entered as well as the pickup timestamp. All the dropoff values will initially be set to NULL (if user has filled in the pickup form). Now, when user goes to the pickup/dropoff form again (pdform.php) it will once again check the first query seen above. This will still return a result greater than or equal to 1, so the pdform will now re-try the above query.



As can be seen from above, the SQL query returned a valid result. This means that the pickup values were set, but the dropoff values were not set. The pdform will now analyze the contents of the dropoffOdometer column. If it finds that the dropoffOdometer has not been set (ie. Its 0 or null) it will know to show the dropoff form and not the pickup form. Upon showing the drop off form the user will be able to enter the dropoffOdometer and dropoffStatus. The dropoffDate timestamp will be generated by the scripts. In addition, in the dropoff form the user will be able to provide feedback on the rental by entering the rating and optionally a comment for the rental. These values are passed onto the pdhandle php form.

The final internal state in the pdform is the already completed rental text shown to the user. This text will be show to the user if all the above query’s column values are filled with valid results (ie. Not null or 0). This implies that the user has completed both the pickup and dropoff aspects of the form and so the form will not allow the user to re-complete the form.

### Pickup Dropoff Form – pdhandle php file

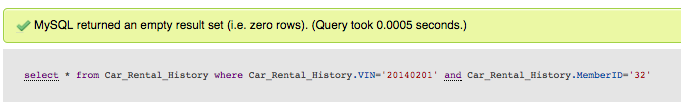
#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *pdhandle* form has internal *pickup* and *dropoff* events that cause internal transitions. *Pickup* queries *Car\_Rental\_History* for the correct car info for the user, while *dropoff* queries *Car\_Rental\_History* and *Rental\_Comments* to return the relevant car info for the user and allow for the user to add comments.

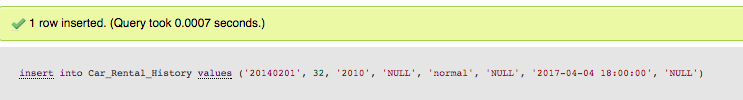
#### SQL Interactions, Logic, & Sample Output

The pdform file passes the user entered input to the given internal state (pickup or dropoff form values) to the pdhandle file upon the user pressing the submit button in pdform. The pdhandle will perform the following queries:

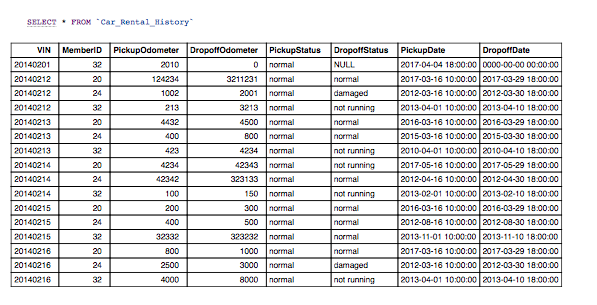
Firstly, the pdhandle file will check to see if there currently exist any objects of Car\_Rental\_History for the given VIN and memberID. This query is essentially the same as that of the pdform. Executing this query initially produces the following results:



Since the SQL query did not return any results, the pdhandle will create a Car\_Rental\_History object with the user entered values (for the pickup part of the form) from the pdform file. This can be seen below:

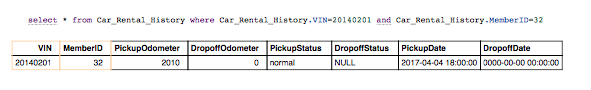


To confirm this worked the Car\_Rental\_History dump is seen below.

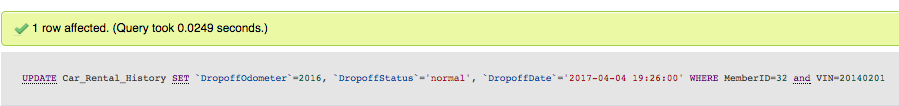


From the results, above, it can be confirmed that the new object was added to the table (the object is seen as the first value in the dump).

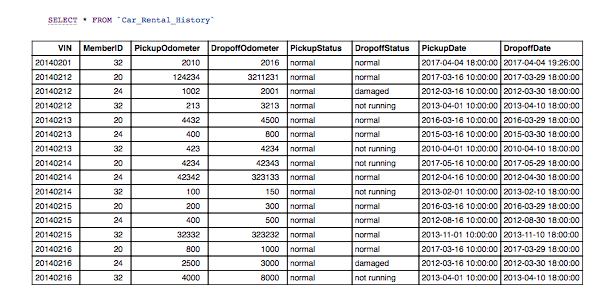
Since the object has now been created and populated with the pickup values the next time pdform calls pdhandle it will pass in the dropoff values. The same first query will be executed in pdhandle to determine if the Car\_Renal\_History objects exists for the given VIN and memberID. This time around it will indeed find the object with the pickup values set as seen below.



Since the pickup values are set, the pdhandle will check to see whether the dropoffOdometer value is set. In this case it is not and therefore the pdhandle will invoke the following query to update this object with the dropoff values supplied by the user in pdform.

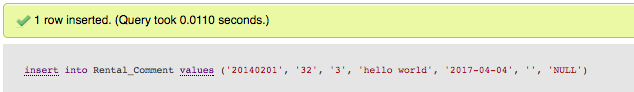


To confirm whether this query worked the table dump can be seen below.

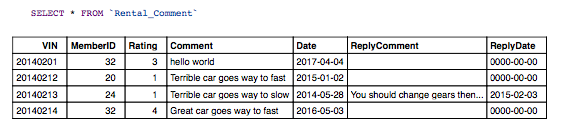


As seen from the table dump above, it is evident that the object was updated with the dropoff values.

In addition to updating the dropoff values in the correct Car\_Rental\_History object the pdhandle will also take the user entered rating and rental comment values (user supplied these values in pdform) and create a Rental\_Comment object. The following SQL demonstrates this.



Doing a table dump produces the following results.



It is evident from above that the new Rental\_Comment object has been inserted. In terms of the ReplyComment by default this is nothing as the admin has not replied to the comment. If the Admin wishes to reply to the users comment in the Admin portal this table will be updated and the change will be reflected.

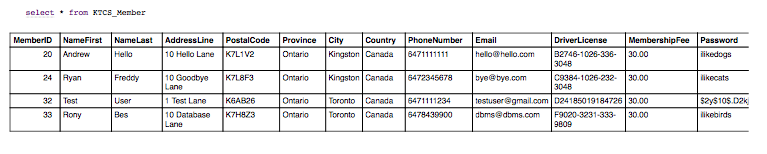
## Admin

### Admin Homepage Form

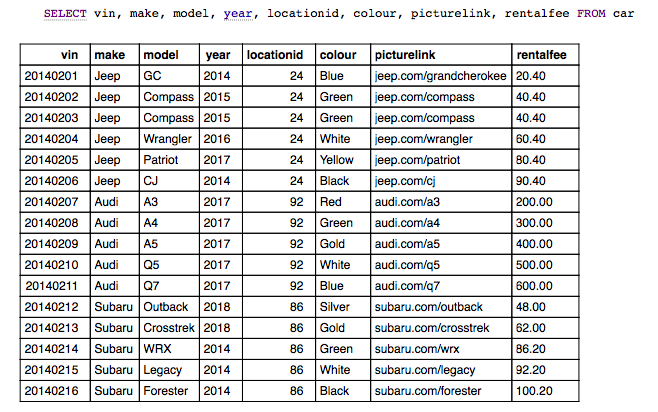
#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *adminHomepage* form has internal *update*, *monthlyInvoiceDropdown, rentalHistoryDropdown, availableCarDropdown, maxMinRentals,* and rentalCommentDropdown events that cause internal transitions. *update* allows the admin to change the *invoiceMemberID, invoiceStartDate, invoiceEndDate, rentalHistoryCar, availableCarLocations, distanceSinceMaintenance, rentalOnDay,* and *rentalCommentMemberID*. The *monthlyInvoiceDropdown* queries the *KTCS\_Member* relation to allow the admin to select a member to view their monthly invoice, *rentalHistoryDropdown* queries the *Car* relation to allow the admin to select a car to view its rental history, and *availableCarDropdown* queries the *Parking\_Locations* relation to allow the admin to view all available cars and reservations, if any, at a selected location. Additionally, *maxMinRentals* queries the *Car\_Rental\_History* and *Car* relations to view the cars that have the most and least amount of rentals, while *rentalCommentDropdown* queries the *KTCS\_Member* relation to view all comments made by a certain user.

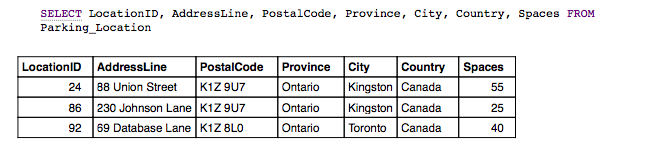
#### SQL Interactions, Logic, & Sample Output

The admin homepage form (adminHomePage.php) contains various SQL queries. Firstly, within this form there is a dropdown allowing an admin to choose a specific user for the monthly invoice section. The dropdown consisting of all the KTCS\_Members performs the following query.

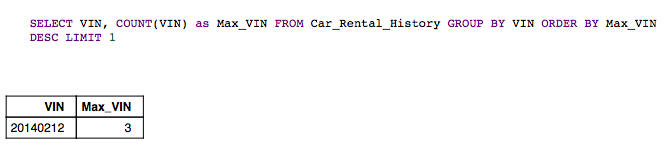
Another query the admin homepage performs is to dynamically load the dropdown list with all the cars in the database for the ‘View Rental History for Cars’ section. The query is seen below.



For the ‘Available Cars at Location’ section of the admin homepage the following query is performed to retrieve the dynamic parking locations dropdown list.

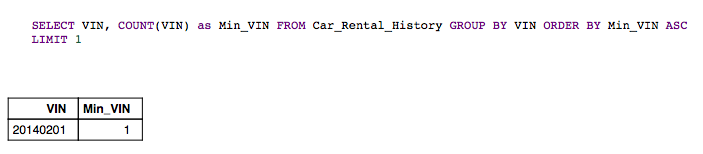


For the ‘View Cars with Most/Minimum Rentals’ section the following set of queries is ran:



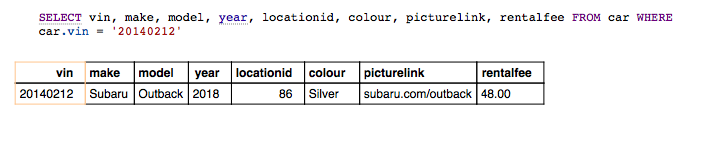
The purpose of the above query is to count the occurrences of each VIN and order in ascending. LIMIT 1 is used to only select the VIN that contains the most occurrences as the max rental is desired.

Similarly, to find the VIN that contains the minimum number of occurrences the following query was ran.

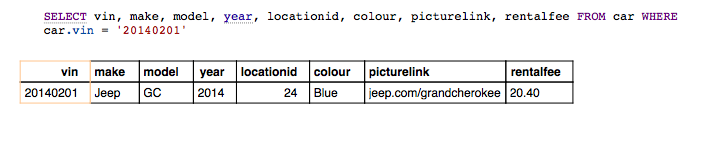


Now that the VIN’s for the max and min rental were found the information for the given VIN’s is found and shown to the user using the following queries:

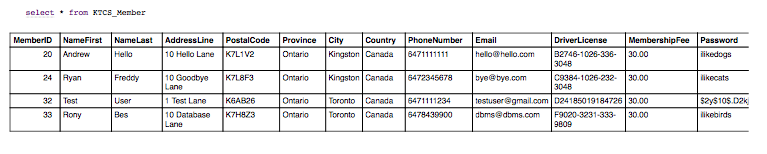
Max Rental – VIN found above to be 20140212

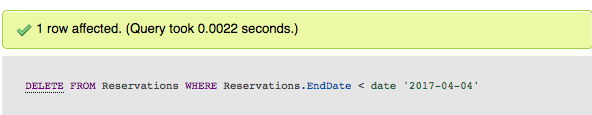


Min Rental – VIN found above to be 20140201



As for the ‘View Rental Comments’ section of the admin homepage the Members dynamic dropdown list is populated through the following query.

Finally, similarly to the user homepage the last query the admin homepage runs is to delete any Reservation objects whose end dates are less than the current date. The following query accomplishes this.



The DELETE query does not return any results, but as seen above it did indeed delete one row whose reservation end date was less than today’s date (April 4th 2017). Doing a current dump of the Reservations table (seen below) it is evident that there is no reservation whose end date is less than today’s date.



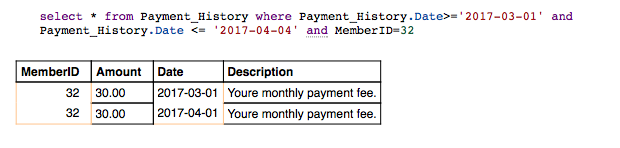
### Monthly Invoice Handle Form

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *monthlyInvoiceHandle* form has an internal *checkPayments* event that causes an internal transition. *checkPayments* queries the *Payment\_History* relation to view all payment objects for a given member.

#### SQL Interactions, Logic, & Sample Output

The monthlyInvoiceHandle.php form is called when the user clicks the Submit button in the Monthly Invoice section of the admin homepage. In the admin homepage, the user specifies the member and start/end dates that they would like to retrieve the Invoice for. These values are passed to monthlyInvoiceHandle and the following query is executed.



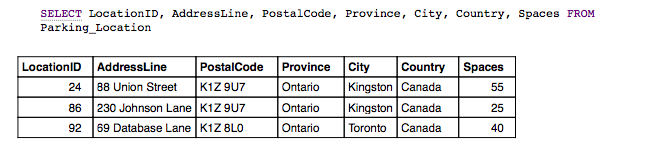
### Add Car Form

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *addCarForm* form has internal *locationDropdown* and *update* events that cause internal transitions. *locationDropdown* queries the *Parking\_Locations* relation to view all locations to add the new car to, while *update* allows the admin to change the *VIN*, *make, model, year, location, colour, picture,* and *rentalFee* fields of the added car.

#### SQL Interactions, Logic, & Sample Output

Within this form admins will add a new car to the fleet. The only SQL statement executed here is to dynamically populate the location to associate the new car with as seen below.



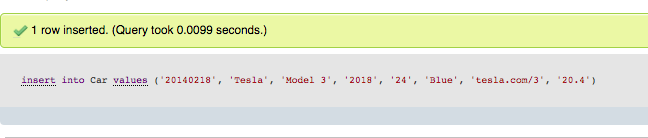
### Add Car Handle Form

#### State Machine Diagram Explanation

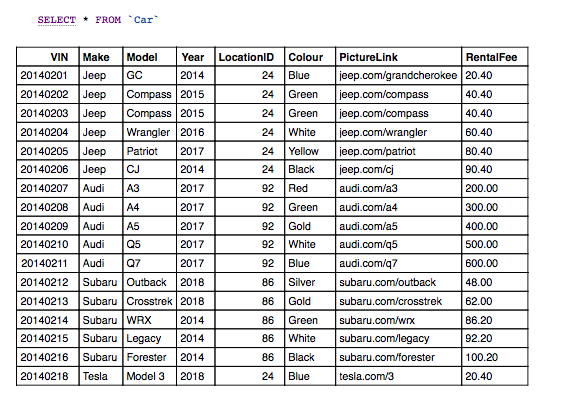
In addition to the entry and exit conditions, the *addCarHandle* form has an internal *insertCar* event that causes an internal transition. *insertCar* queries the *Car* relation to insert the provided new car information into the table.

#### SQL Interactions, Logic, & Sample Output

The addCarHandle form contains all the admin entered information from the addCarForm file. The addCarHandle performs the following query to add the car to the fleet with the admin entered car information.



Doing a dump of the table gives the following results.



As can be seen above, the new car (Tesla Model 3) was added to the fleet.

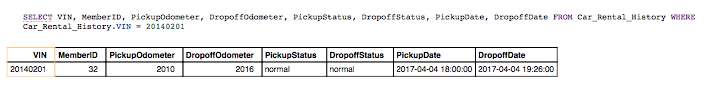
### Car Rental History Handle Form

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *carRentalHistoryHandle* form has an internal *displayRentals* event that causes an internal transition. *displayRentals* queries the *Car\_Rental\_History* relation to return all rental history on the selected car.

#### SQL Interactions, Logic, & Sample Output

The carRentalHistoryHandle file is called from the ‘View Rental History for Cars’ section of the admin homepage. Within the admin homepage admins specify the car they would like to retrieve rental information about and the chosen car’s VIN is passed to the carRentalHistoryHandle form. The following query is then performed.



### Available Cars at Location Form

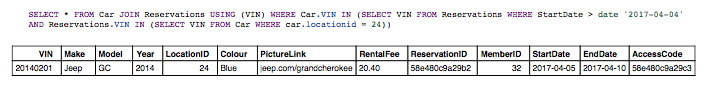
#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *availableCarsAtLocation* form has an internal *carAndReservationInfo* event that causes an internal transition. *carAndReservationInfo* queries the *Car* and *Reservations* relations to view all available cars at a location and reservations for them, if any.

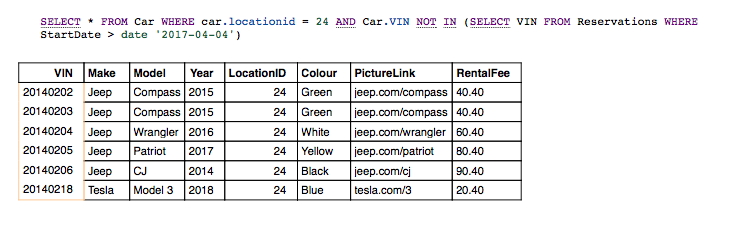
#### SQL Interactions, Logic, & Sample Output

From the admin homepage admins select the parking location they would like to view information about the current available and reserved cars. The following queries are then performed.

For cars that have current reservations:



For cars that do not have current reservations:



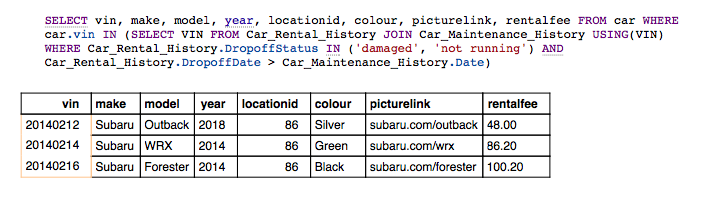
### Damaged or Need to be Repaired Cars Handle

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *DamagedRepairCarsHandle* form has an internal *viewBrokenCars* event that causes an internal transition. *viewBrokenCars* queries the *Car\_Rental\_History*, *Car\_Maintenance\_History,* and *Car* relations to return all cars that have been ‘damaged’ or ‘not running’ and have not yet been maintenance to repair them.

#### SQL Interactions, Logic, & Sample Output

Upon admins entering the following form the following SQL query will be executed.



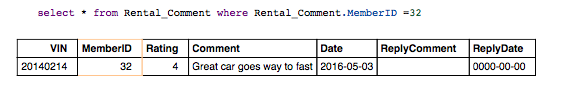
### Rental Comment Handle Form

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *rentalCommentHandle* form has an internal *viewRentalComments* event that causes an internal transition. *viewRentalComments* queries the *Rental\_Comment* relation to view all rental comments made by a user.

#### SQL Interactions, Logic, & Sample Output

Upon users selecting a member and hitting submit in the ‘View Rental Comments’ section the following query will be executed.



Since the admin has not replied to the users comment a ‘Reply to Comment’ button will be seen. If the admin hits this button it will take them to the rentalCommentObj file which will ask them to enter their reply.

### Rental Comment Obj Form

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *rentalCommentObj* form has an internal *update* event that causes an internal transition. *update* allows the admin to edit a reply comment to a user’s comment that has yet to be responded to by an admin.

#### SQL Interactions, Logic, & Sample Output

Upon entering this form, admins will see the original comment and will have a text field to reply to the comment. No SQL queries are executed here.

### Insert Reply Comment Form

#### State Machine Diagram Explanation

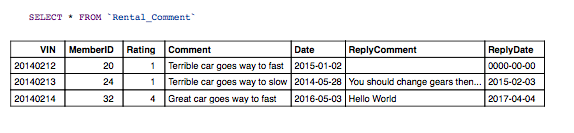
In addition to the entry and exit conditions, the *insertReplyComment* form has an internal *updateReplyComment* event that causes an internal transition. *updateReplyComment* queries the *Rental\_Comment* relation to insert the admin’s reply comment to the user into the relation.

#### SQL Interactions, Logic, & Sample Output

Upon admins entering their response to the user and pressing submit the insertReplyComment form will update the record in the database using the following query.



The table dump can be seen below.



As can be seen above the replyComment and ReplyDate fields were updated for the specific user comment.

### Cars Distance Since Maintenance Handle

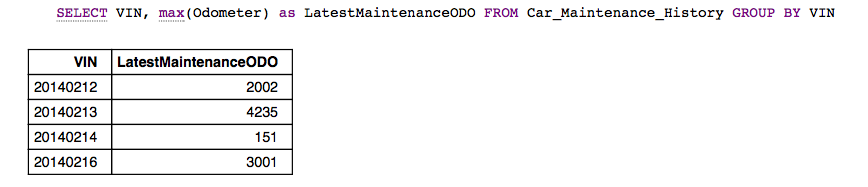
#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *CarsDistanceSinceMaintenanceHandle* form has an internal *carsTravelledSinceUserEnteredDistance* event that causes an internal transition. *carsTravelledSinceUserEnteredDistance* queries the *Car\_Maintenance\_History* , *Car\_Rental\_History*, and *Car* relations to return all cars that have travelled more than the user-entered distance since their last maintenance.

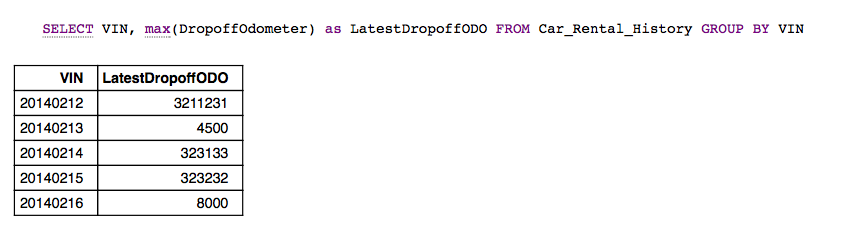
#### SQL Interactions, Logic, & Sample Output

The carsDistanceSinceMaintenanceHandle form takes in the user input for the odometer value of the cars since maintenance from the admin homepage. With this value, it then performs the followings queries.

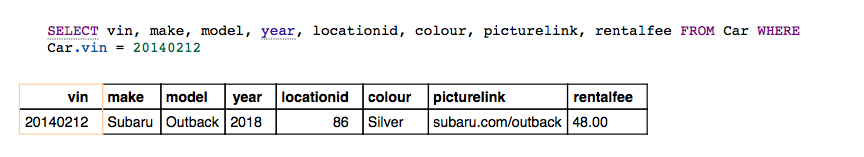
First it queries all unique VINS in Car\_Maintenance\_History and returns its highest odometer value (its latest one).



Next the following query is performed to query all unique VINS in Car\_Rental\_History and return its highest dropoff odometer value (its latest one).



Finally, the results from both queries are compared to one another through various php logic and finally the following query is executed to query all unique car information for the relevant VIN in the array. The query shown below just shows this for one VIN of 20140212.



To get the final variable seen on the form which is the km travelled since last maintenance the first two queries seen above were compared to one another and various php logic was used to find this value. This value will be displayed for the given car info seen above.

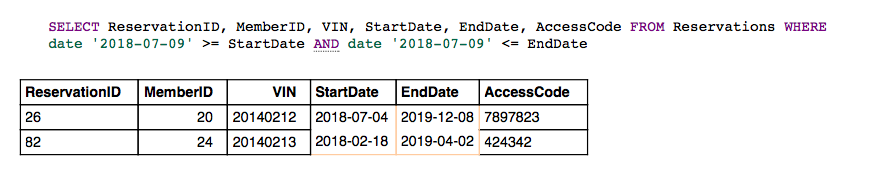
### Rentals on Day Handle Form

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *RentalsOnDayHandle* form has an internal *viewReservationsOnDate* event that causes an internal transition. *viewReservationsOnDate* queries the *Reservations* relation to view all reservations on scheduled during the selected date.

#### SQL Interactions, Logic, & Sample Output

The rentalsOnHandle form recieves the user input for the date to view rentals on that the user specified in the adminHomepage. It then performs the following query.



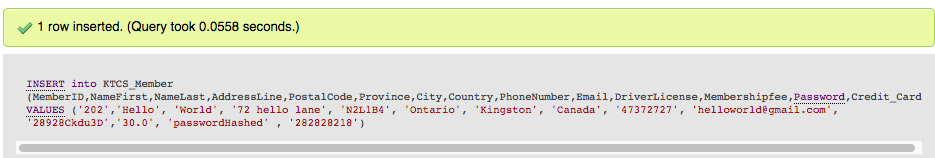
## Register Form

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *Register* form has internal *update* and createUser events that cause internal transitions. *update* allows the new user to enter their information, namely their *firstName, lastName, address, postalCode, city, province, country, phoneNumber, email, driverLiscense,* and *creditCard.* *createUser* queries the *KTCS\_Member* relation to insert the new user object and their information into the relation.

#### SQL Interactions, Logic, & Sample Output

Upon the user entering all the values in the registration form and hitting the submit button on the Register form the following query will be executed with the users entered values. The user entered values can be seen within this query.



As can be seen above the user hello world was registered into the system. Doing a SQL dump of the KTCS\_Member gives the following results.



From the above SQL export, it is evident that the Hello World user was added to the system.

## Logout

#### State Machine Diagram Explanation

In addition to the entry and exit conditions, the *logout* form has an internal *clearSession* event that causes an internal transition. *clearSession* destroys the current session that was running while the user or admin was logged in.

#### SQL Interactions, Logic, & Sample Output

Logout does not perform any SQL commands. It solely destroys the session and redirects to the Login page.

# Discussion

## Encountered Problems & Solutions

Throughout the development process of the KTCS service, the team experienced a variety of problems, but the team was successfully able to overcome them. The most prominent issue experienced was the steep learning curve of PHP and HTML at the beginning of the project. None of the members on the team had worked with PHP before and this resulted in quite a bit of time at the start to be devoted to learning the basics of PHP. This included learning how to pass values between forms, embed HTML into PHP, retrieve SQL results and display results to user. The team was able to solve this issue by starting the project early. The team started to work on the coding aspect of the project almost immediately after the Phase 1 report was due. This gave the team ample time to learn the required tools.

In regards to PHP one of the biggest issue was figuring out how to properly use the SESSION variables. The team used SESSION variables to pass around prominent information and ensure it persisted between forms. Some of the values passed through SESSION variables included the currently logged in memberID as well as their name. The purpose of using global SESSION variables was to optimize the product to remove the need for redundant queries (otherwise would need to query KTCS\_Member to get user info on every form). The team ran into issues with the SESSION variables later in the development cycle as it was noticed that at times the SESSION variables were not updating with the correct values, even though they were being set. This issue was fixed by ensuring that the cache and cookies were cleared before starting the KTCS service.

Debugging in PHP was a common issue experienced by all members of the team. To simplify debugging and to ensure consistency between members when debugging the various php forms, all the SQL statements executed were echoed along with every row’s value for every column of a given table. Debugging was one of the most time-consuming aspects of the project as it would seem that the query executed without errors, but then when looking at the database the object would not be inserted/updated. Therefore, echoing all the SQL statement results ensured the team could see what the issue was. One aspect where debugging was not possible was when populating the database using the KTCS\_DB script. When inserting values to the tables through the script at times it was noticed that the database was not populated correctly. The KTCS\_DB did not give any debug info so the team had to solve these issues by explicitly copying and pasting the SQL query into PHPMyAdmin to check the error that it was giving.

Finally, with the high volume of php files produced for this project it became hard to track them at times. At times the team had a standalone form (ie. monthlyInovice.php) that was merged as part of the homepage, but the monthlyInvoice.php form was not deleted from git. This resulted in confusion between members when it came to editing. The team overcame this issue by consistently refactoring the files in git and removing the files that are no longer in use.

## Design & Implementation Decisions

One of the most important design decisions the team made was spending a lot of time designing the ER schema. The ER schema developed ensured that it would be able to meet all the functional requirements using 9 tables with a combination of weak/strong entity sets, primary and foreign key relationships, various cardinalities (such as many to one) and integrity constraints. By stressing the importance of the ER schema, it allowed the team to easily convert the ER schema into relational algebra which was then converted into SQL to create a strong foundational architecture. This allowed the team to more seamlessly perform the various queries involved in this project as the underlying DB design was designed to ensure these queries were handled correctly.

Another design/implementation detail the team focused on is modularity. Firstly, the team solely created one MySQL connection object that was passed to every form through the use of the include PHP command. This allowed the project to only contain one database connection object that could be accessed from any form instead of having to create a separate connection object in every form. In addition, for every user interaction form there was an associated handle form that would be called when the user hits submit. The purpose of having separate handle forms and not having both user interaction and handle in one form is for modularity. It allowed the team to separate user interaction from functional SQL code. Moreover, with this approach the user interaction form code could be altered without altering the functional SQL code (in the handle form).

## Technologies & Tools Used in Development

As for technologies used in this project the team used HTML, PHP, CSS and Bootstrap for the code as well as GIT for file tracking. The reason why the HTML, PHP, CSS and Bootstrap suite were selected is due to their ease of use in comparison to other tools such as JavaScript or building a Python based server. The team members had very limited knowledge of HTML and essentially no knowledge of PHP or CSS. The inclusion of Bootstrap for the HTML made styling and constraints easier. GIT was used for file tracking and for revision monitoring. GIT allowed all members to maintain the same view of the code at any point and if an issue with a given members code did occur it was very easy to isolate it and revert to a commit where the feature was working. PHPMyAdmin as well as XAMPP were used to provide the MySQL database to the application.

## Project Reflection

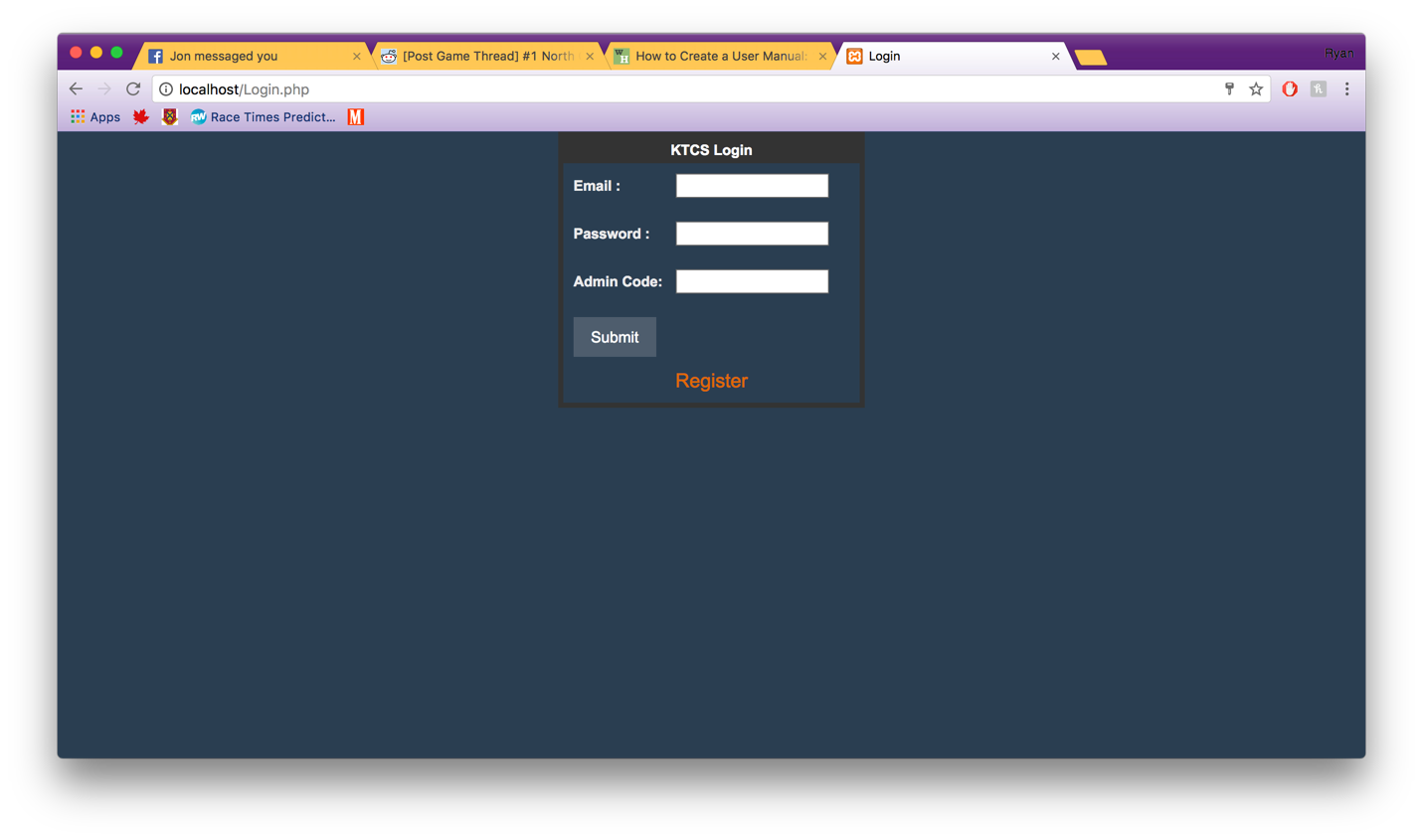
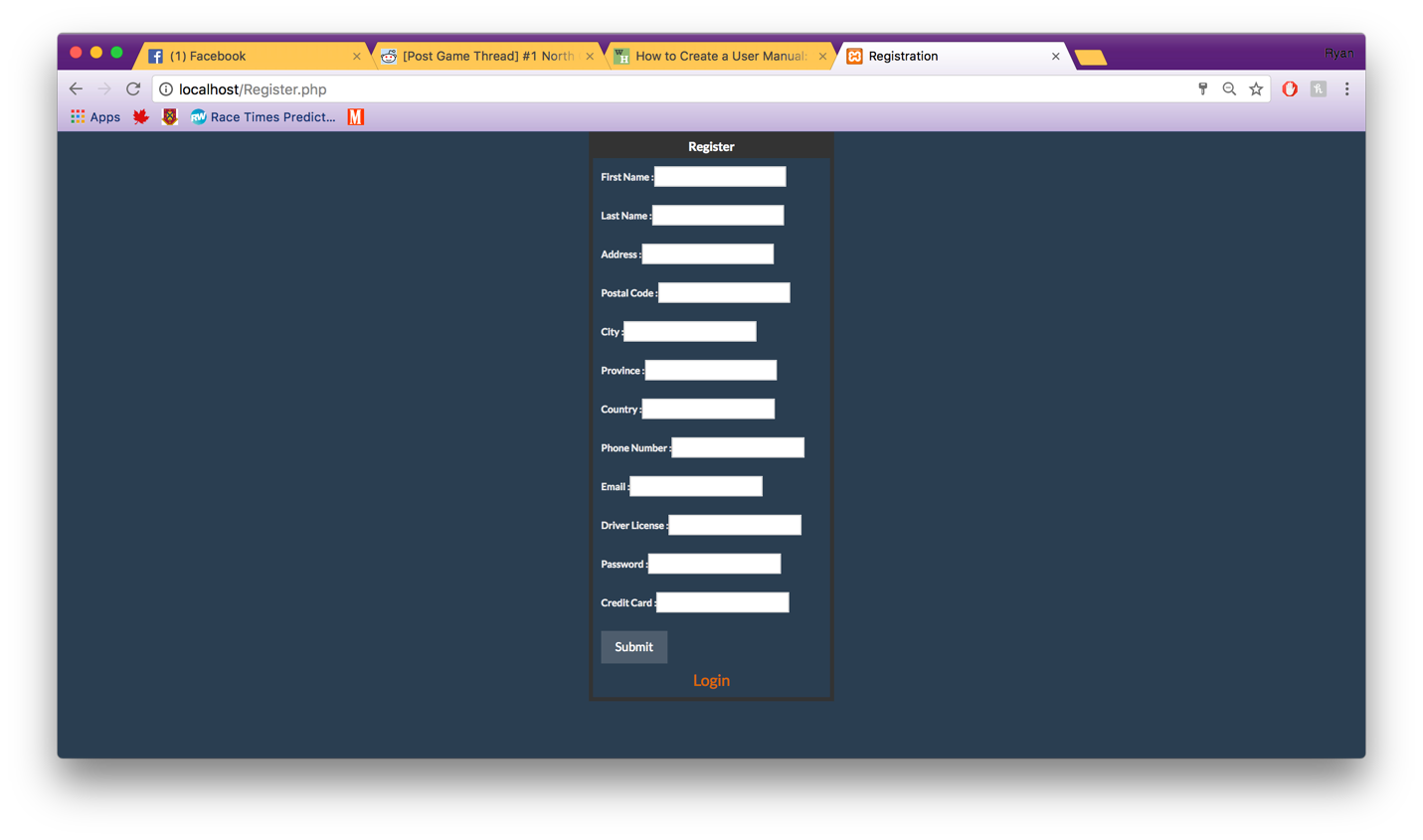
If the team had the chance to go back the team would change a few things. Firstly, the team would research more into using SESSION variables as a global way of passing variables between forms. The SESSION variables were used throughout the project, but it did lead to issues and at times a SESSION variable would not update to a new value. By researching more into this area, the team would understand it better and ensure that such a problem did not happen. In addition, the team would protect against SQL injection by using prepared statements. Currently, the project developed does not use prepared statements and thus it is susceptible to SQL injection. Finally, the team would try to host the website on a server rather than running it locally to create a more coherent development environment. If the website was hosted on a server, the team would be able to develop on this server and thus all team members would always be viewing the same DB and code (instead of local copies).

Overall, upon completing the project, the team learned a lot about HTML, PHP, CSS, Bootstrap, PHPMyAdmin, MySQl and GIT. The team came to understand the complexity and potential of these tools and gained some experience using these tools. The team made various important design decisions that guided the team throughout the development process such as stressing the importance of the ER schema and a modular form design. The team also ran into several issues such as tracking old files that are no longer in use and the learning curve for PHP, but the team was able to overcome these challenges. In conclusion, the team is proud of the work it accomplished with the KTCS service and hopes that one day this service will become a viable product that all students and residents of Kingston can use to gain more flexibility and convenience.

# User Manual

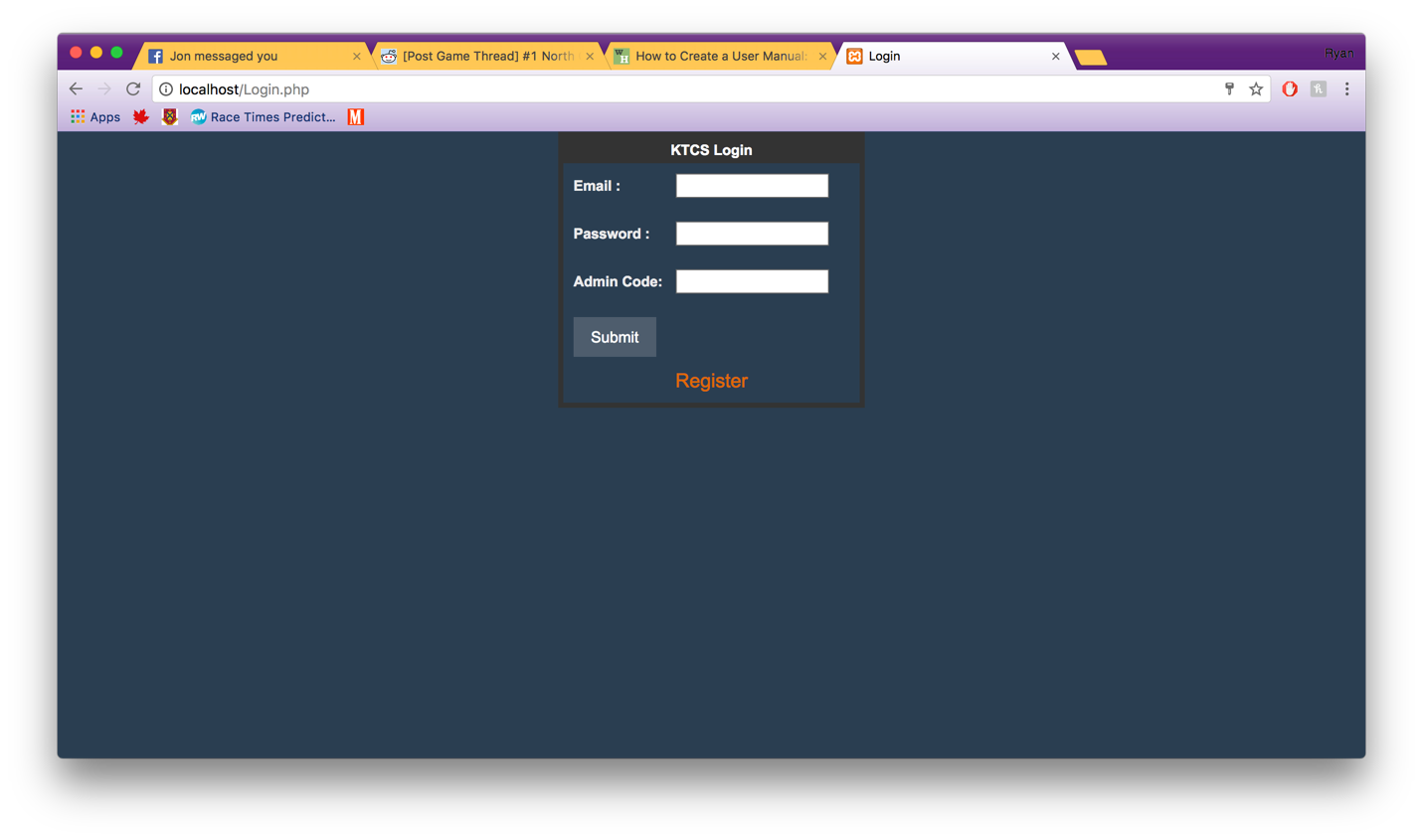
## Regular User

### Register as New User

1. From the login screen click the register button.
2. This will launch you to the Register page where you should fill out all of the fields then click Submit to complete.
3. If all needed information is entered this will launch you back to the Login screen which you will be allowed to log in with your new account.

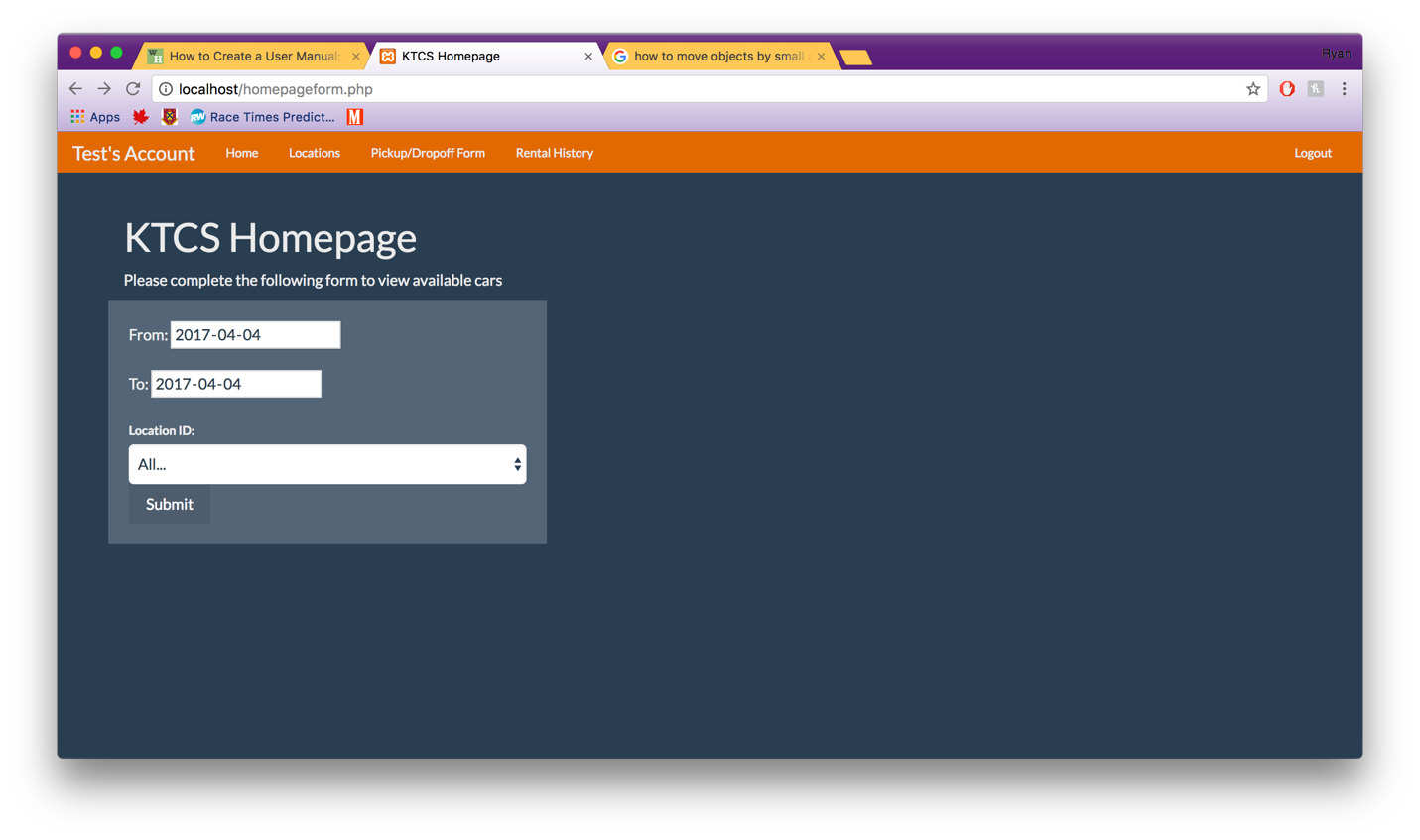
### Login

1. Enter login information as you entered it in the register page. Click submit to Login.

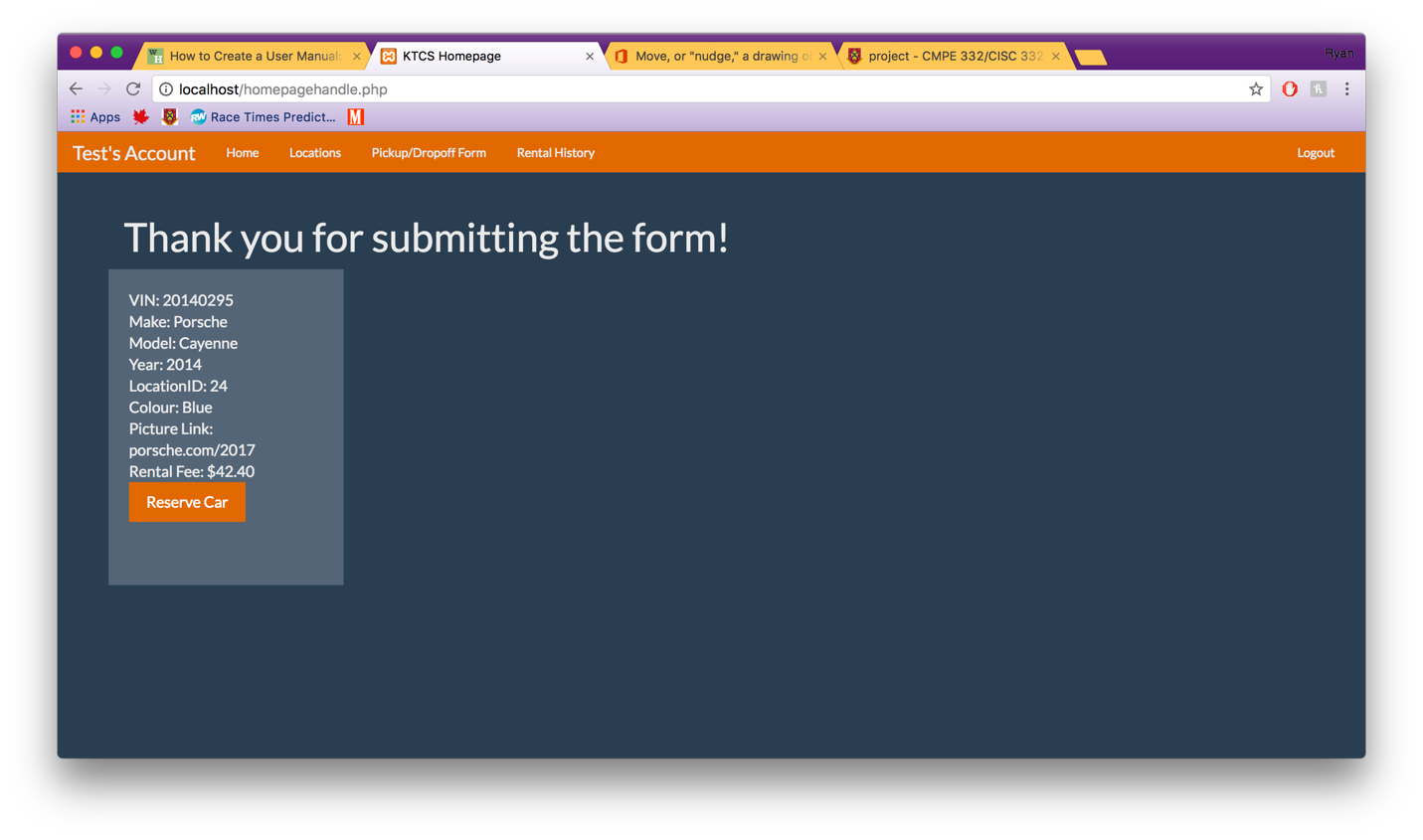


### Find all cars to rent on a day

1. On the homepage select the day you would like to see what cars are available and then in the Location ID dropdown select the “All…” option and click submit.

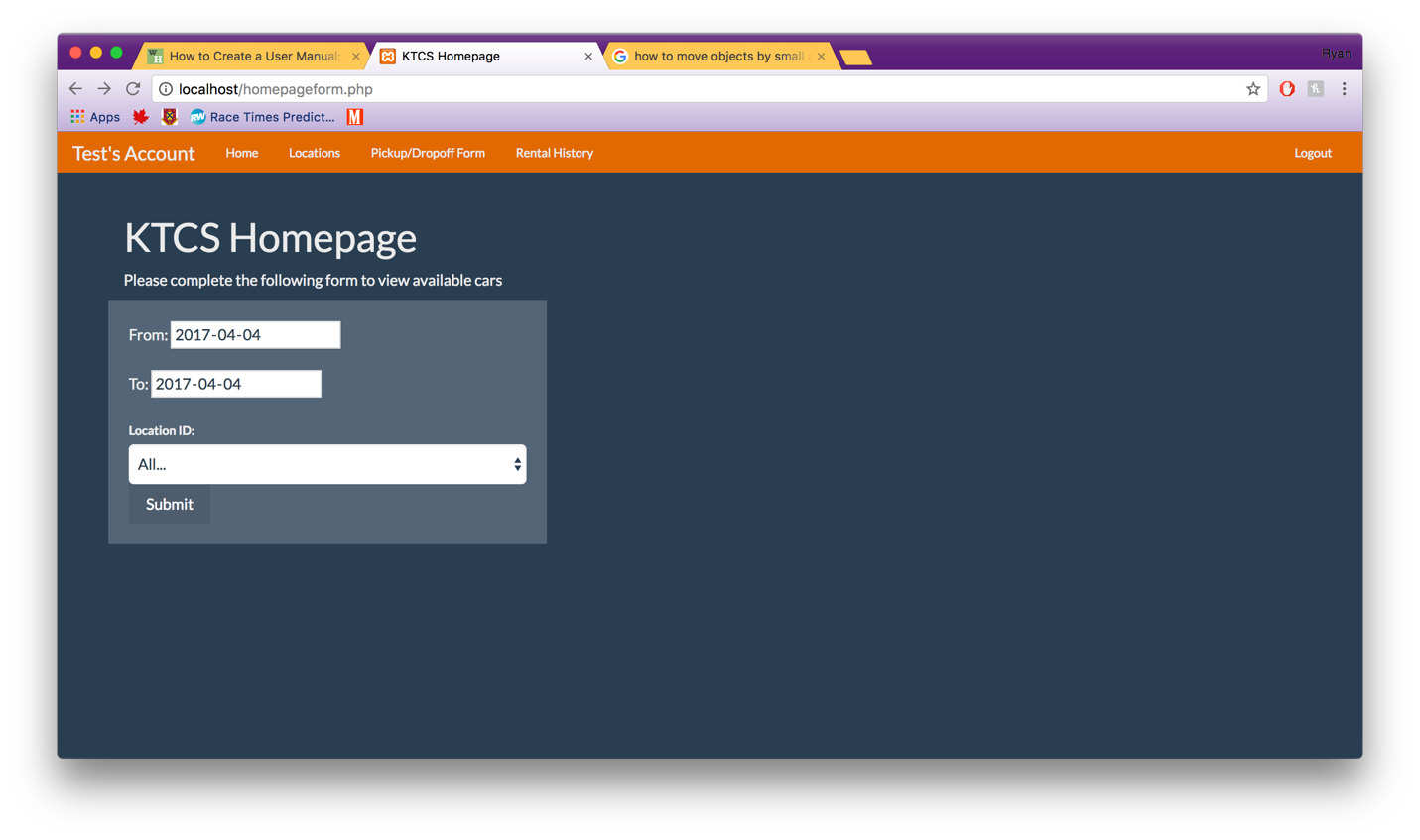


1. This will show you all the available cars for the date range you selected.

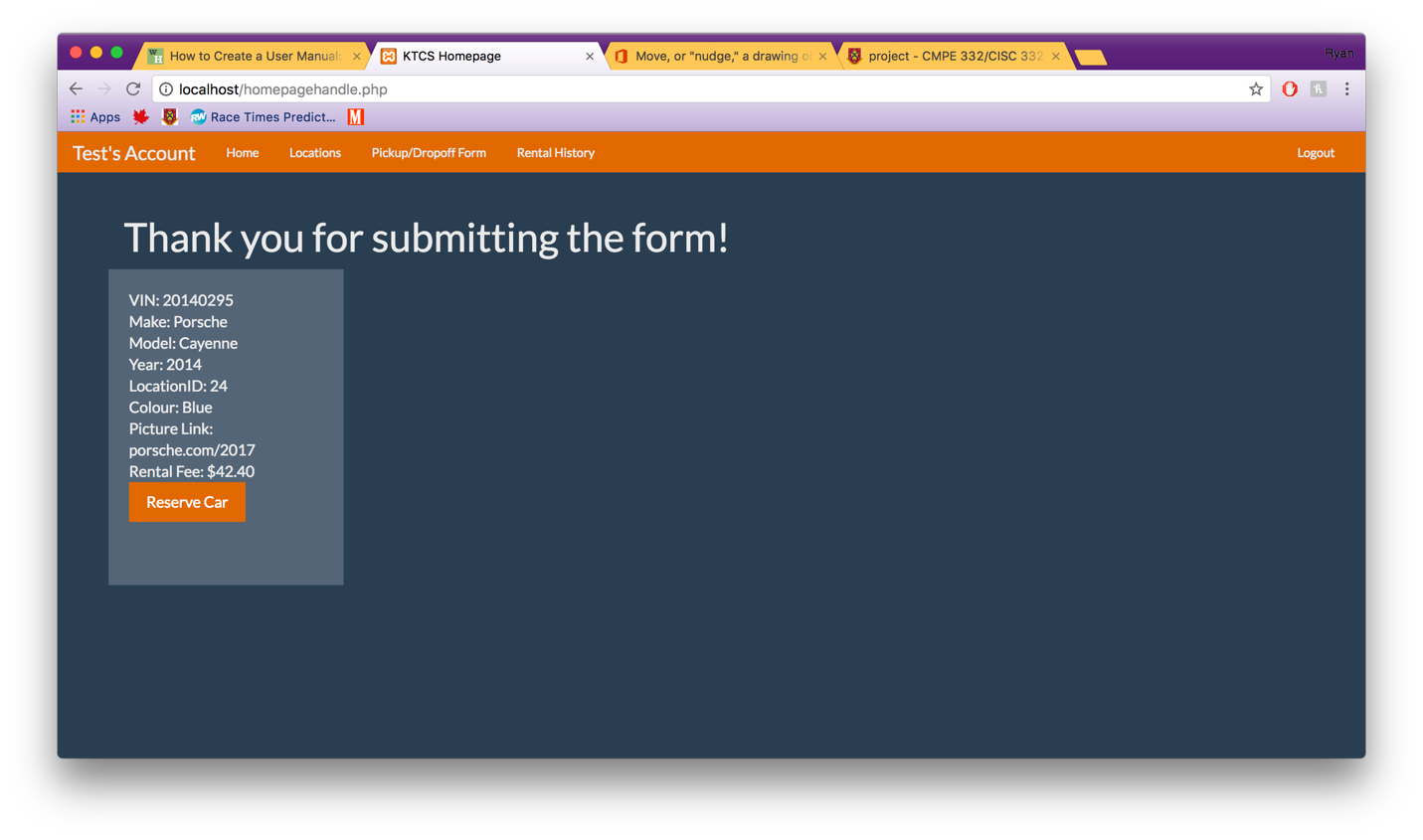


### Reserve a Car

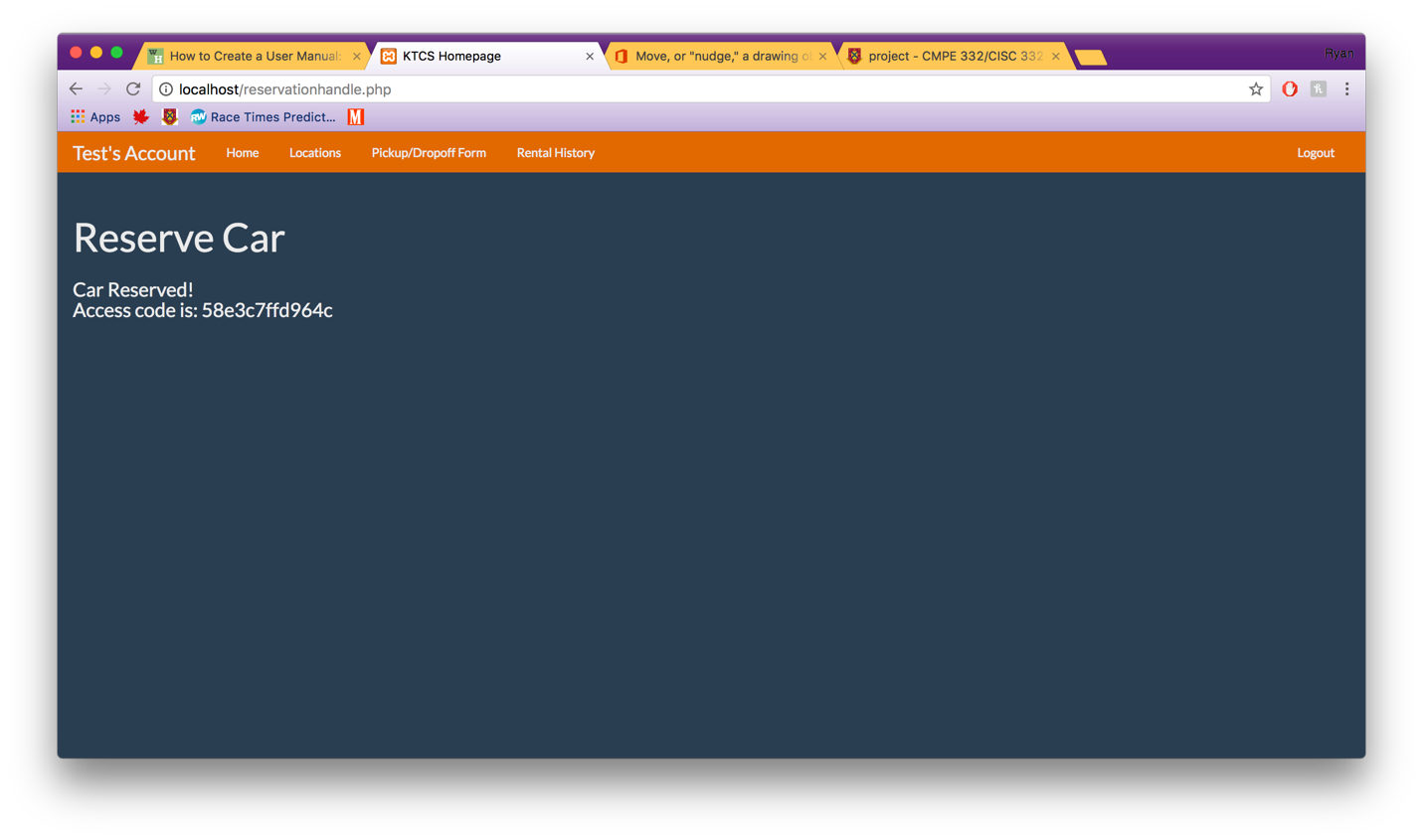
1. On the homepage select the day you would like to see what cars are available and then in the Location ID dropdown select the location you would like to see



1. Once you have found a car you would like to reserve for the previous dates you selected click the Reserve button

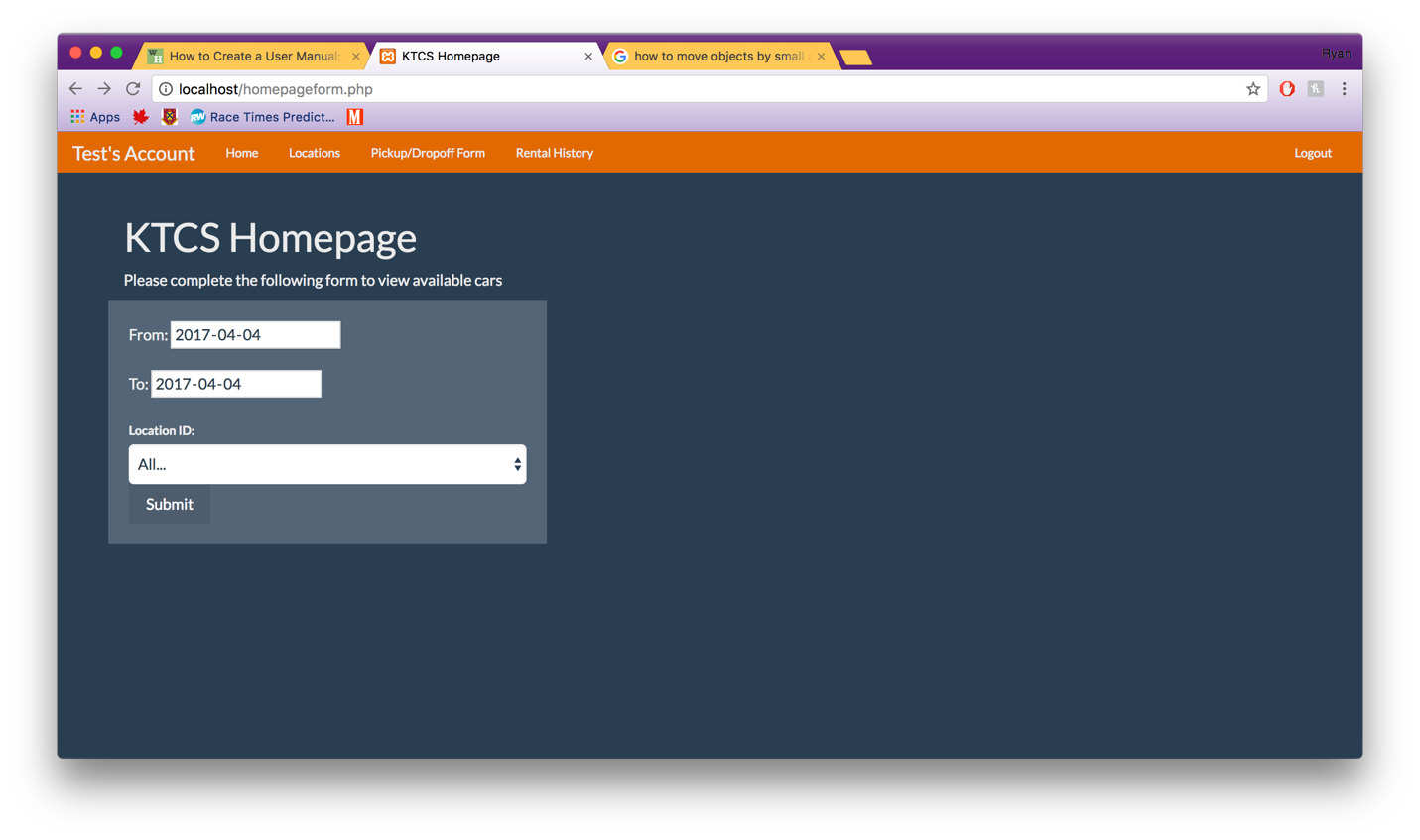


1. Once submitted it will give you an access code which you will need to get into the car so save it.



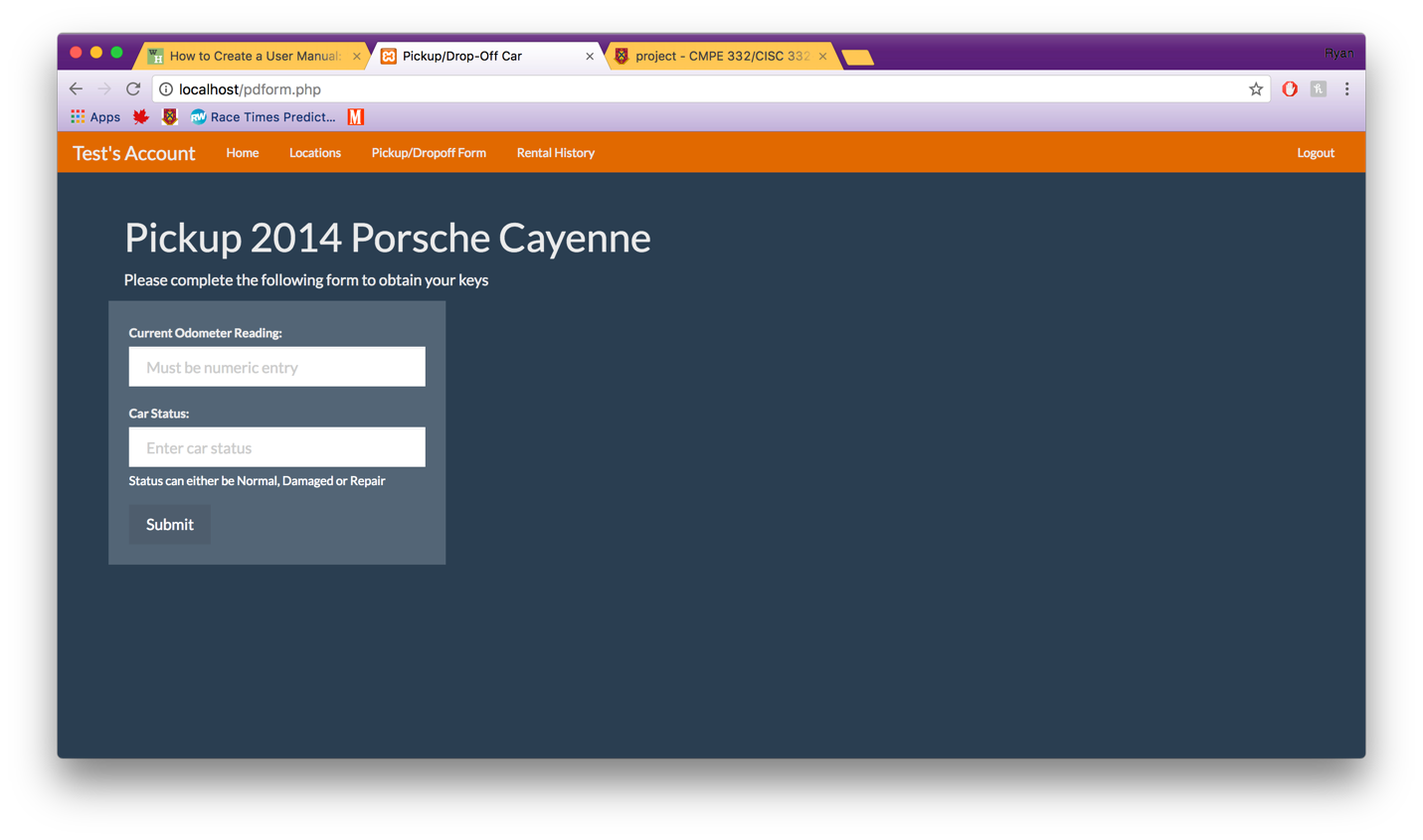
### Find all Locations

1. Click the Locations tab at the top of any page.



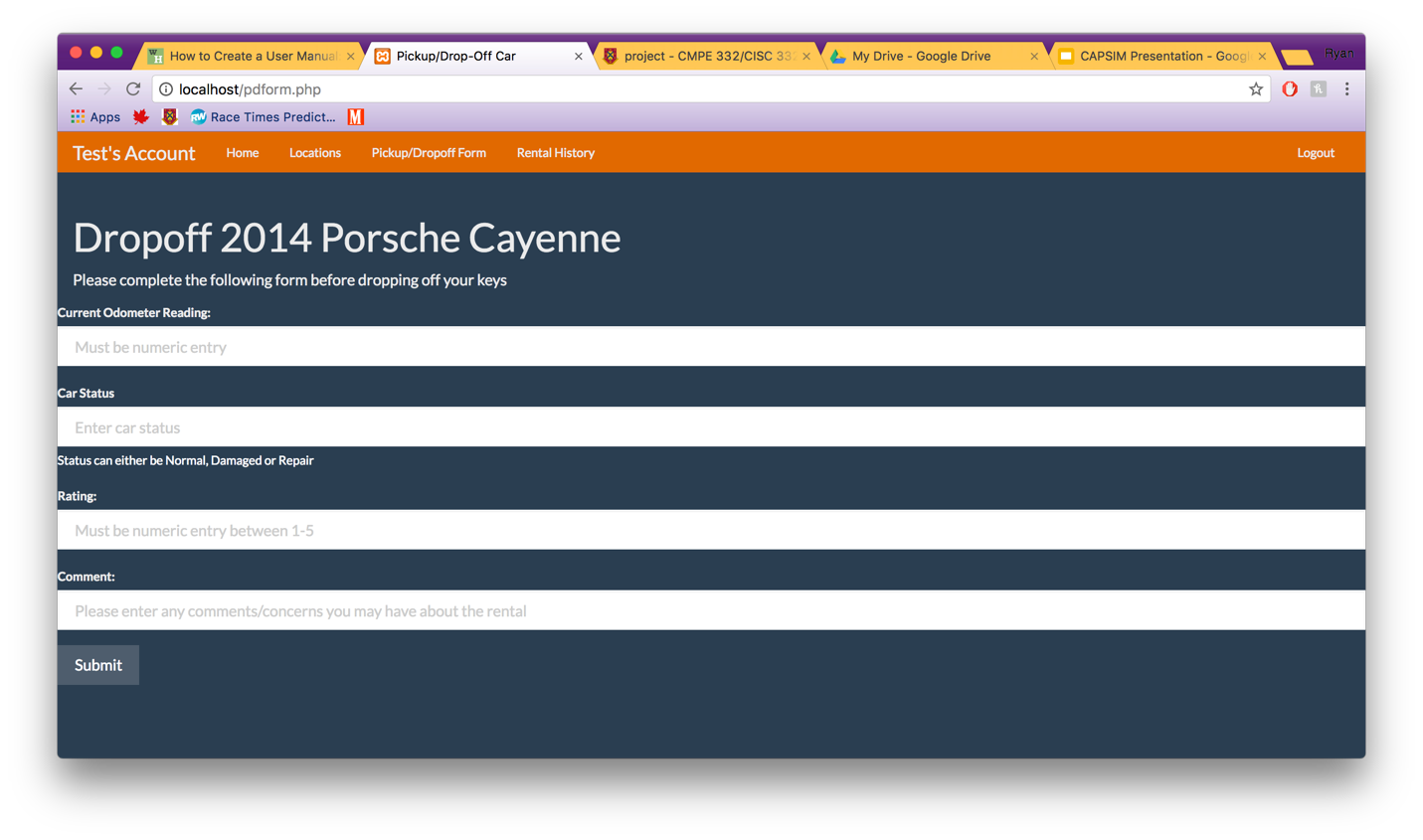
### Pick up a Car

1. Click on the Pickup/Dropoff tab at the top it will bring you to the following page. Where you fill out the odometer reading and car status and click submit. Then you are set to drive



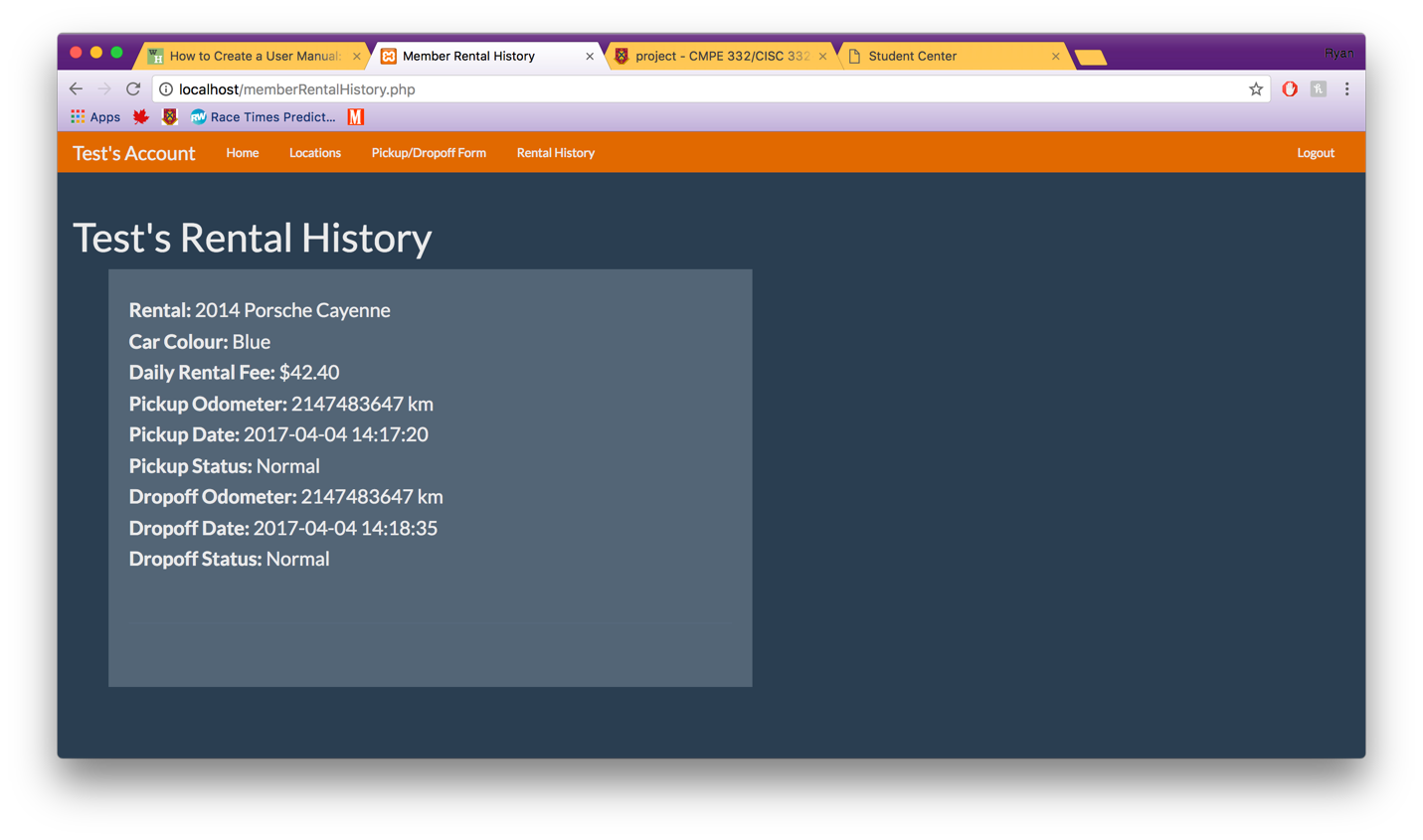
### Drop off Car and Provide Feedback

1. Once you have completed the pickup form you can reload the pickup/dropoff form tab which will then load the dropoff form, like the one below. Then complete the form and click submit to complete the dropoff. You can provide feedback on this screen as well.



### Rental History

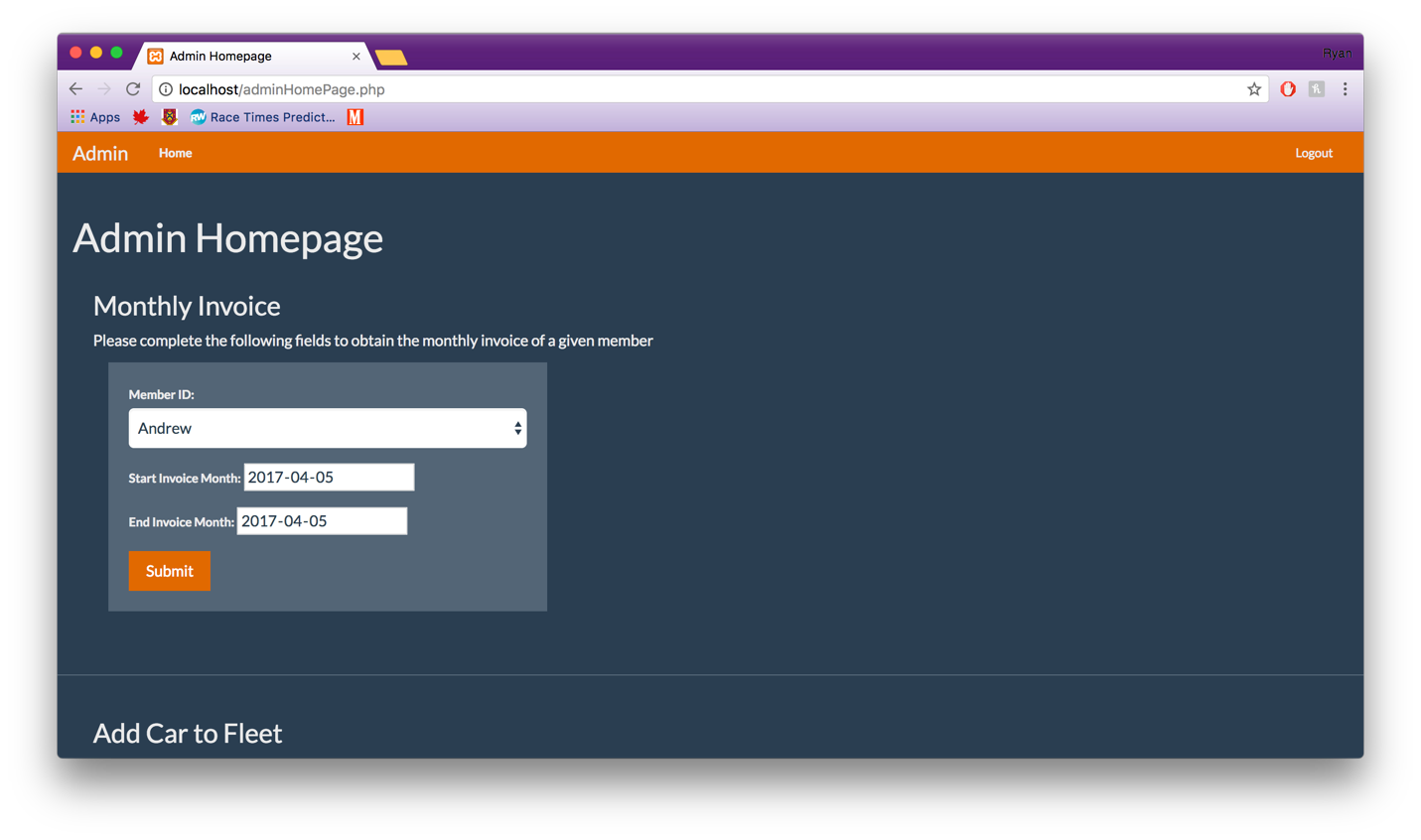
1. Click the Rental History tab and a history of your rentals will appear like the diagram below.



## Admin

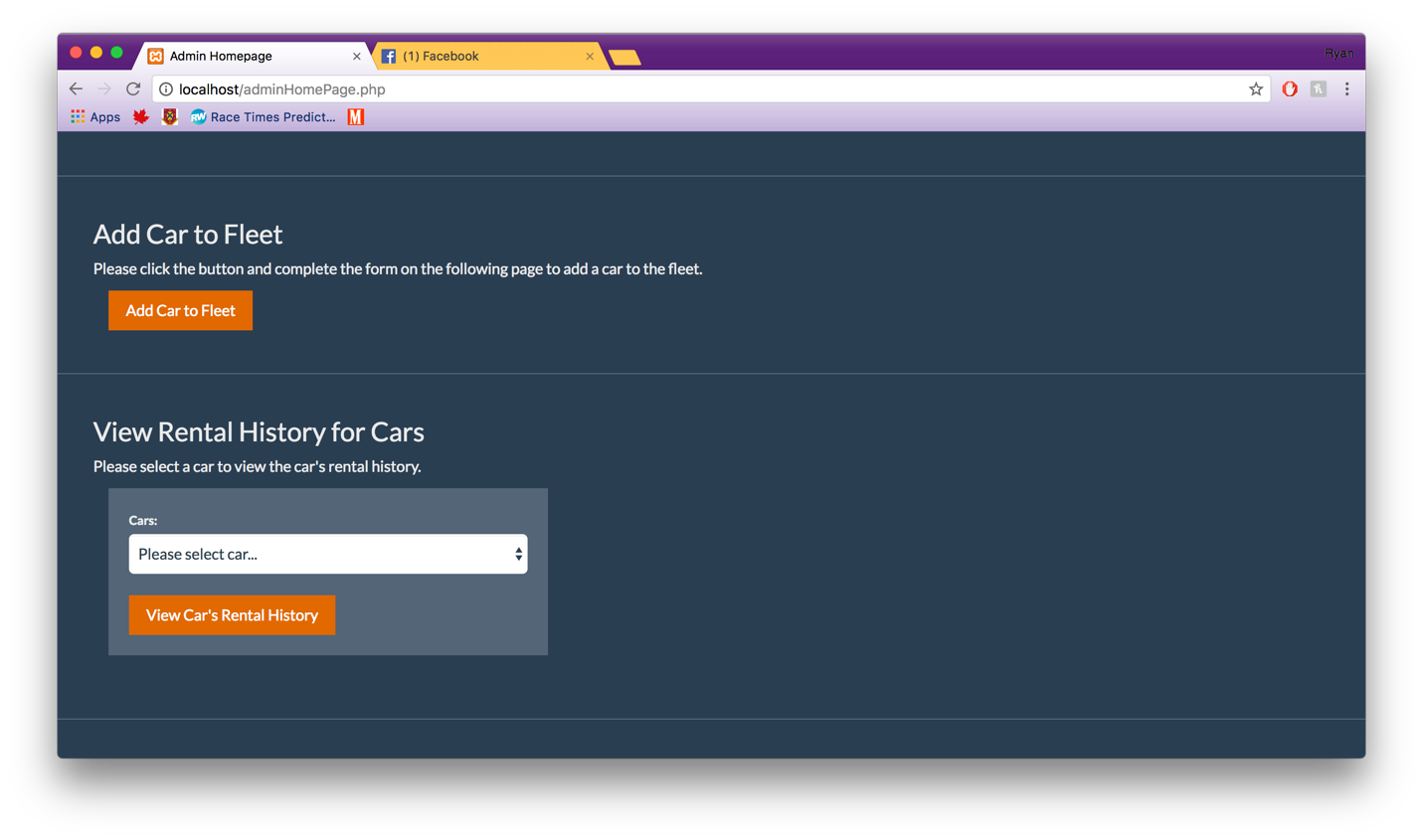
### Generate Monthly Invoice for User

1. On the Admin Homepage under the Monthly Invoice section select the user and month you would like to generate the invoice for, click submit.

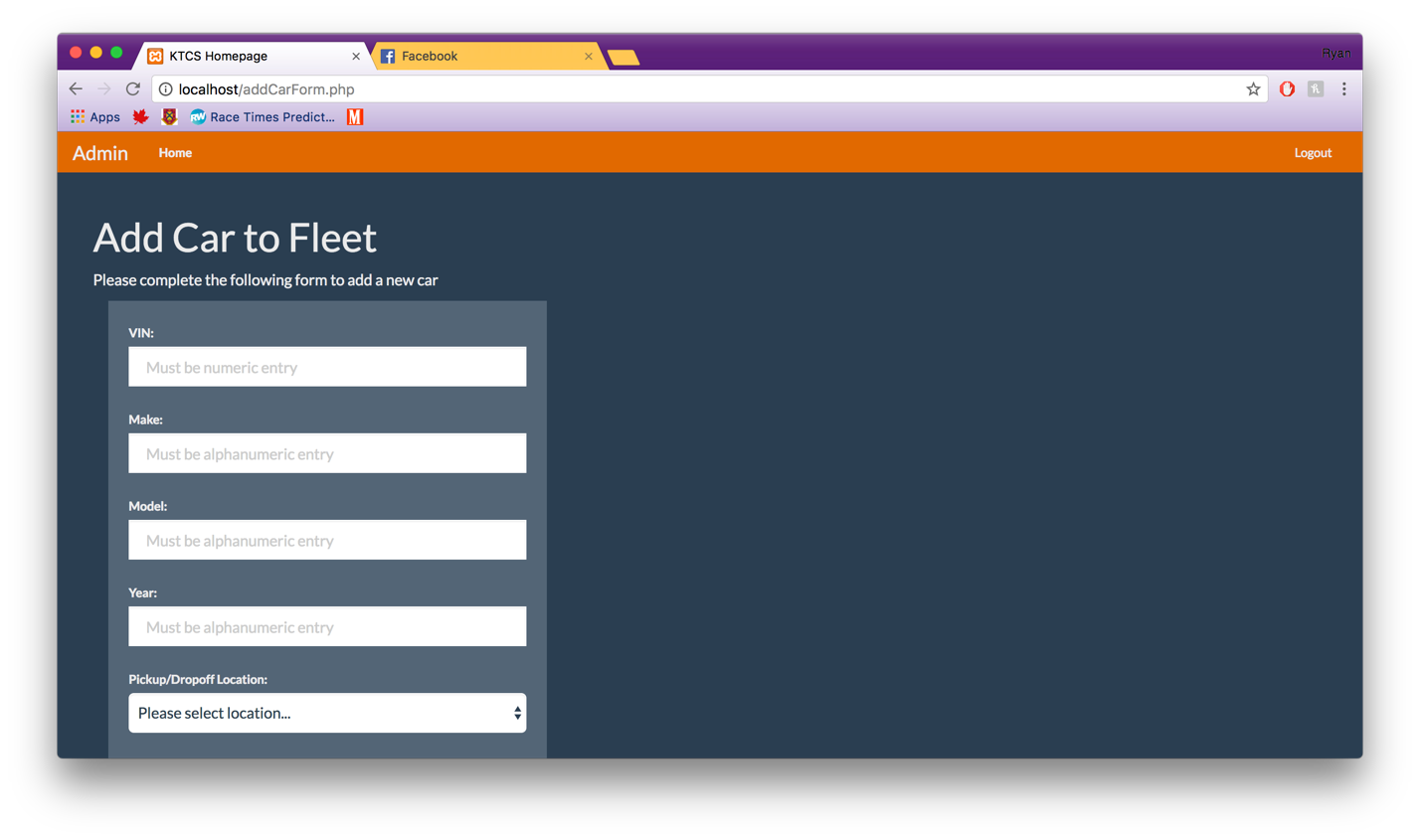


### Add a Car to Fleet

1. Go to the add car button and click it to launch into the form.

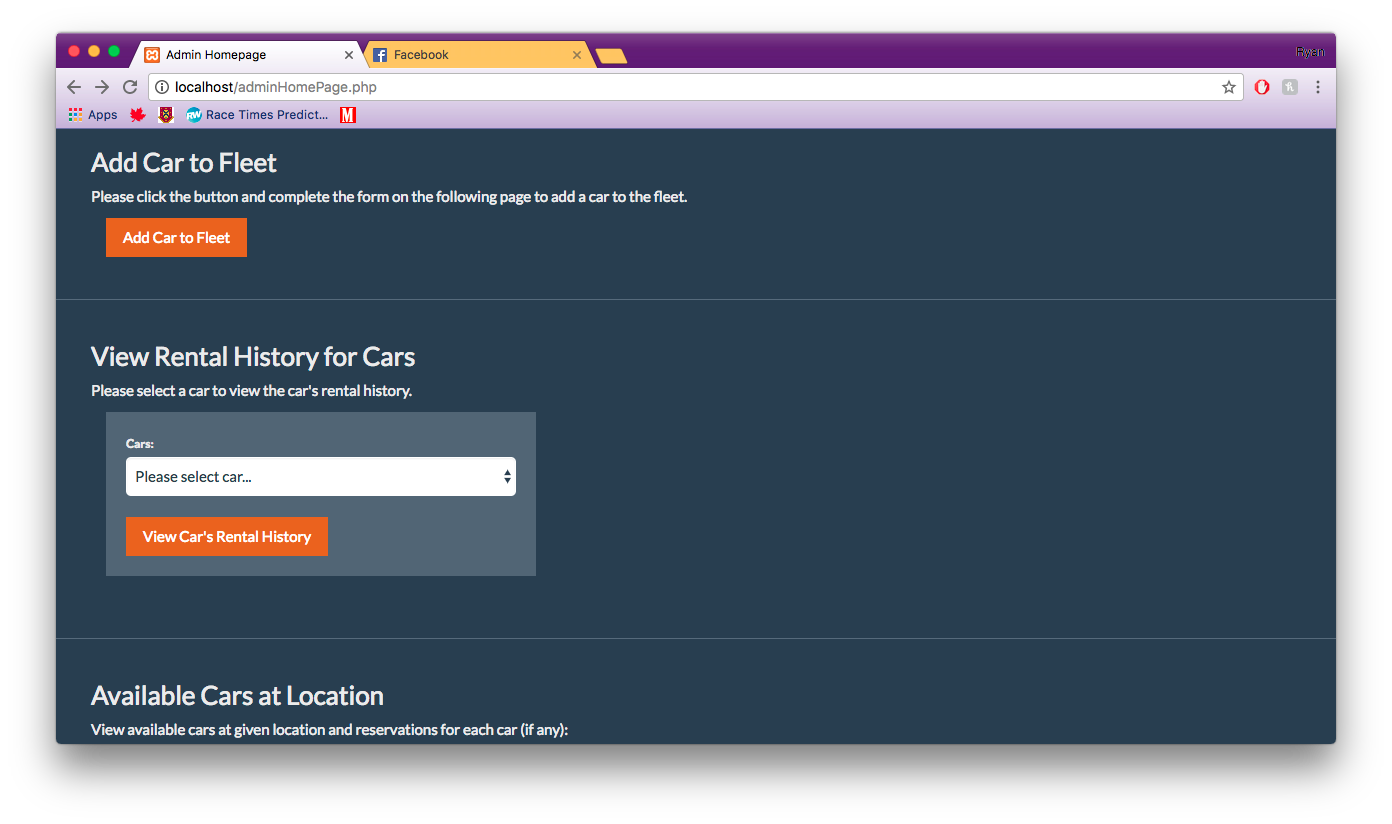


1. Fill in the form and click submit when new car is ready to be added.



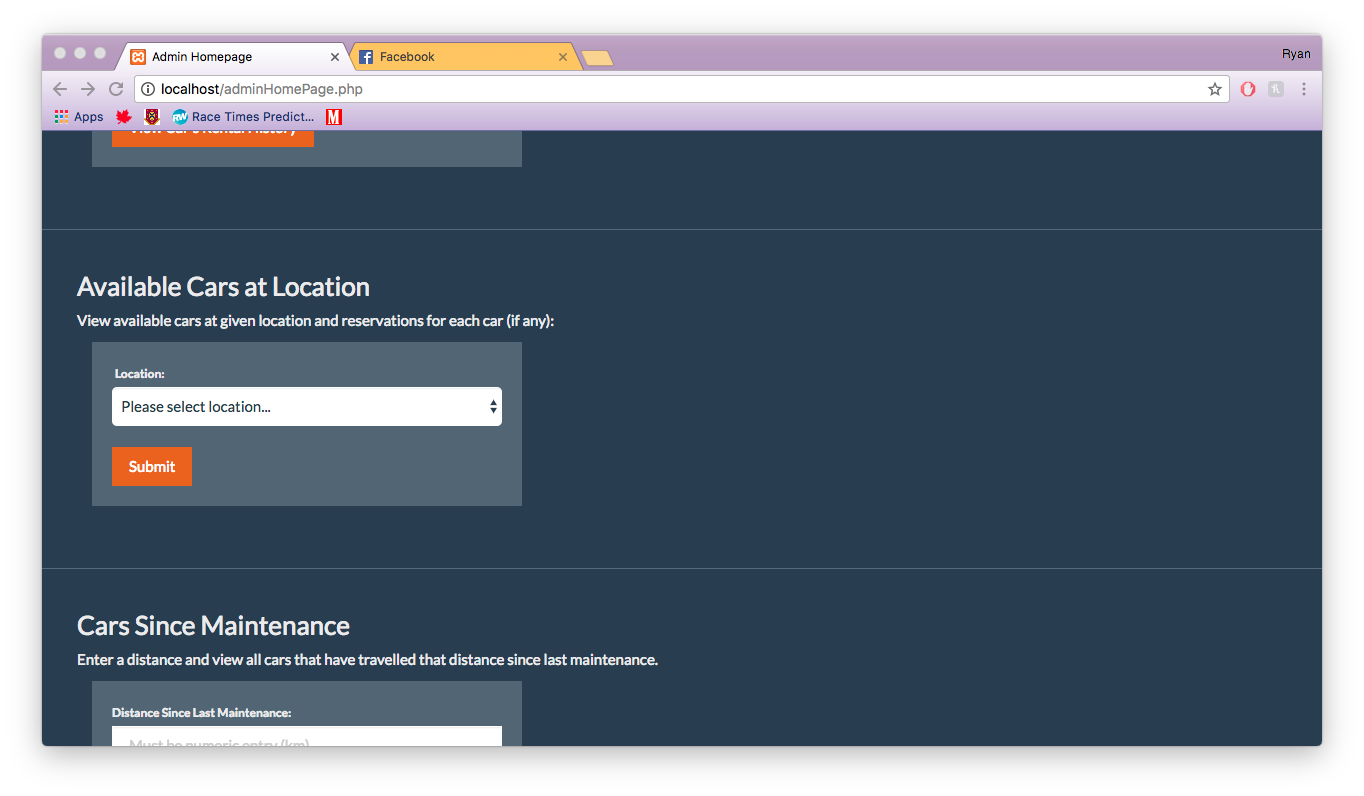
### Show Rental History for Car

1. On the Admin Homepage under the View Rental History for Cars section select the car you would like to history for, click View Car’s Rental History.



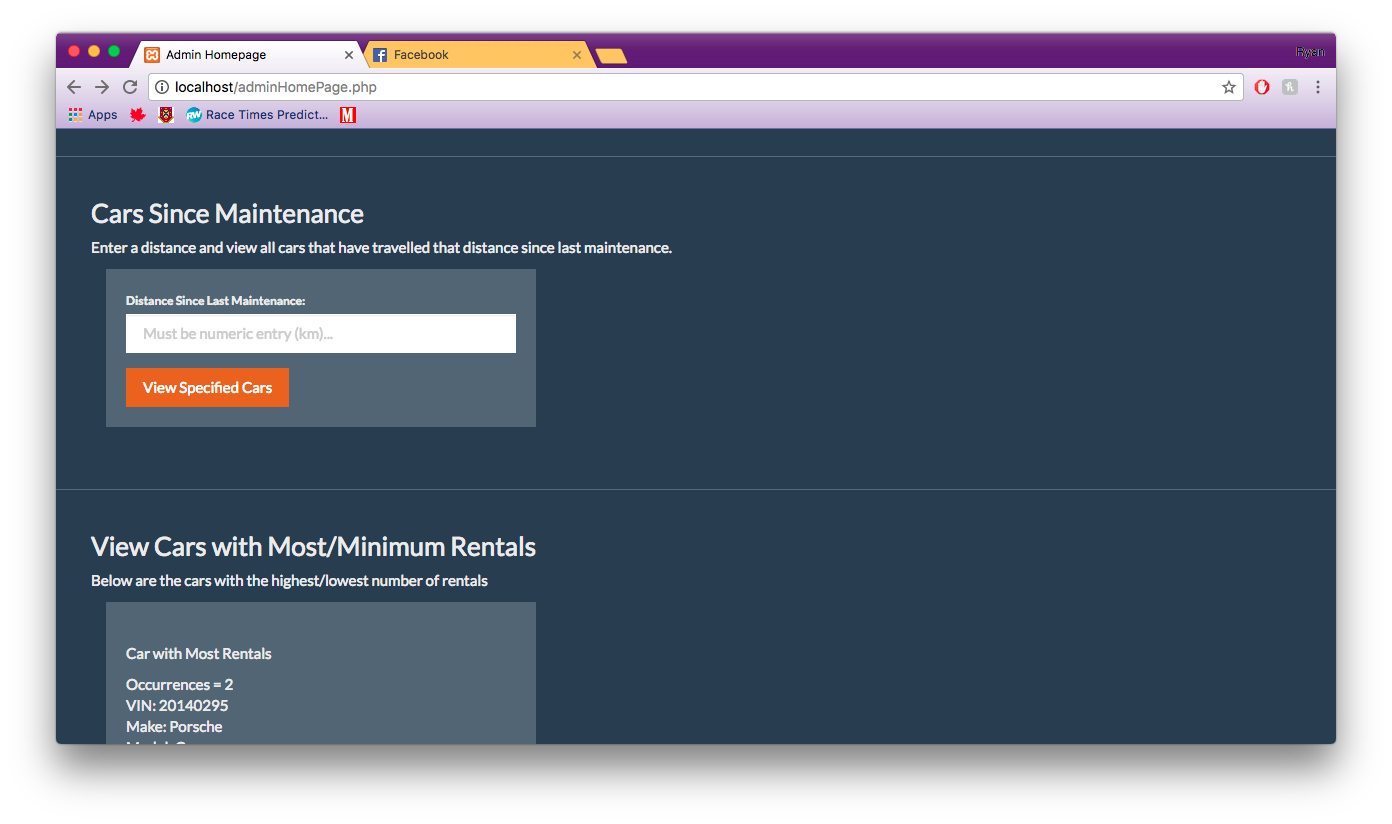
### Show Cars at Location

1. On the Admin Homepage under the Available Cars at Location section select the location you would like to find the cars for, click Submit.



### Show Cars with Distance Since Maintenance

1. On the Admin Homepage under the Cars Since Maintenance section enter the amount of kilometers you would like to see the cars for, click View Specified Cars.



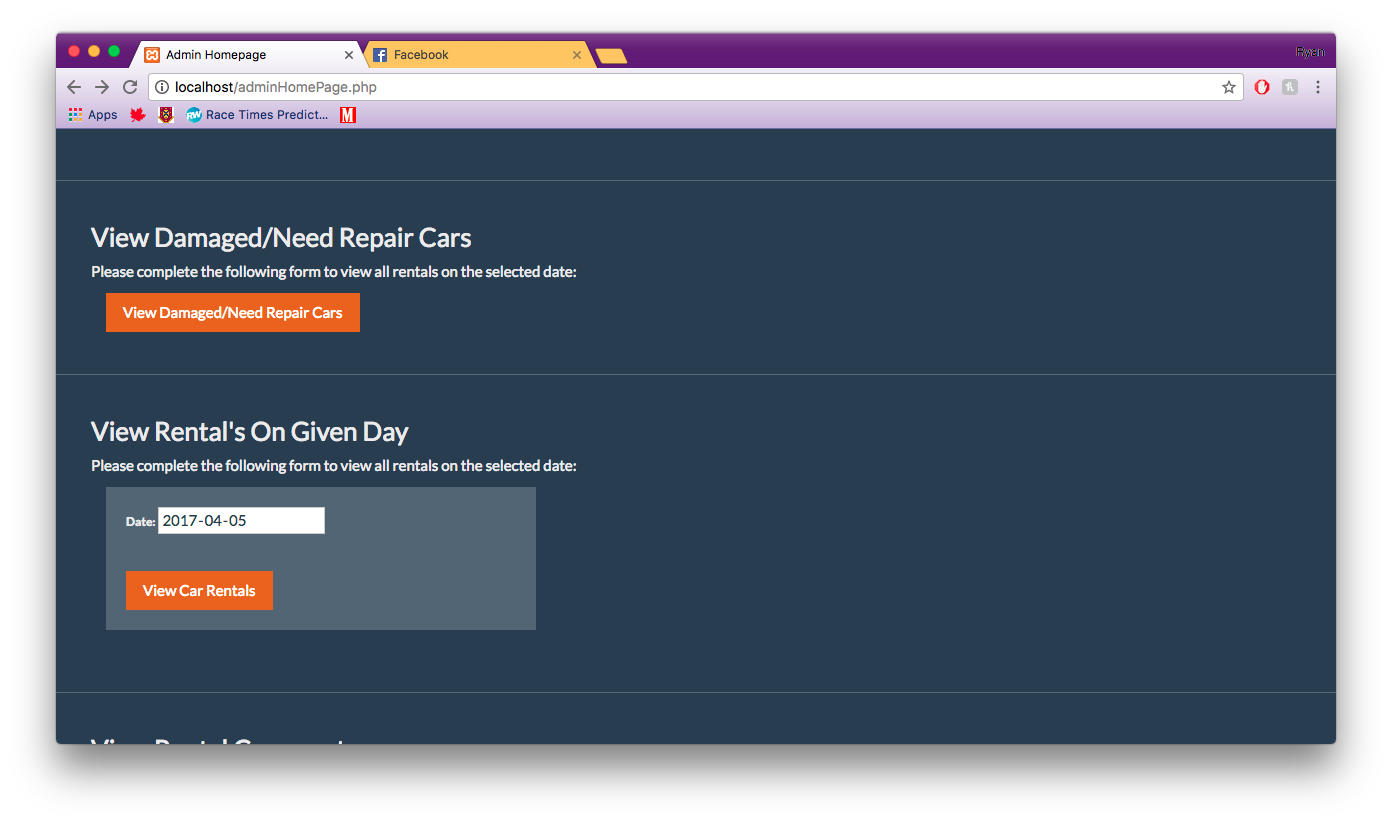
### View Cars with the most and least number of rentals

1. On the Admin Homepage under the View Cars with Most/Minimum Rentals there will be the two cars.



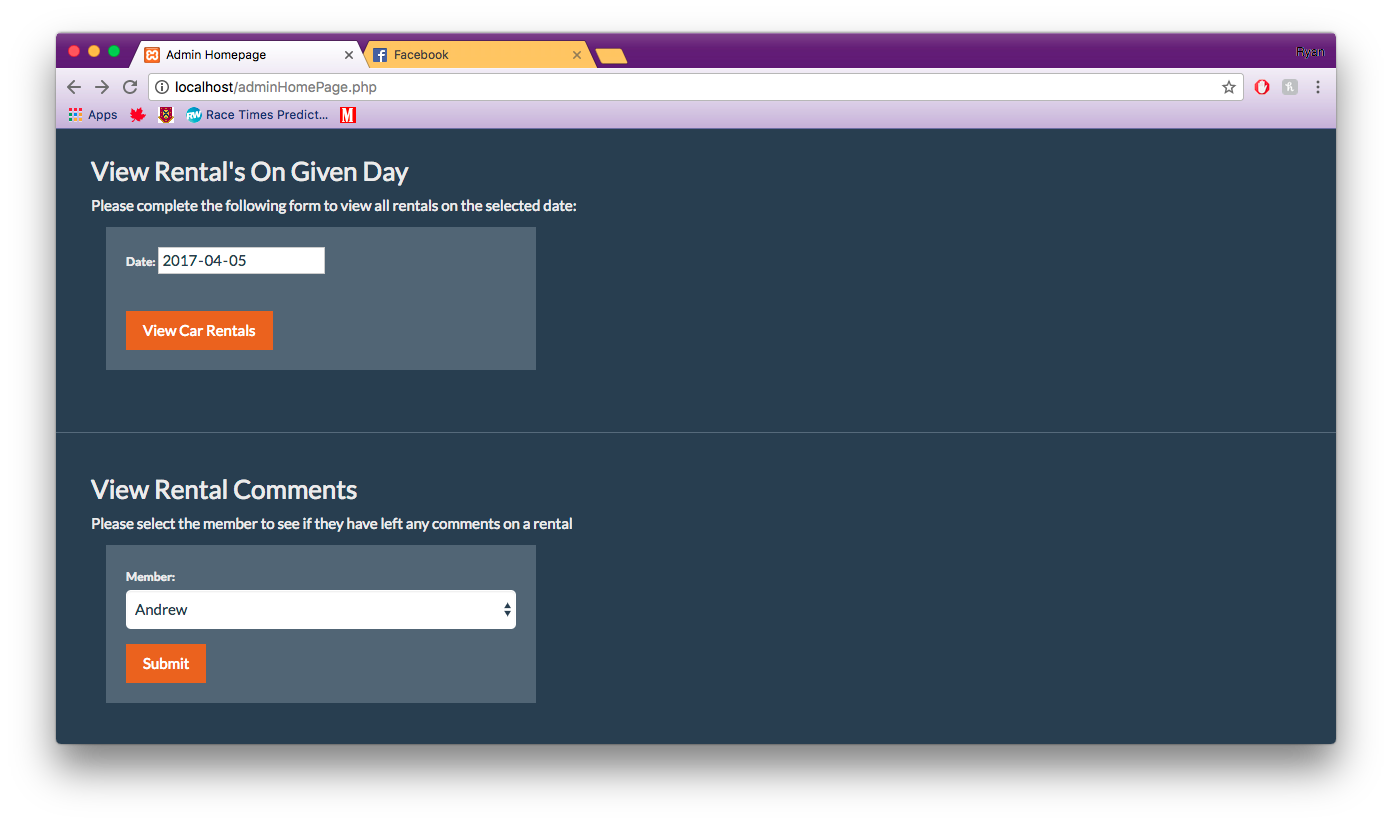
### Generate Monthly Invoice for User

1. On the Admin Homepage under the View Damaged/Need Repair Cars, click the button.



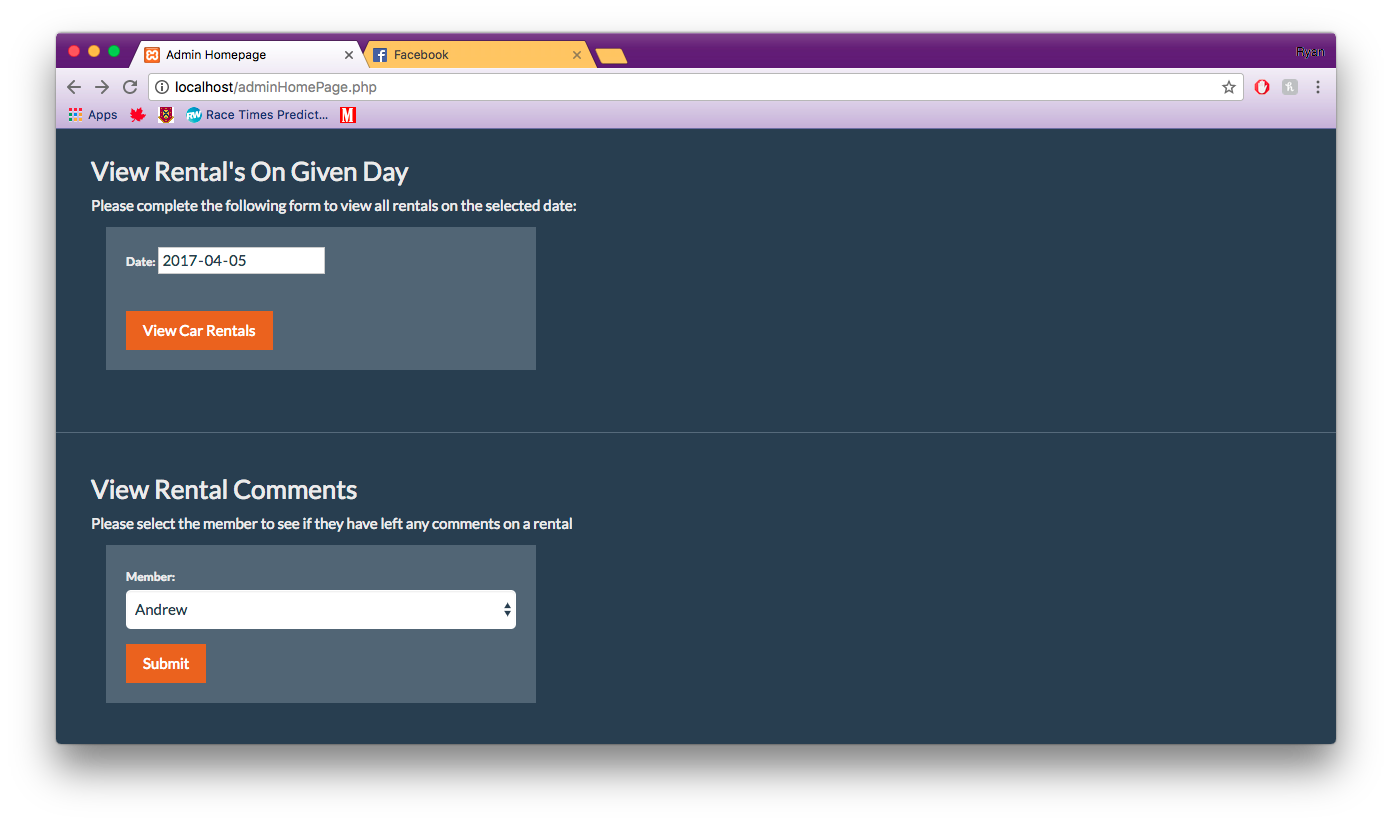
### Show all Reservations on a Day

1. On the Admin Homepage under the View Rental’s on a Given Day section enter the day you would like to see the cars for, click View Car Rentals.



### Respond to a Members Comment

1. On the Admin Homepage under the View Rental Comments section enter the the user you would like to respond too, click click Submit.



# Appendix

## SQL Dumps of All Tables Preloaded with the KTCS\_DB script

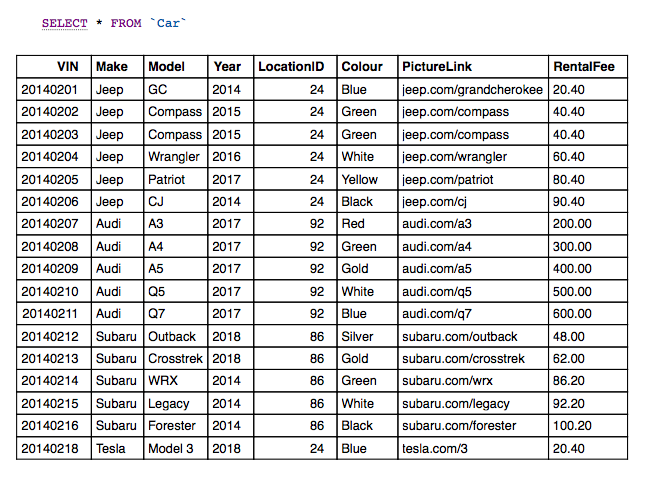
### Admin Table

The SQL table dump can be seen below.



### Car Table

The SQL table dump can be seen below.



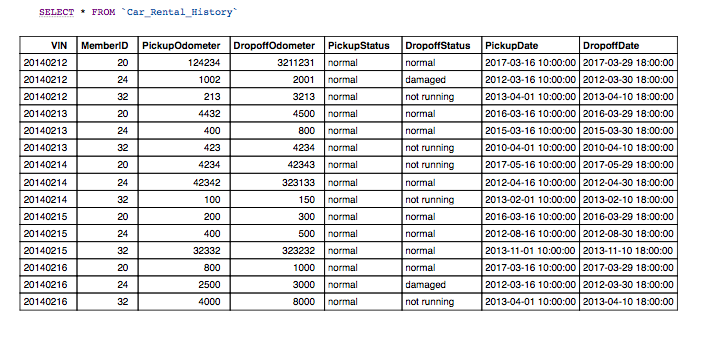
### Car\_Maintenance\_History Table

The SQL dump can be seen below



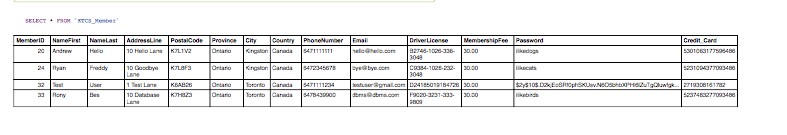
### Car\_Rental\_History Table

The SQL dump can be seen below.



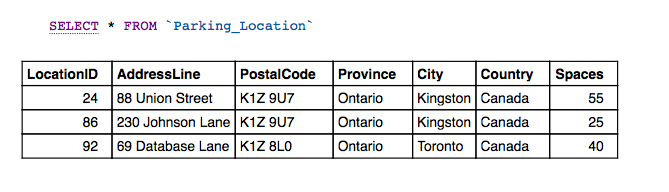
### KTCS\_Member Table

The SQL dump can be seen below.



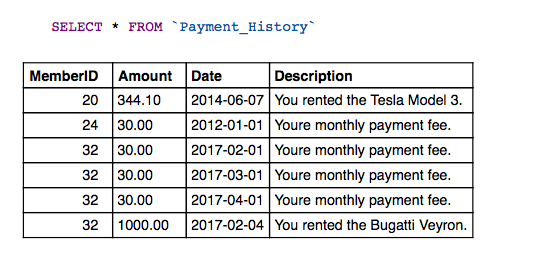
### Parking\_Location Table

The SQL dump can be seen below.



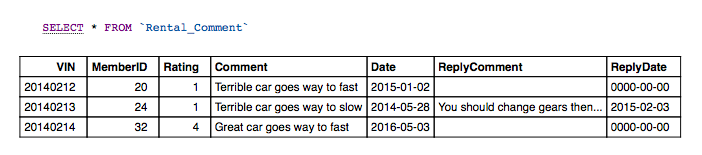
### Payment\_History Table

The SQL dump can be seen below.



### Rental\_Comment Table

The SQL dump can be seen below.



### Reservations Table

The SQL dump can be seen below.

