Database Systems for Software Applications

Group Report containing

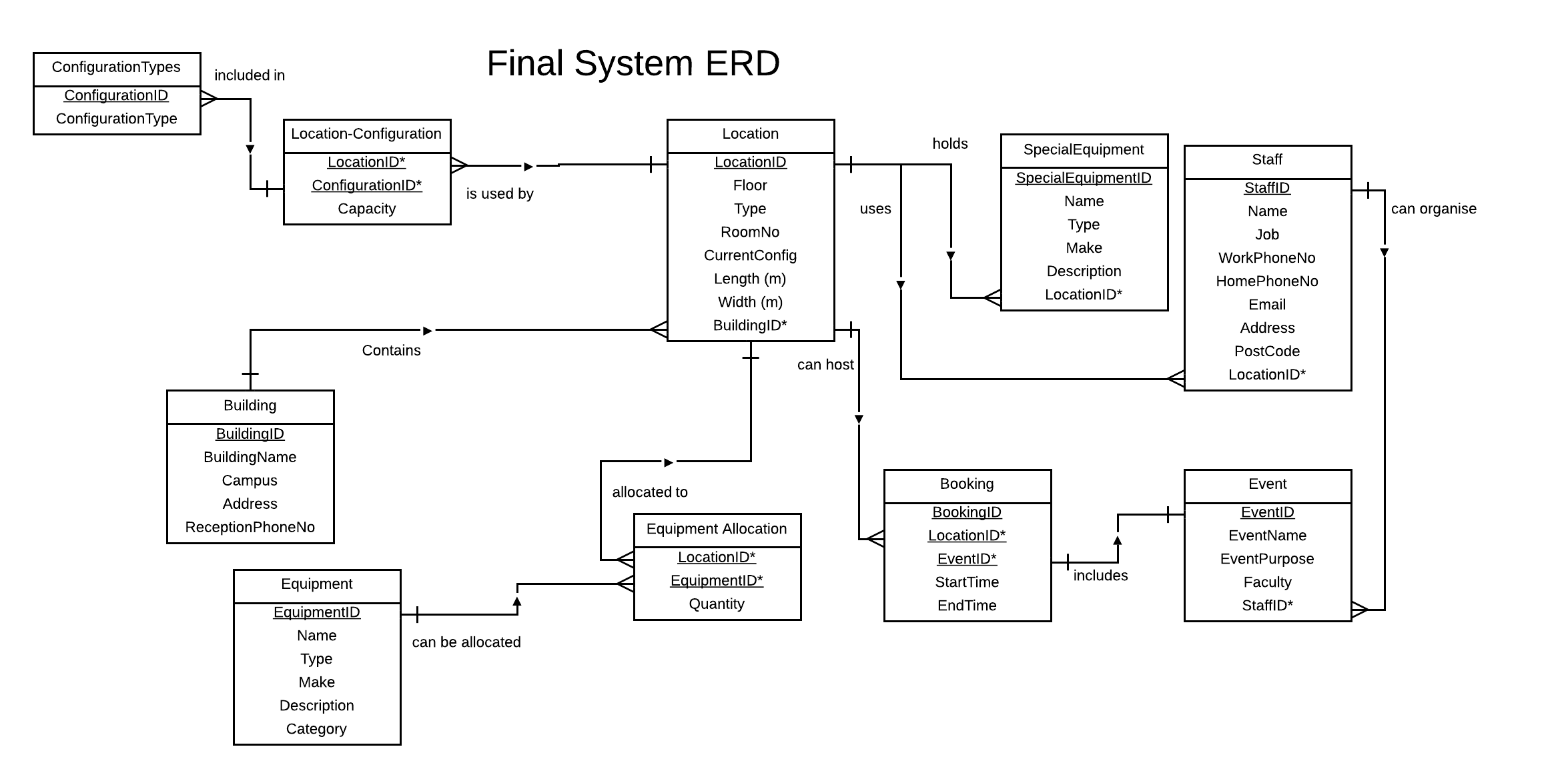
Non-SQL documentation

Group Assignment

Chad Simpson

Luke Ward

Shaun Webb



*Justification of consolidated changes*

The first change that we made when bringing the individual ERD’s together was to remove the OFFICE table that Luke had in his initial diagram. The entity was not necessary due to the fact that all of the data about the locations should just be held in the LOCATIONS table and would just be duplicated in the OFFICE table. Another change was to relate the STAFF and OFFICE relationship to a relationship between STAFF and LOCATION consistent with the rest of the group’s feedback. Initially we had it as a one-to-one relationship, but did not consider the possibility that there could be several staff in a single office, which led us to change the relationship to one-to-many.

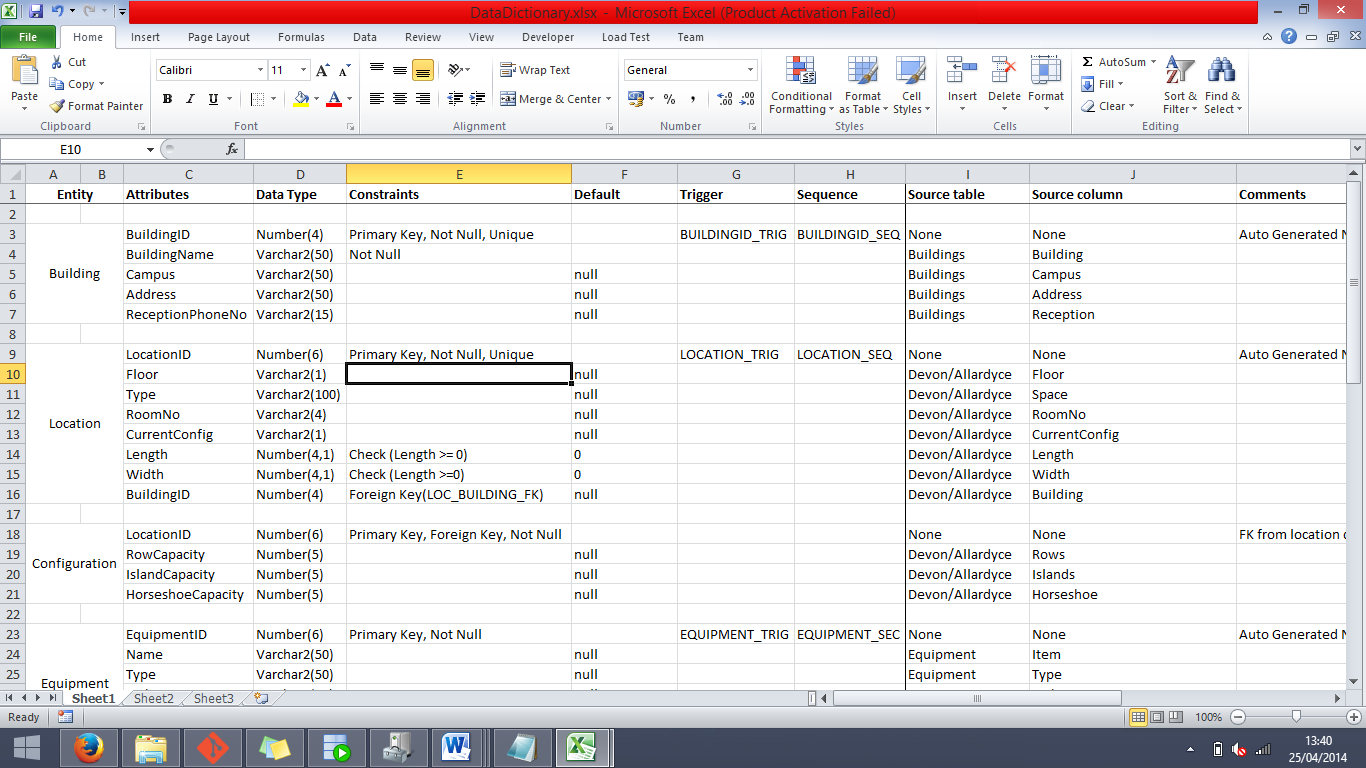
The next major change that we decided to make was to the CONFIGURATION table, and that change involved stripping most of the attributes that were in there previously, and creating another table which would form a many-to-many relationship between CONFIGURATIONTYPES and LOCATION, thus making it necessary to have an intermediate table between the two. This table took form as CONFIGURATION, allowing each location to have many types of configuration, and each of these pairs have a capacity according to the location and the configuration type for that entry.

Changes were made to accommodate and relate both events themselves and a booking system for the events. In our individual ERD’s, we either accommodated for bookings alone or events alone, so we more or less took the events and bookings from all of our versions and integrated the best features. The booking table simply joins the ID’s from the LOCATION table and the EVENT table, creating a composite key.

Some of the slightly smaller changes to the final ERD that we made were to fill in gaps, missing attributes in the STAFF table, adding ones for things such as home and work phone number.

We also took a look at a few of the relationships throughout the ERD, revising some of them. The most notable change was to change the relationship between the configurations and location. Before it was many locations to one configuration, but after further thought it was decided to change to many configurations to one location, allowing each location to have its own unique set of configurations if necessary.

**Database Specification Comments/Documentation**

This piece of documentation will go over every section of the database specification and provide comments on each column and significant constraint in the entity fields.

*(Note: The names of Primary Keys and Foreign Keys are listed next to the comments)*

**BUILDING**

***Attributes***

BuildingID (BUILDING\_PK) – The purpose of this column is to provide a unique identifier generated from a sequence for the Building entity to serve as a primary key.

BuildingName – The purpose of this column is to store the name of the building.

Campus – The purpose of this column is to store the campus in which the associated building is located in.

Address – The purpose of this column is to store the address, most likely just the street name and number, of the associated building.

ReceptionPhoneNo – The purpose of this column is to hold the phone number that is linked to the reception in the building.

***Constraints***

*Sequences*

BUILDING\_SEQ – The purpose of this sequence is to populate the BuildingID column whenever a new entry is inserted into the Building entity.

*Triggers*

BUILDING\_TRIG – The purpose of this trigger is to act as an activator for the BUILDING\_SEQ sequence, set to trigger after a row of data is inserted into the Building entity.

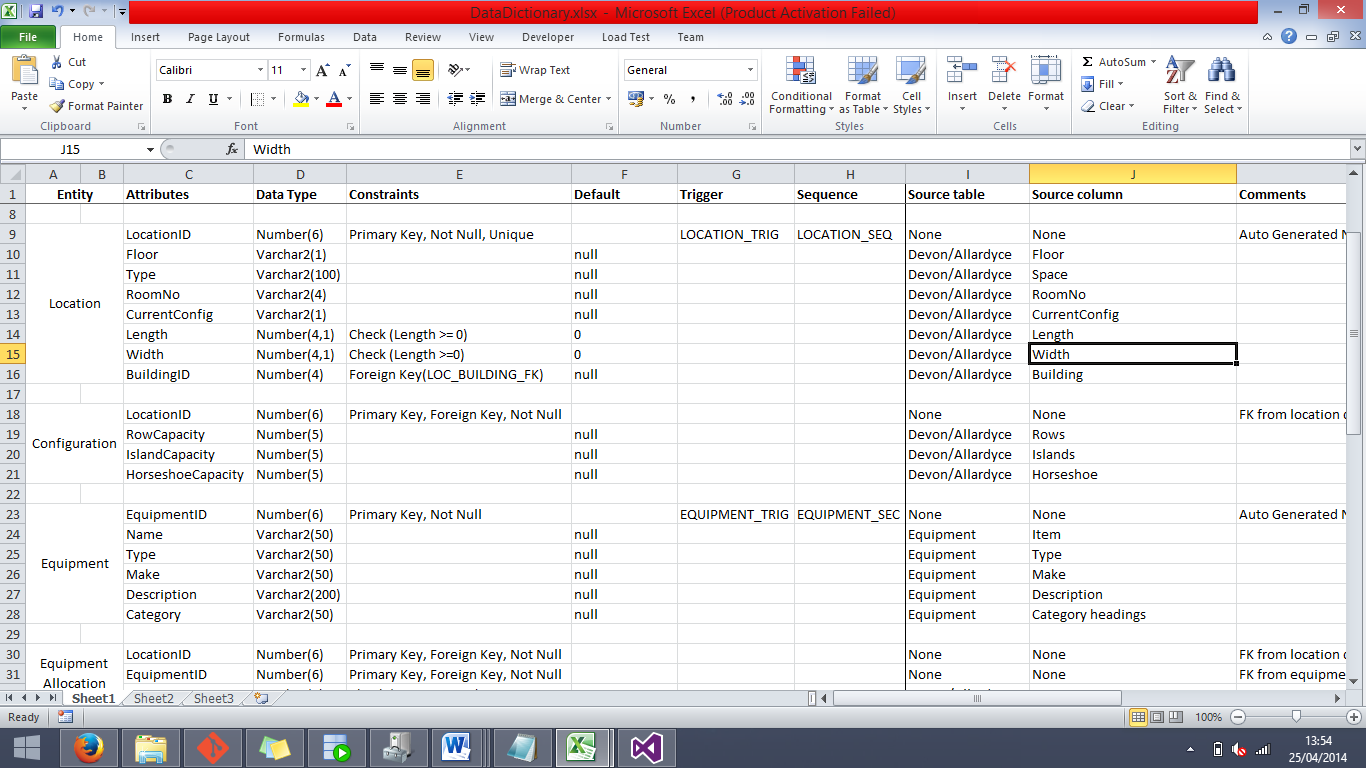
*Other Constraints*

BuildingID (Unique) – Due to the fact that this field is a Primary Key, it is by default unique.

BuildingName (Not Null) – We made this field Not Null because it would not make sense to add a building into the Building entity without even supplying it with a name, making it a redundant entry.

Campus, Address, ReceptionPhoneNo (Default Null) – These pieces of information about the building are less vital so we thought it would make sense to have the option to leave them as null to be added in later if desired.

**LOCATION**



**Attributes**

LocationID (LOCATION\_PK) – The purpose of this column is to provide a unique identifier generated from a sequence for the Location entity to serve as a primary key.

Floor – The purpose of this column is to specify the floor on which each location is on (G, 1, 2 etc).

Type – The purpose of this column is to specify the types of locations are stored in this entity. Examples of this are ones such as Classroom, Lecture Hall, Corridor etc.

RoomNo – The purpose of this column is to hold the room numbers of each location that it is applicable to, note that this only includes locations that class as teaching areas.

CurrentConfig – The purpose of this column is to specify the current configuration of that location, specified as a single letter (H, R, I).

Length – The purpose of this column is to specify the length (in meters) of the location, note that there is a CHECK constraint to ensure that this number cannot fall below 0.

Width – The purpose of this column is to specify the width (in meters) of the location, note that there is a CHECK constraint in place to ensure that this number cannot fall below 0.

BuildingID (LOC\_BUILDING\_FK) – The purpose of this column is to provide information about which building that the location is located in and to create a relationship between the two entities.

***Constraints***

*Sequences*

LOCATION\_SEQ – The purpose of this sequence is to populate the LocationID column whenever a new entry is inserted into the Location entity.

*Triggers*

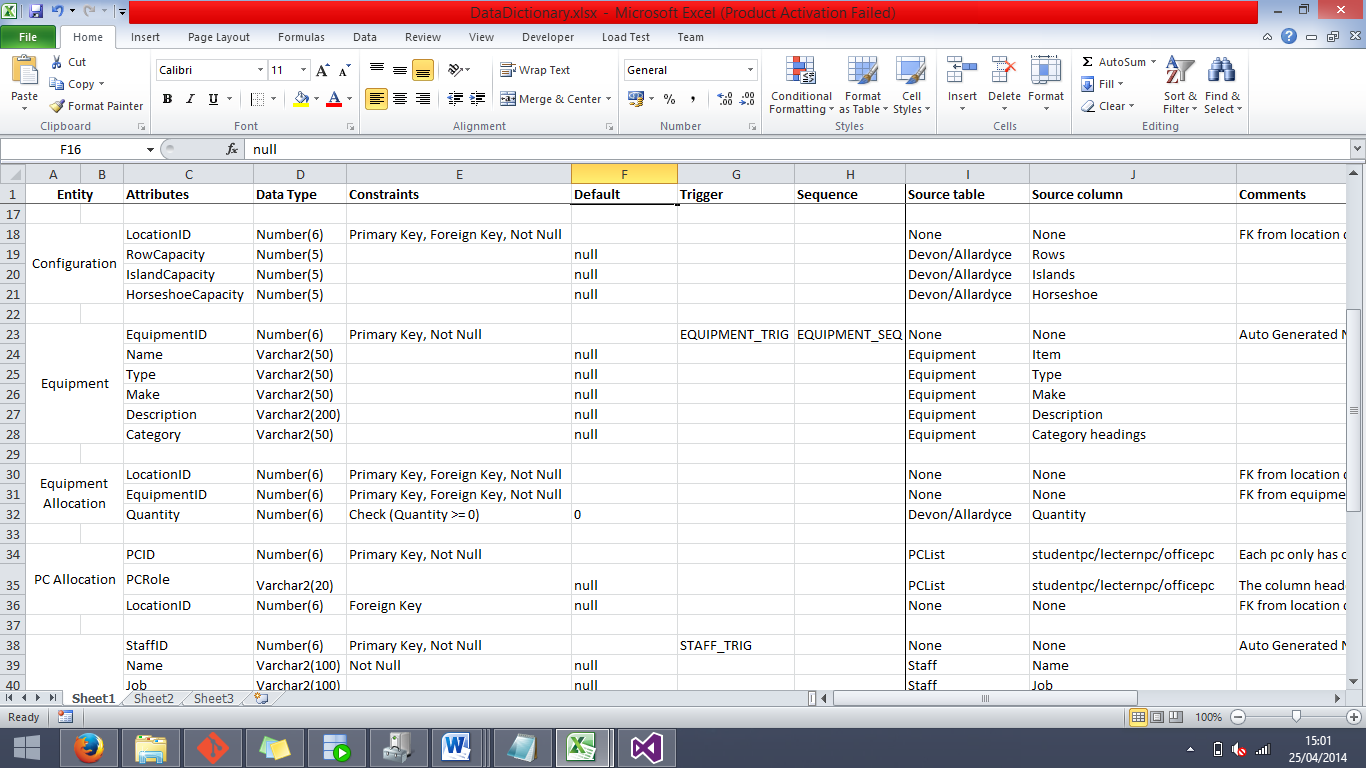
LOCATION\_TRIG – The purpose of this trigger is to act as an activator for the LOCATION\_SEQ sequence, set to trigger after a row of data is inserted into the Location entity.

*Other Constraints*

LocationID (Unique) – Due to the fact that this field is the primary key, it is automatically required to be unique.

Floor, Type, RoomNo, CurrentConfig, Length, Width, BuildingID (Default Null/0) – There is no absolute need for any single piece of data in this table, so everything has a default of null or 0 to let the user choose which information is included in the beginning.

Check (Length, Width) – These checks are in place to ensure that no value below 0 can be entered as location dimensions, as having a location with negative length and width would not make any sense.

**CONFIGURATION**

**Attributes**

LocationID (CONFIGURATION\_PK, CONFIG\_LOCATION\_FK) – The purpose of this column is to both provide a unique identifier for the Configuration entity (as there will not be a single location that has two sets of configurations), and to provide a relationship link between the Location and Configuration entities.

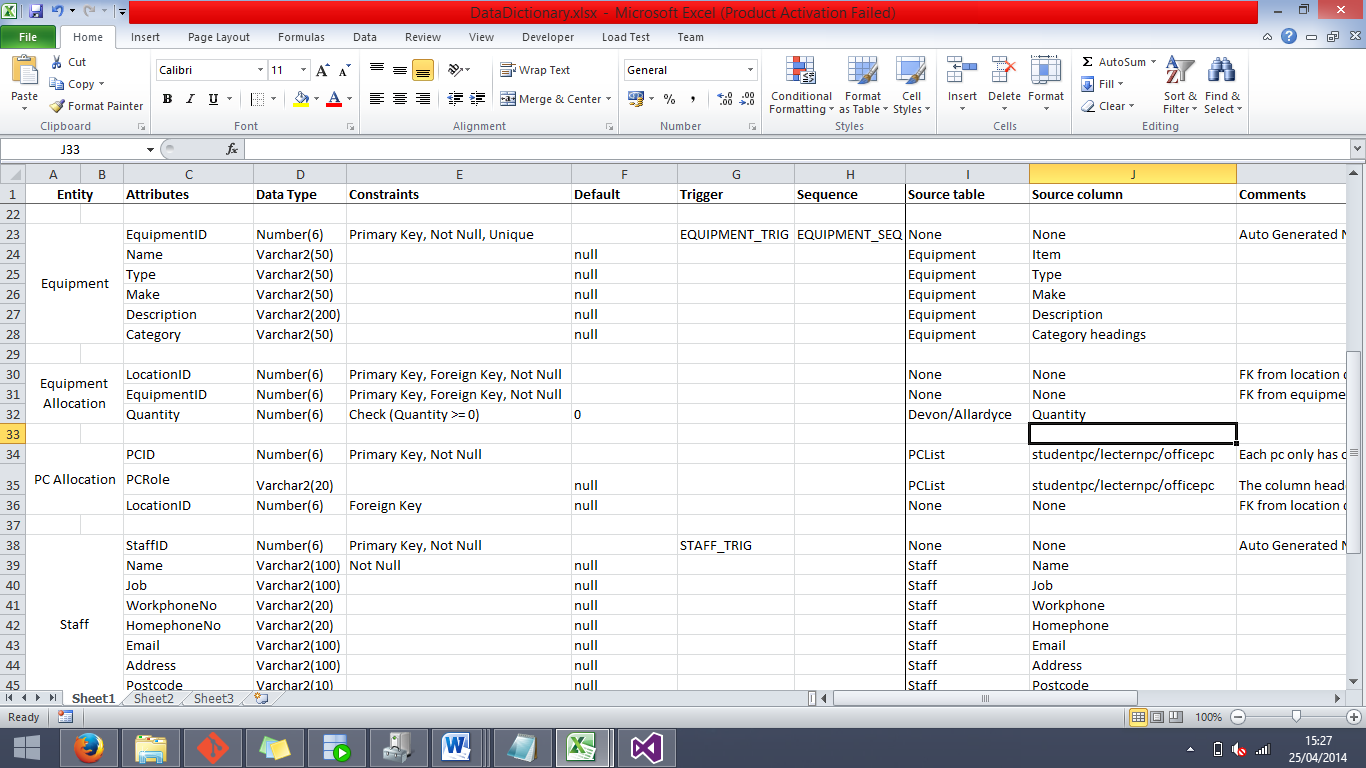
RowCapacity – The purpose of this column is to store the capacity of the specified location as if it had the “Row” table layout in the room.

IslandCapacity – The purpose of this column is to store the capacity of the specified location as if it had the “Island” table layout in the room.

HorseshoeCapacity – The purpose of this column is to store the capacity of the specified location as if it had the “Horseshoe” table layout in the room.

**Constraints**

RowCapacity, IslandCapacity, HorseshoeCapacity (Default Null) – Due to the fact that not every location is going to have a capacity for each of these layouts (some locations will not have valid data for any of the layouts), the easiest solution was to include every location in the beginning and have Default Null for all three of these fields.

**Equipment**

**Attributes**

EquipmentID (EQUIPMENT\_PK) – The purpose of this column is to provide a unique index and serve as the Primary Key for the Equipment entity.

Name – The purpose of this column is to store the names of the pieces of equipment in the entity.

Type – The purpose of this column is to store the types of each piece of equipment, examples of this are table, chair, vacuum cleaner etc.

Make – The purpose of this column is to specify the manufacturer make and model number of the piece of equipment.

Description – The purpose of this column is to provide a short written description of the piece of equipment.

Category – The purpose of this column is to provide a rough category for each piece of equipment, examples of this are: cleaning equipment, furniture etc.

***Constraints***

*Sequences*

EQUIPMENT\_SEQ – The purpose of this sequence is to populate the EquipmentID column whenever a new entry is inserted into the Equipment entity.

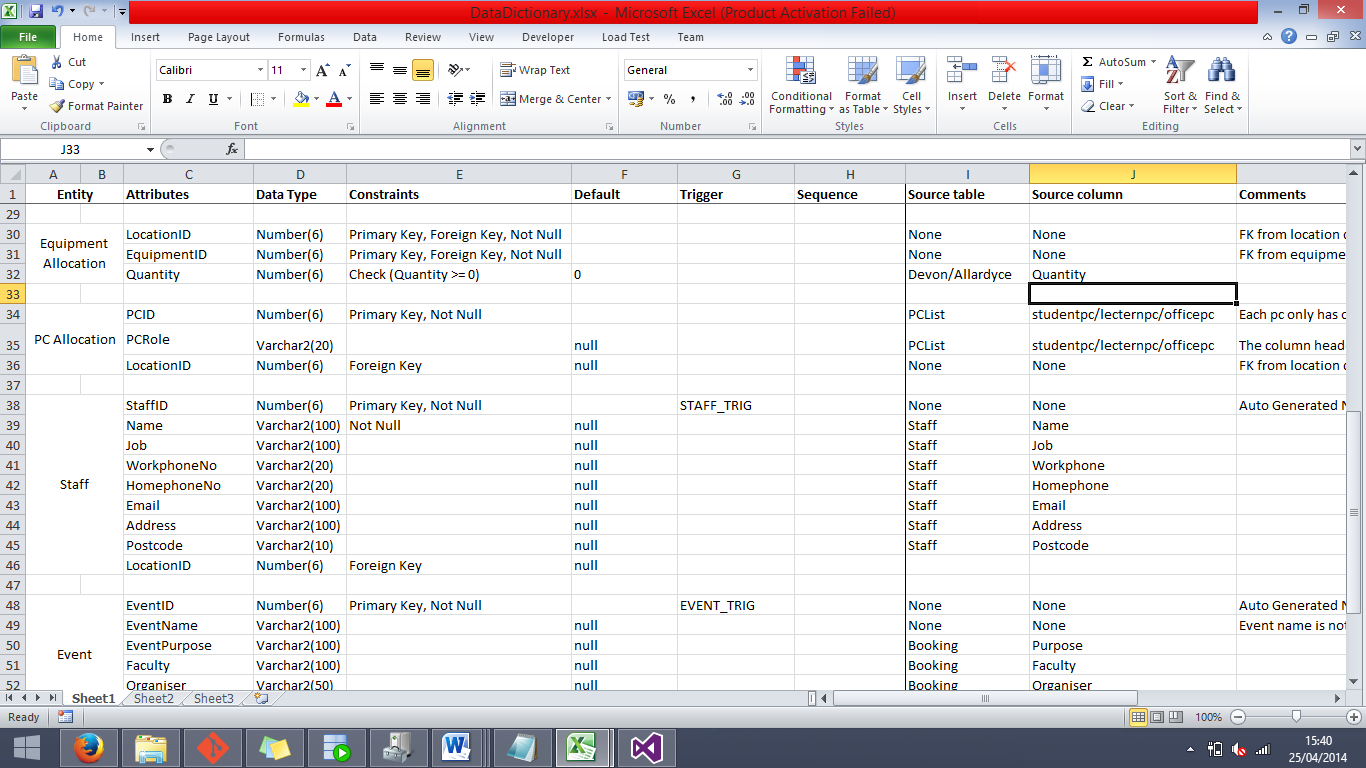
*Triggers*

EQUIPMENT\_TRIG – The purpose of this trigger is to act as an activator for the EQUIPMENT\_SEQ sequence, set to trigger after a row of data is inserted into the Equipment entity.

*Other Constraints*

EquipmentID (Unique, Not Null) – Due to the fact that this field is a Primary Key, it is automatically counted as Unique and Not Null.

Name, Type, Make, Description, Category (Default Null) – As none of these pieces of information are completely necessary at the creation of each row of data, we made the default for each field null.

**EQUIPMENT ALLOCATION**

**Attributes**

LocationID (EQUIPMENTALLOCATION\_PK, EQUIPALLO\_LOCATION\_FK) – The purpose of this field is to work in conjunction with EquipmentID to provide a unique index for the Equipment Allocation entity to be used as a Primary Key, and to form a relationship with the Location entity to derive locations for each piece of equipment.

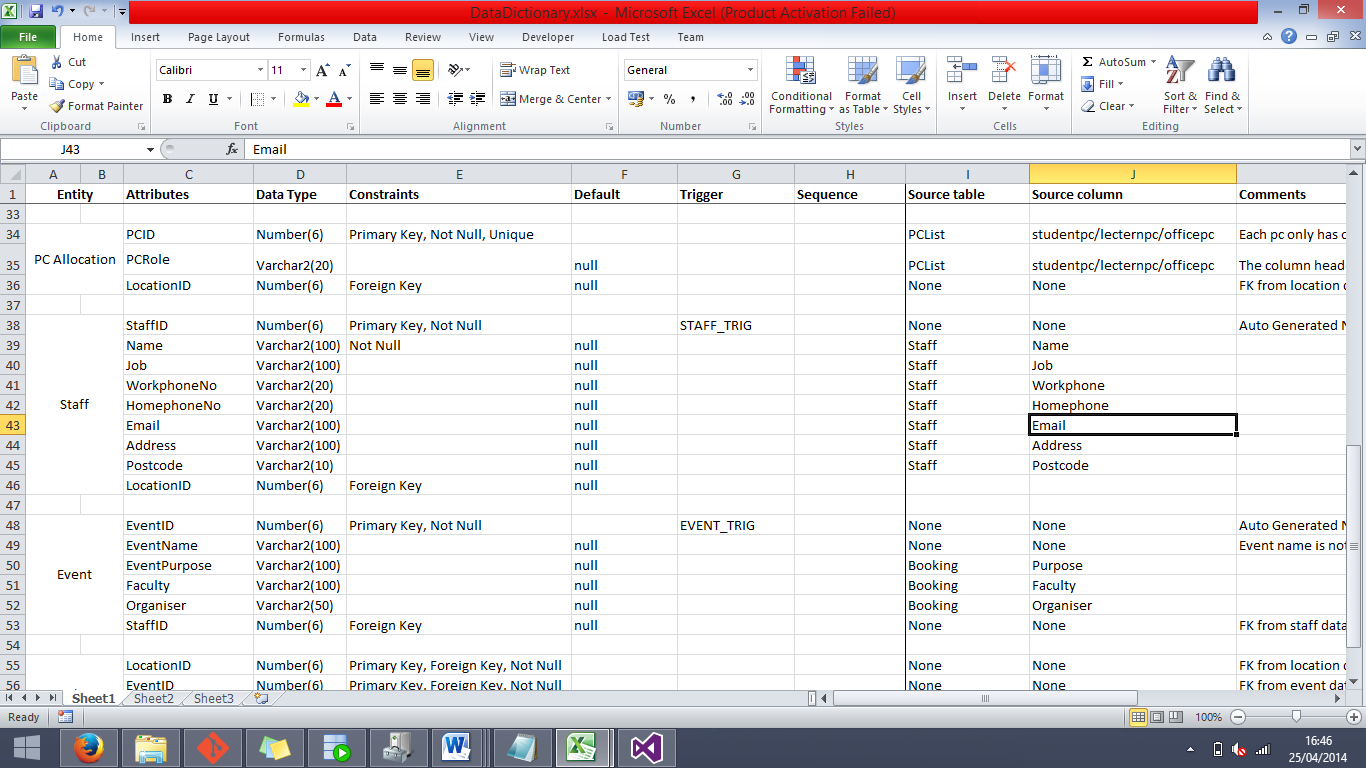
EquipmentID (EQUIPMENTALLOCATION\_PK, EQUIPALLO\_EQUIPMENT\_FK) – The purpose of this field is to work in conjunction with LocationID to provide a unique index for the Equipment Allocation entity to be used as a Primary Key, and to form a relationship with the Equipment entity to derive pieces of equipment for each location.

Quantity – The purpose of this column is to represent the quantity of the item in every Location/Equipment pair.

***Constraints***

LocationID, EquipmentID (Not Null) – At the pair of these fields are what forms the Primary Key for this entity; it would not do very well to have either one of them as null at any time. Note that these fields are not required to be unique as most other Primary Keys are in other entities; this is due to the fact that there will be many pieces of the same equipment used in multiple locations and many locations will have multiple pieces of equipment.

Quantity (CHECK, Default 0) – This check is in place to make sure that the quantity entered cannot be below 0, as this would not make sense. The option is also there to leave the field blank for it to be set as 0, and fill it in later.

**PC ALLOCATION**

**Attributes**

PCID (PCALLOCATION\_PK) – The purpose of this field is to provide a unique index and to serve as the Primary Key for the PC Allocation entity.

PCRole – The purpose of this field is to specify the role of each PC that is listed within the PC Allocation table.

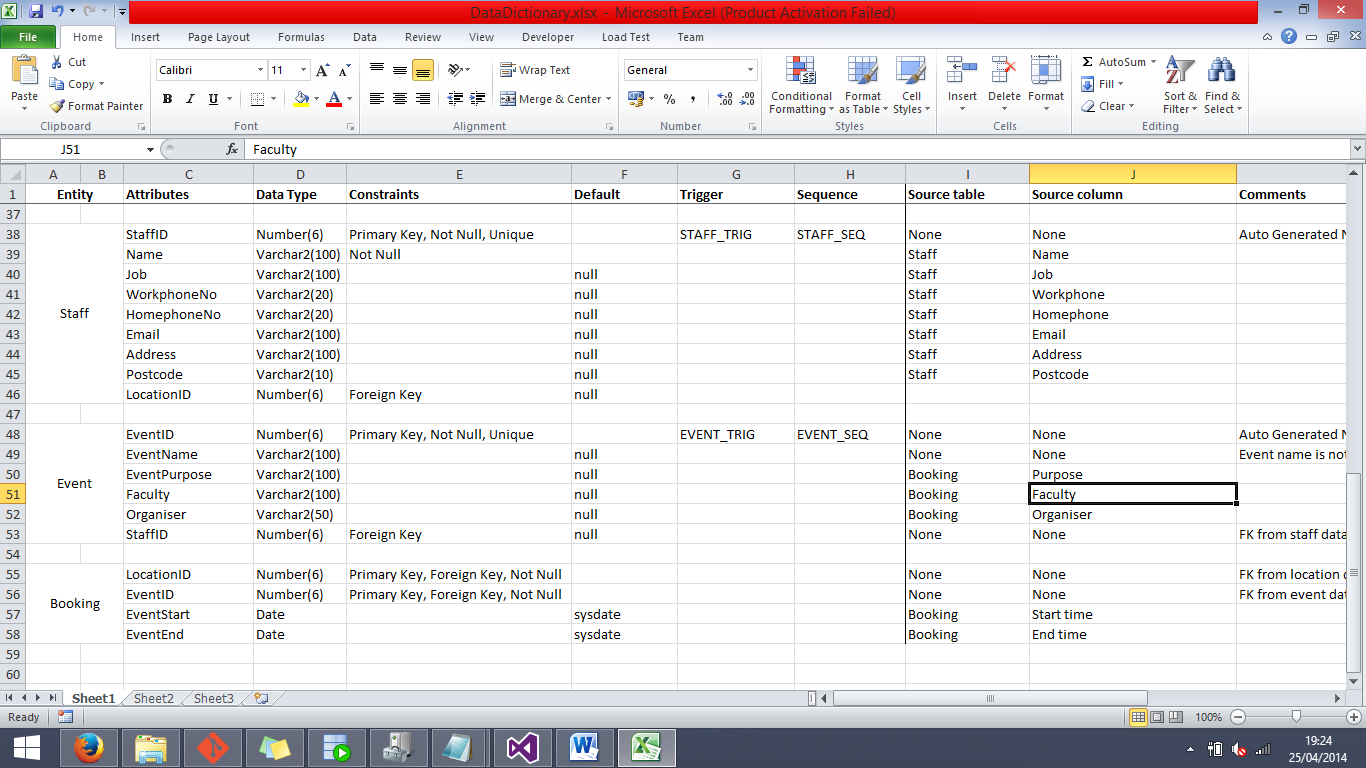
LocationID (PCALLO\_LOCATION\_FK) – The purpose of this field is to provide the location of each PC in the entity, and form a relationship between the PC Allocation and Location tables.

**Constraints**

PCID (Unique, Not Null) – Due to the fact that this field is the Primary Key for the PC Allocation entity, it is automatically set to Unique and Not Null.

PCRole, LocationID (Default Null) – As you don’t need both of these pieces of information to justify a row of data in this entity, these fields have a default of null so that you can come back later to fill it in.

**STAFF**



**Attributes**

StaffID (STAFF\_PK) – The purpose of this column is to provide a unique index which is populated via a Sequence and Trigger, and to act as a Primary Key for the Staff entity.

Name – The purpose of this column is to store the names of each member of staff in this entity.

Job – The purpose of this column is to store the position that each member of staff hold at the university.

WorkphoneNo – The purpose of this column is to store the workplace phone numbers of each member of staff.

Homephoneno – The purpose of this column is to store the home phone numbers of each member of staff.

Email – The purpose of this column is to store the email addresses of each member of staff.

Address – The purpose of this column is to store the home address of each member of staff.

Postcode – The purpose of this column is to store the home postcode of each member of staff.

LocationID (STAFF\_LOCATION\_FK) – The purpose of this column is to provide a reference to the room that the member of staff is currently using as an office and to form a relationship between the Staff entity and the Location entity.

***Constraints***

*Sequences*

STAFF\_SEQ – The purpose of this sequence is to populate the StaffID column whenever a new entry is inserted into the Staff entity.

*Triggers*

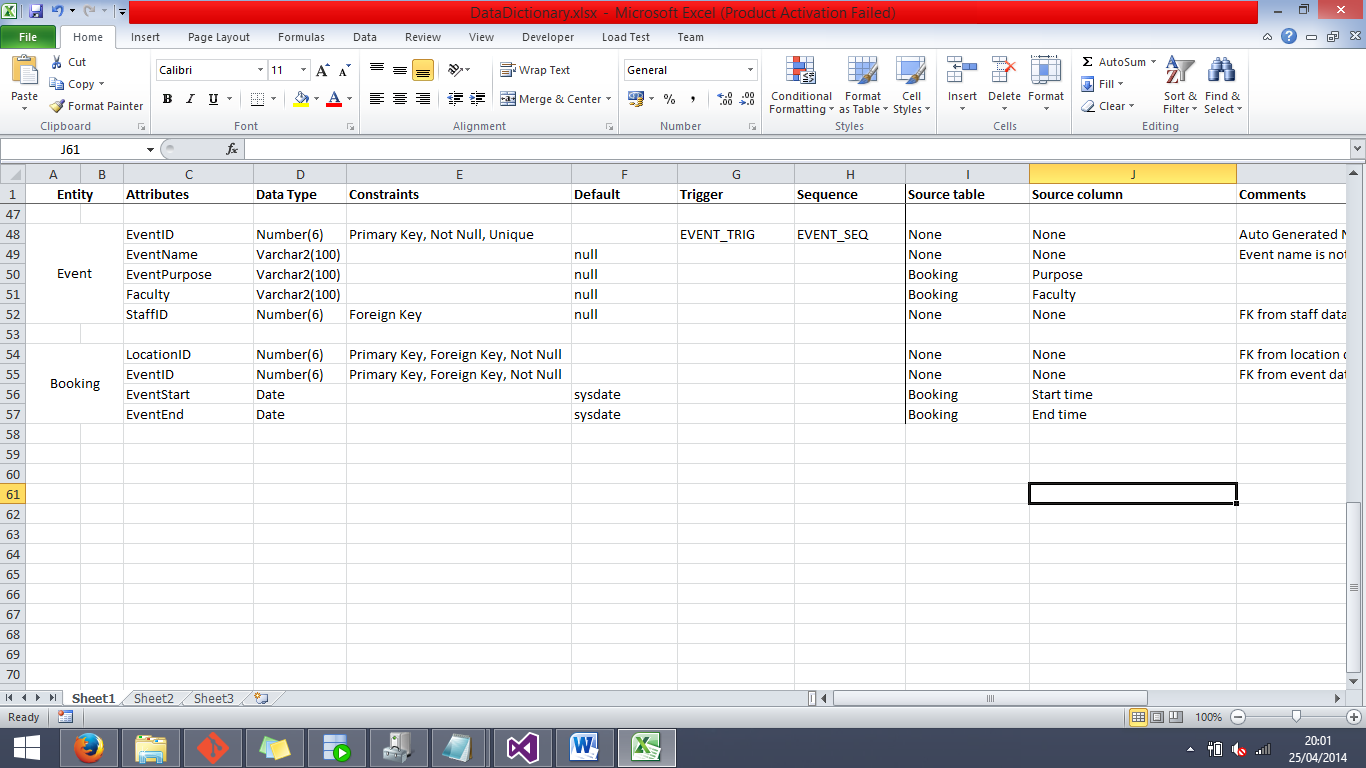
STAFF\_TRIG – The purpose of this trigger is to act as an activator for the STAFF\_SEQ sequence, set to trigger after a row of data is inserted into the Staff entity.

*Other Constraints*

StaffID (Unique, Not Null) – Due to the fact that this field is the Primary Key for the Staff table, it is automatically contributed the Unique and Not Null constraints.

Name (Not Null) – It would not make sense to create a new Staff entry without supplying at least a name, so we made the field with a Not Null constraint.

Job, WorkphoneNo, HomephoneNo, Email, Address, Postcode, LocationID (Default null) – None of this information is completely compulsory for an entry of a Staff immediately, so we gave them all a default value of null.

**EVENT**

**Attributes**

EventID (EVENT\_PK) – The purpose of this column is to provide a unique index for the Event entity by acting as a Primary and being populated by a Sequence.

EventName – The purpose of this column is to store information about the name of the event.

EventPurpose – The purpose of this column is to hold a short description of the event purpose.

Faculty – The purpose of this column is to specify which faculty of the university will be hosting the associated event.

StaffID (EVENT\_STAFF\_FK) – The purpose of this column is to provide a point of reference to see which member of staff is organising the event, and also to form a relationship between the Event and Staff entities.

***Constraints***

*Sequences*

EVENT\_SEQ – The purpose of this sequence is to populate the EventID column whenever a new entry is inserted into the Event entity.

*Triggers*

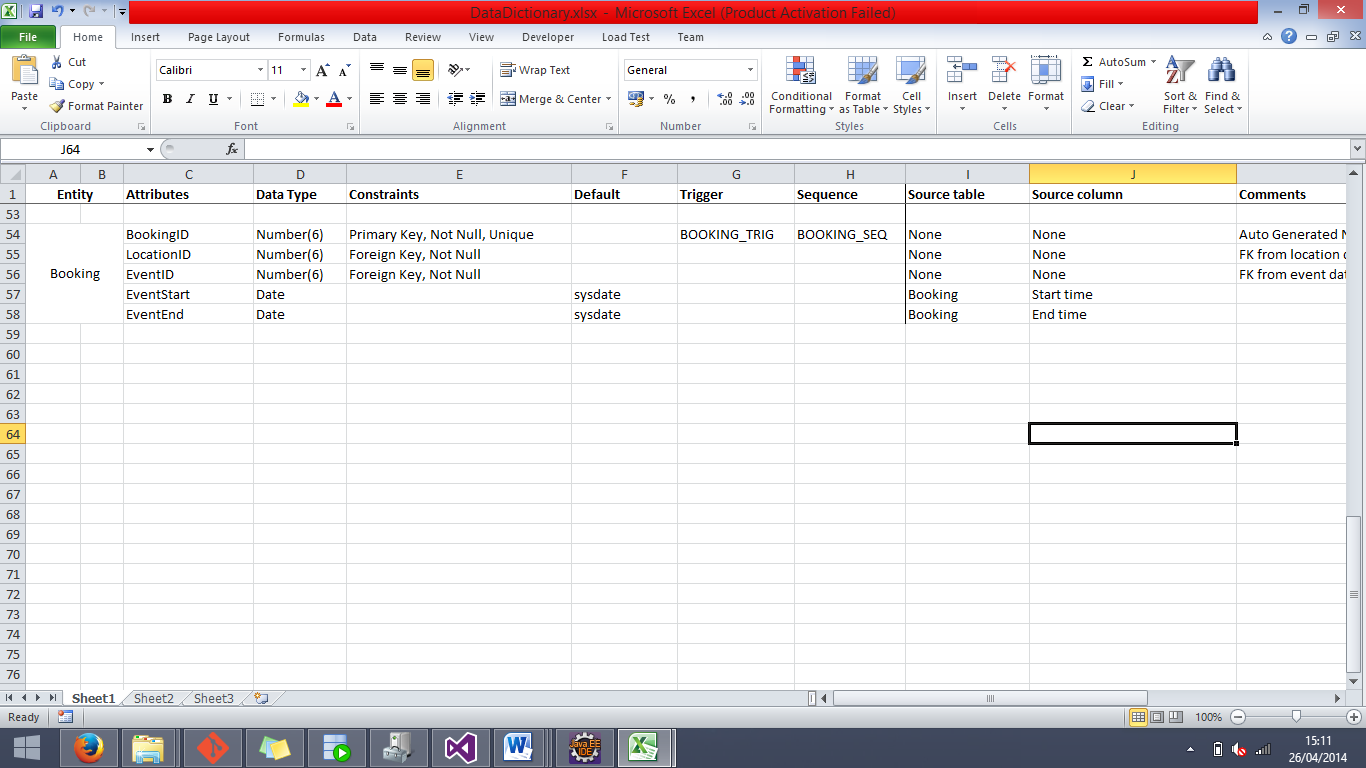
EVENT\_TRIG – The purpose of this trigger is to act as an activator for the EVENT\_SEQ sequence, set to trigger after a row of data is inserted into the Event entity.

*Other Constraints*

EventID (Unique, Not Null) – Due to the fact that this column acts as a Primary Key, it automatically gains the Unique and Not Null constraints.

EventName, EventPurpose, Faculty, StaffID (Default Null) – As not all of these columns are necessary to create a sensible entry to this entity, they were all given a default value of null so that they can be filled in later.

**BOOKING**

**Attributes**

BookingID (BOOKING\_PK) – The purpose of this column is to provide a unique index for the BOOKING entity to use as a Primary Key.

LocationID (BOOKING\_LOCATION\_FK) – The purpose of this column is to provide a reference with which to get the location that the corresponding booking is registered at, and to form a relationship between the BOOKING and LOCATION entities.

EventID (BOOKING\_EVENT\_FK) – The purpose of this column is to provide a reference with which to get the event that the corresponding booking is hosting, and to form a relationship between the BOOKING and EVENT entities.

EventStart – The purpose of this column is to store the date and time at which the event starts.

EventEnd – The purpose of this column is to store the date and time at which the event ends.

***Constraints***

*Sequences*

BOOKING\_SEQ – The purpose of this sequence is to populate the BookingID column whenever a new entry is inserted into the Booking entity.

*Triggers*

BOOKING\_TRIG – The purpose of this trigger is to act as an activator for the BOOKING\_SEQ sequence, set to trigger after a row of data is inserted into the Booking entity.

*Other Constraints*

BookingID (Not Null, Unique) – This column serves as the Primary Key for the BOOKING entity, and as such it is automatically attributed the Not Null and Unique constraints.

LocationID, EventID (Not Null) – These columns were given the Not Null constraint due to the fact that it would not make sense to insert a new booking into the system without supplying both an event and a location.

EventStart, EventEnd (Default sysdate) – As a matter of just keeping dates in the date fields, it was decided to default these two fields to have the current system date and time if they are not specified at creation, and can just be changed later.

**Differences between Final ERD and the final Database implementation**

**Configuration tables:**

The final ERD ‘Configuration’ consists of two separate tables ‘Location-Configuration’ and ‘Configuration types’ both created independently of the source data.

In this implementation the ‘Location-Configuration’ table has a one-to-many relationship with location and contains the capacity and the configurationID for that location. The ConfigurationID is then stored in a separate ‘ConfigurationType’ table with a one-to-many relationship with ‘Location-Configuration’.

Though these relationships work in the ERD, when creating the database having two tables caused difficulties and unnecessary complications when importing the data.

This was changed in the database implementation to instead use one Configuration table containing the 3 different capacities for a given location, having a one-to-one relationship with Location.

**Special equipment:**

When creating the equipment tables in the ERD we felt we needed a ‘Special equipment’ table to incorporate equipment such as PC’s. This table was made separate from the Equipment table as the relationship between PC’s and location differed; the special equipment didn’t require having a linker table such as ‘Equipment allocation’.

When implementing the database this table was changed to ‘PCAllocation’ as PC’s were the only special equipment that contained different attributes. This table consisted of the PC’s role, its id and the location which is it found.

***Data Loading Strategy and Problem Resolution***

Data Loading Strategy

The general strategy that we used to load the source data into the database was to first target the tables that did not rely on any foreign keys and establish them. This made it easier when creating the tables which rely on primary keys from other tables as foreign keys.

We did not put any constraints into the database until all of the data was loaded and formatted correctly into the correct data types. The primary reason for this was because a fair portion of the data had to be further validated and formatted after it was imported into the database. An example of this would be to take the date, start time, and end time of each event. All of these entries are meant to be dates however they have to be imported into the database first as normal strings then later formatted into a date data type within the database.

One of the biggest problems that we faced when devising our strategy for loading the source data was using the primary keys of tables as foreign keys. Finding a way to effectively populate those tables with the correct foreign data proved to be a fairly large challenge. The way that we overcame the task was to use an update query on the foreign key column, which matches up the correct key from the foreign table with the appropriate entry in the current table. The most notable example of this in action would be with the LOCATIONID from the LOCATION table, due to how often it is used as a foreign key. The common steps that we took when utilising this column as a foreign key were:

1. Load all data relevant to the table that we are currently loading, leaving the entire LOCATIONID foreign key column ***null*** for the time being.
2. Add a number of extra ***temporary columns*** to the current table which could be used in the update query to avoid ambiguity when matching rows (for the LOCATIONID foreign key, these temporary columns were usually Building, RoomNo, and RoomName).
3. Execute an update query on the foreign key column, adding the corresponding values from the primary key table.
4. Run queries and manually check that the values entered as foreign keys match correctly to the data that is contained in the source excel document.
5. Remove the temporary columns from the table, and add constraints and primary keys.

When creating the tables, one of the things that we made sure to do on creation was to make the triggers and sequences for the appropriate surrogate keys when needed.

The first setback that we encountered was that we realised that we first had to organise the source. There was no conceivable way that we would have been able to import the data as it was in the initial document so we decided to split it up into a series of documents that correspond to the different entities that were going to be included in our database.

The strategy that we used to find out what data we needed for each of the tables was to first analyse the data and then look back at our final ERD for the system and see how compatible it was with the data we were presented with. What we found was that some aspects of the ERD not sync very well with the source data. To combat this we had to go through our tables and make certain changes ranging from adding, removing or renaming attributes of a given table, to removing a table entirely.

An example of us invoking this strategy would be with the CONFIGURATION table currently in the database; it differs from that of the table in the ERD. We found that the data that is associated with the configuration of each room is not compatible with the table layout of that in the ERD.

-----

Issues were encountered when loading the data specifically the way in which the unique identifiers for the PCs were generated inside excel document. The fact that the numbers were randomly generated within a range meant that there was a possibility that not all of the IDs were unique. This meant that we had to validate the data before importing it into the database. To do this our primary method was to employ a built-in function inside of excel that checks for duplicates within a selected amount of cells (Function: Duplicate Values, Path: Home > Conditional Formatting > Highlight Cells Rules > Duplicate Values).

-----

One of the most common but less threatening issues that we ran into during the data loading stage was several different inconsistencies in the excel source document.

***Problem***: Misspelt data in the database, causing entries to remain null when attempted to update the foreign key column.

**Solution**: The amount of cases we had of this was relatively few, so our solution to this issue was to just manually correct the spelling errors.

***Problem:***There were cases from time to time where there were entries in the data that were inconsistent in the rest of the data. A prime example of this would be that there were entries in the equipment allocation data containing times that were not even listed as times in the equipment data.

**Solution**: In most cases of this situation we thought it was most prudent to just delete the line which had the non-existent time, as it didn’t make sense for the database to claim that an item was there that wasn’t even recorded.

***Problem*:** In several different fields of the excel source document, the actual data types of the values stored were incorrect for the context that they were in. For example, the numbers stored for the capacity values for each different configuration for each location were stored as text in the spreadsheet, which translated into strings for the database, which caused issues when trying to import them into a column that is of the number data type.

**Solution**: In excel you are able to change the type of data that is stored inside the cells, so it was a fairly simple matter to select all of the incorrectly typed cells and change them to the correct format.

***Problem***: In practically every section of the excel source document, there were empty rows scattered throughout the data, which Oracle reads in as an empty line, creating many unnecessary rows inside of the database when imported.

**Solution**: There were a couple of different ways that we tackled this problem, finding out the more efficient way later on. The first solution that we came up with was to go through the excel document and manually delete the blank rows for each section of data, which proved to be a fairly time consuming and arduous task. The second, and much more time efficient solution that we came up with, was to import all of the empty rows with the data, and use a delete script on the table to remove all of the rows that have any given column as null, as that means the rest of the entry will be null too.

**Detailed Description**

For this section of the Data Loading explanation, there will be more in depth explanations as to how we achieved certain specific aspects of the data loading.

The first exercise to be discussed is how we managed to successfully merge the time and date columns from the source data into a single field inside the database, which is of data type “Date”.

First of all, the reasoning that we applied when making this decision was that, due to the fact that Oracle databases do not have a field data type that is specifically for time, it would make more sense to combine the time with the date as the time would have to be stored in a field with the date type “Date” anyway. The only major change that this would make in terms of exporting the database or importing new data into the database is the formatting that would have to be applied to the data when extracting it from the database.

There was an issue that was found with formatting the time however, when extracting the database; it does not take the time into account the time when making the insert statements. The only way that we found to counter this issue was to later go and manually change the insert scripts to include the date data and change the formatting to take the time into account. This kind of fix is viable in a case such as this with such a small amount of data, but in a much larger situation, an alternate method would need to be used.

The first action that we had to perform was to make two temporary fields inside of the BOOKING entity that would hold the separated time and date of each booking. Both of these fields we of data type “Varchar2” as it will be necessary to treat them as strings at a later point.

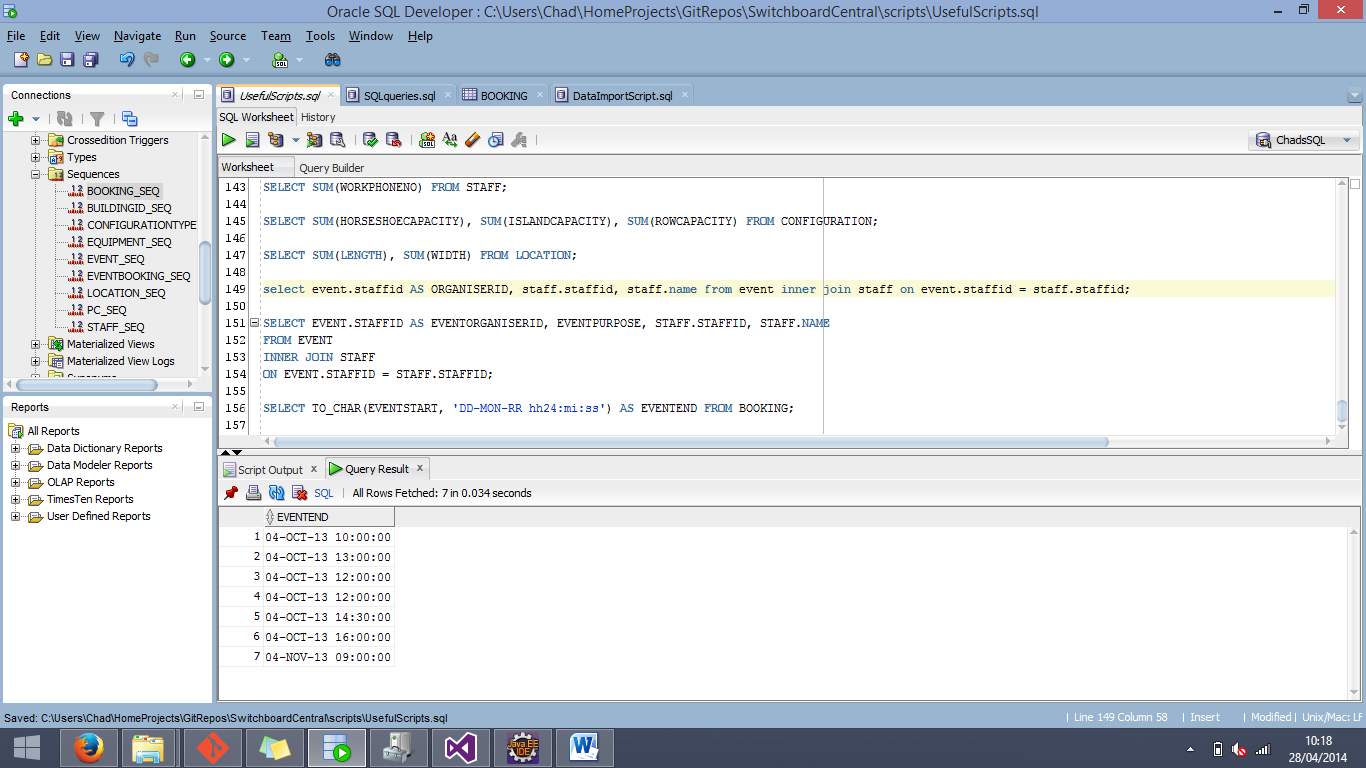
The next step was to then create the field that would hold the final data in the date format. This column had to be made with the Default Null constraint and then be filled in later with an update statement.

Before attempting the update, we had to test that we would successfully be able to update the Date column with something that is able to be formatted into a date. For this we used a specialised select statement that concatenated the time and date when extracting it from the database, which is shown below:

SQL Script:

SELECT TO\_CHAR(TO\_DATE(TO\_CHAR(BOOKINGDATE || ' ' || STARTTIME), 'dd-mm-yy hh24:mi:ss'), 'yyyy-mm-dd hh24:mi:ss') as NEWDATE FROM BOOKING;

The results of the above query are shown below, indicating that it is possible update the permanent column in this fashion.



The SQL Script used for the update is shown below:

SQL Script:

UPDATE EVENTEND BookingNew

SET ENDDATE = (SELECT TO\_DATE(TO\_CHAR(BOOKINGDATE || ' ' || ENDTIME), 'DD-MON-RR hh24:mi:ss')

FROM BOOKING BookingOld

WHERE BookingOld.BOOKINGID = BookingNew.BOOKINGID);

UPDATE EVENTSTART BookingNew

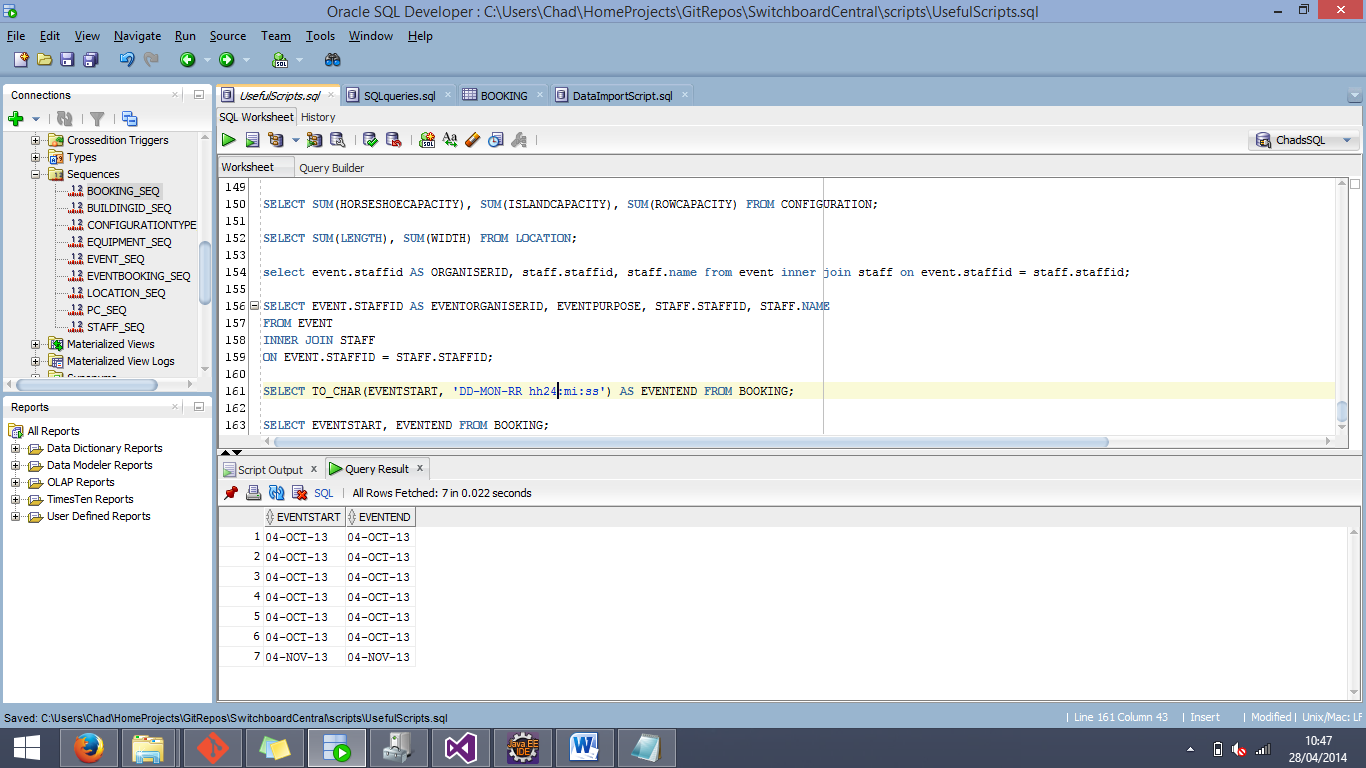
SET ENDDATE = (SELECT TO\_DATE(TO\_CHAR(BOOKINGDATE || ' ' || STARTTIME), 'DD-MON-RR hh24:mi:ss')

FROM BOOKING BookingOld

WHERE BookingOld.BOOKINGID = BookingNew.BOOKINGID);

Using these statements, we took the date and time that was imported from the source document, and concatenate them by converting them into a single string with a space in-between the two values. Then, using database aliases, we updated the permanent fields with the data relevant to that row.

Once the columns were updated, the data in the columns were showing only the date.

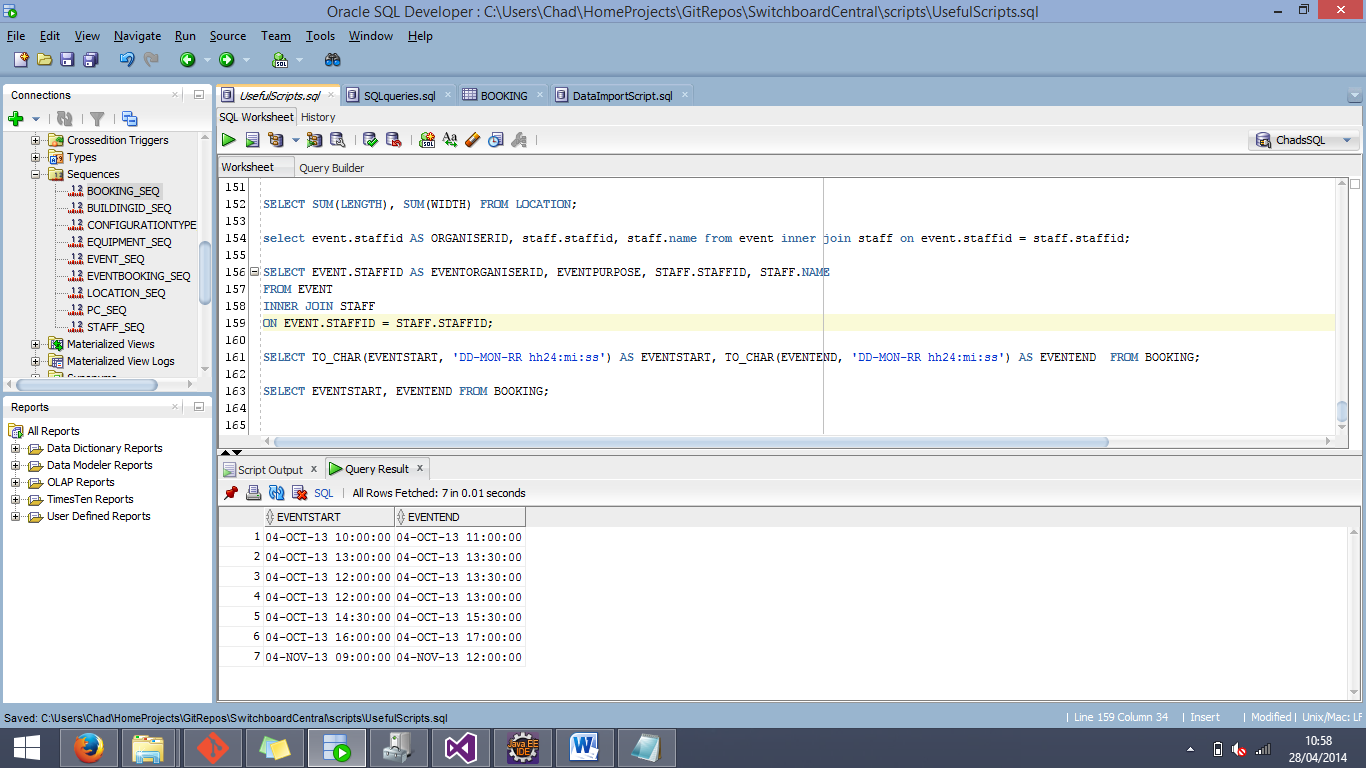


To counter this issue, we had to once again employ the TO\_CHAR() function in SQL to format the data as it was coming out the database.

SQL Script:

SELECT TO\_CHAR(EVENTSTART, 'DD-MON-RR hh24:mi:ss') AS EVENTSTART, TO\_CHAR(EVENTEND, 'DD-MON-RR hh24:mi:ss') AS EVENTEND FROM BOOKING;

After using that script to extract the date and time from the database, it was confirmed that we would be able to successfully convert the date from the database to include the time and store them both in a single field.



------

The next section of detail data loading that will be shown is how we successfully managed to import the PC data into the PCALLOCATION entity. This caused a large amount of initial confusion due to how the source data was structured in the Excel document. The fact that the numbers for the PC identification numbers changed every time you performed an action in the Excel document meant that there were very real possibilities that there would be duplicates. This of course would pose a major problem especially if they were going to be used as a unique Primary Key.

Apart from using the Excel function to check for duplicates, another method that we used was first of all enable the Primary Key constraint on the PCID column of the PCALLOCATION entity, and then to continue trying to import them until we come across a combination of numbers that does not have duplicates in them, a crude method to be sure, but effective.

First of all though, we had to figure a way to import the PC numbers while keeping them linked to their roles.

The method that we used to tackle this problem was to simply import one column of the PC ids at a time (due to the fact that each column of PC ids had a different role associated with it). Doing this, we were able to ensure that the numbers were unique (by checking to see if there were any violations of the Primary Key at any stage), and also make sure the roles for each PC are accurate by using the following SQL statements at each corresponding stage of the loading.

SQL Script:

UPDATE PCALLOCATION

SET PCROLE = ‘Lectern’

WHERE PCROLE IS null;

UPDATE PCALLOCATION

SET PCROLE = 'Office'

WHERE PCROLE IS null;

UPDATE PCALLOCATION

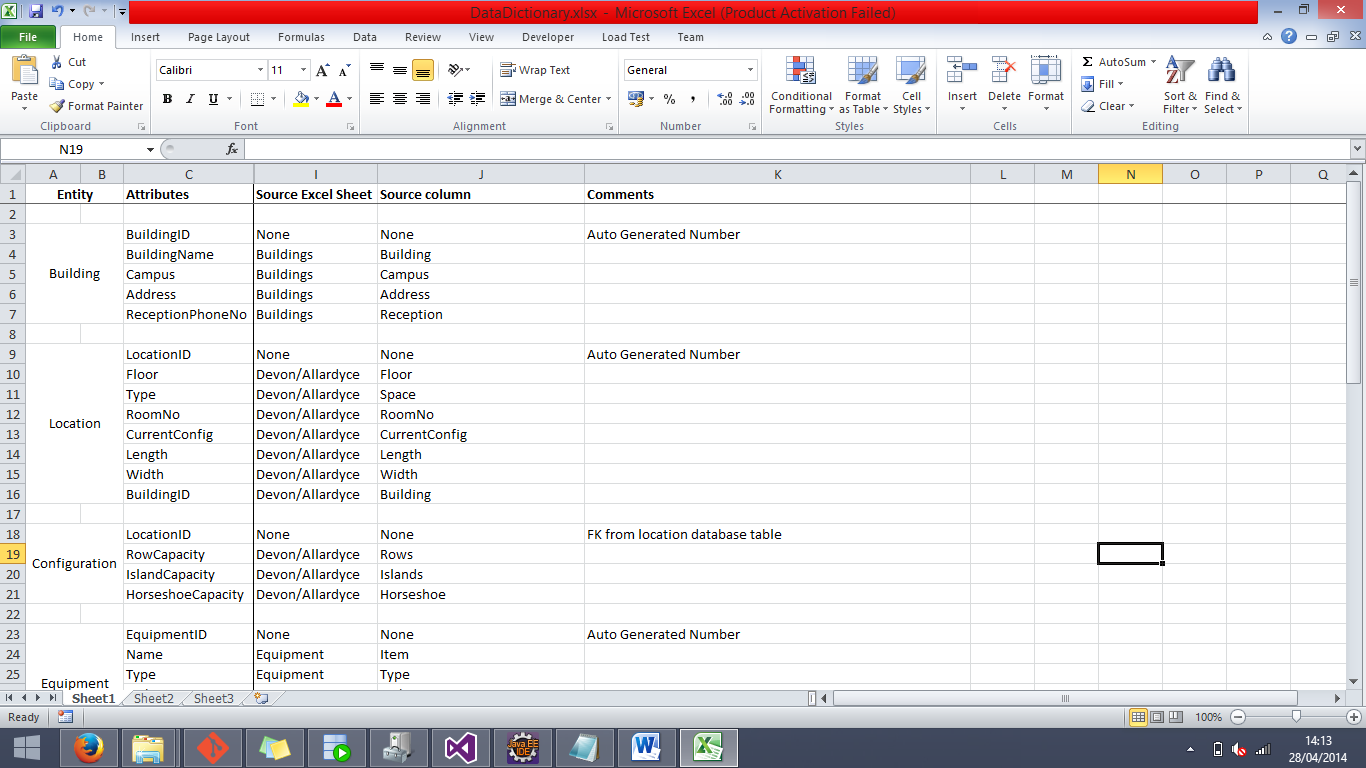
SET PCROLE = ‘Student’

WHERE PCROLE IS null;

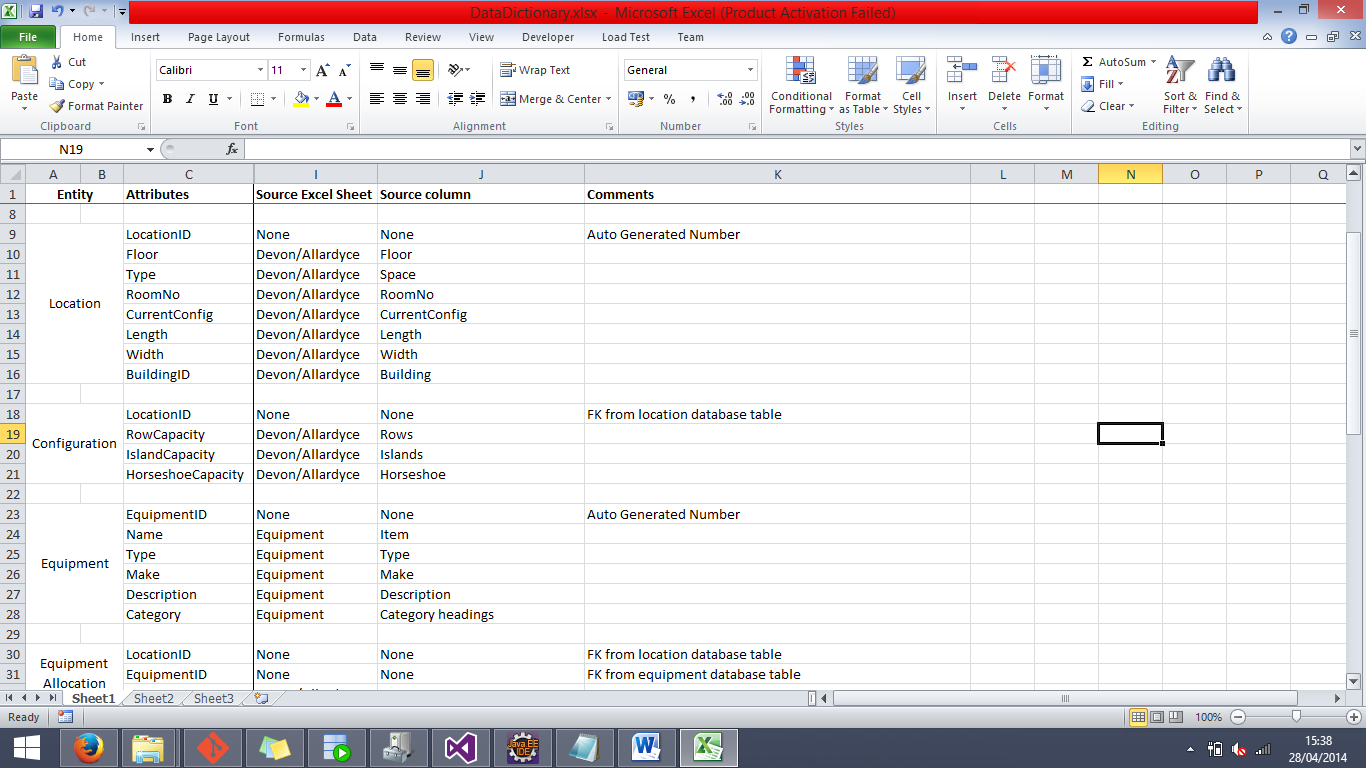
By using each of these SQL statements in turn, we were able to keep the roles of the PCs accurate with the corresponding PC.

**Data Mapping**

This section of the document will provide information about the relationship between the source data that we were given in the form of an Excel document, and the attributes of the final database in which the data was loaded. This document will specify which column of each sheet of the Excel document that was loaded into its database counterpart.

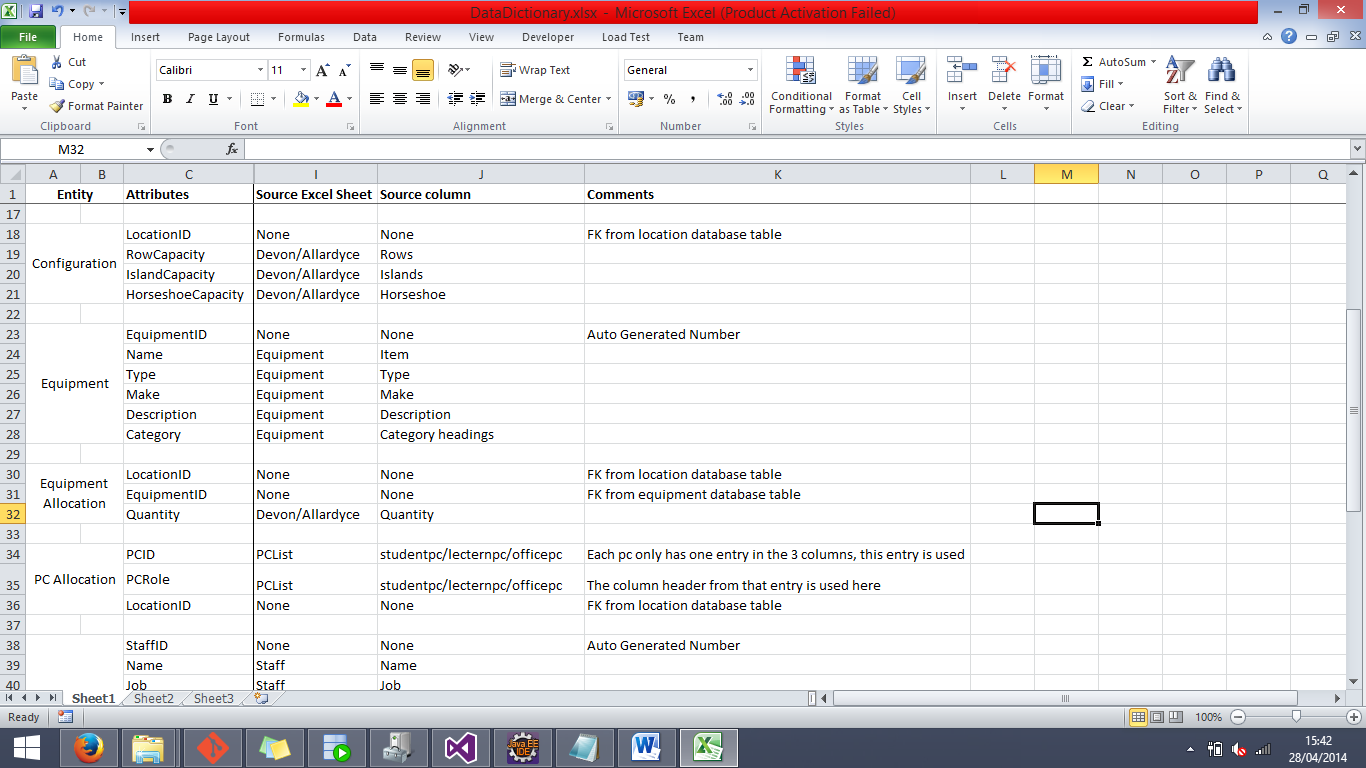
**BUILDING**

The BUILDING entity is one of the simplest in the bunch; it was very clear cut as to which source column would be going into be going into which database column, as such, no clarification is needed on the above image.

**LOCATION**

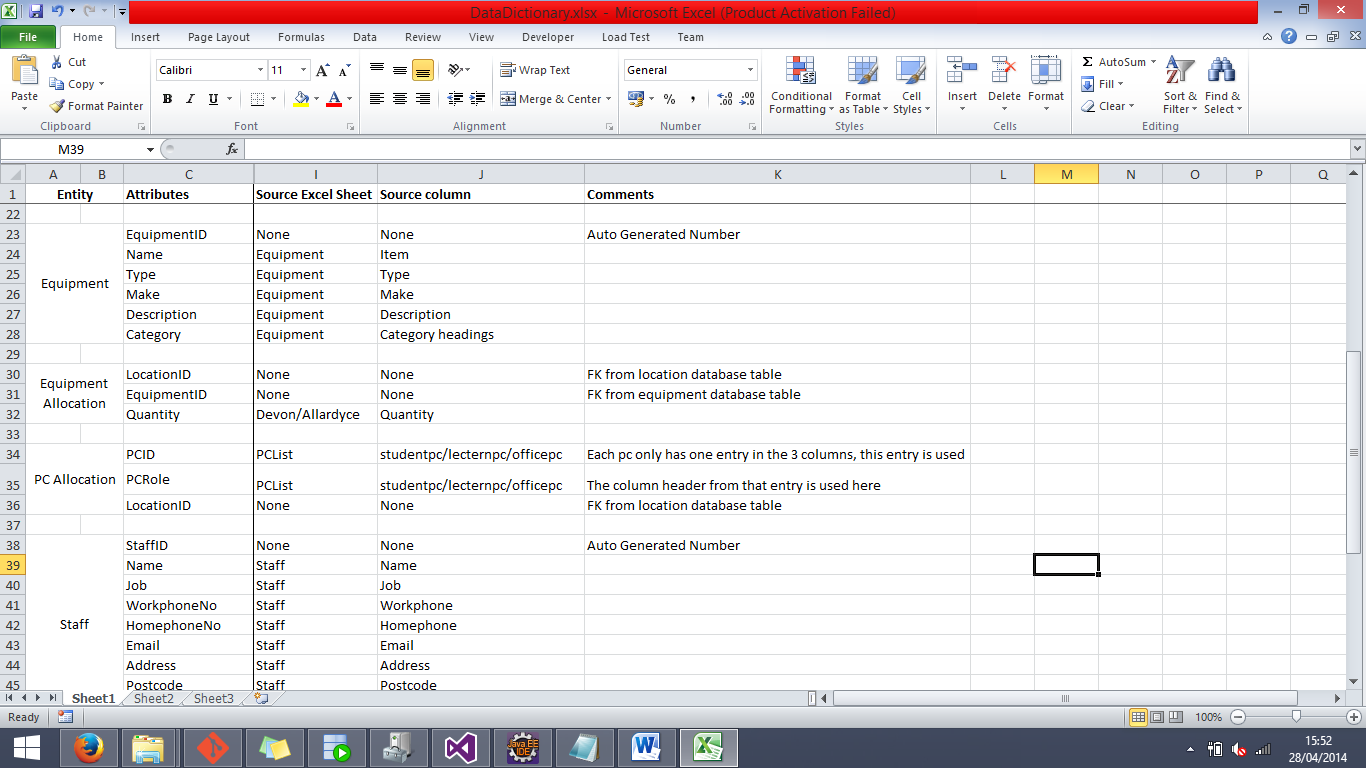
The source data for the LOCATION entity came from multiple sheets (Devon & Allardyce) of the source document, as noted in the image above. The BuildingID attribute technically did have the source column of Building from the Buildings sheet, but this column was technically populated as a foreign key from the BUILDING entity.

**CONFIGURATION**

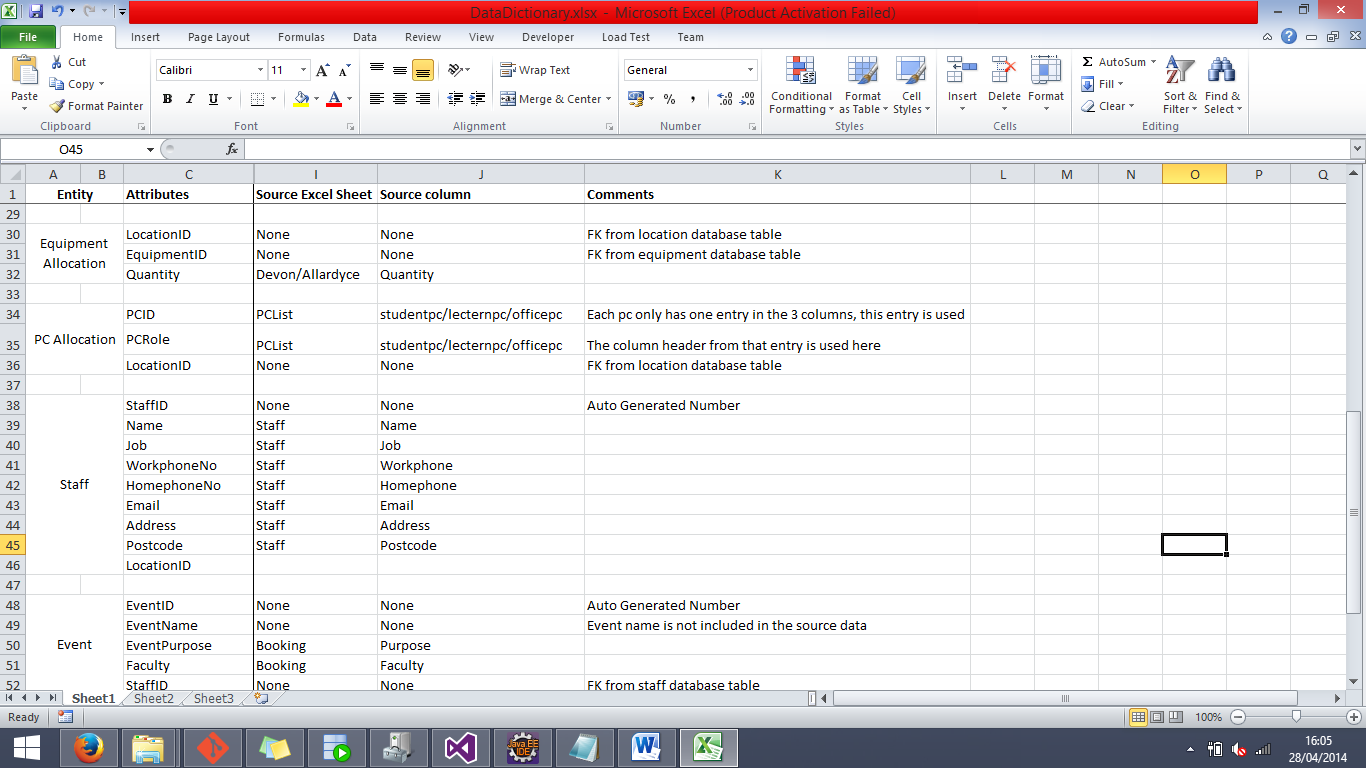


The LocationID from LOCATION serves as both the Primary Key of the table and a Foreign Key. It therefore does not have a source column from the Excel file.

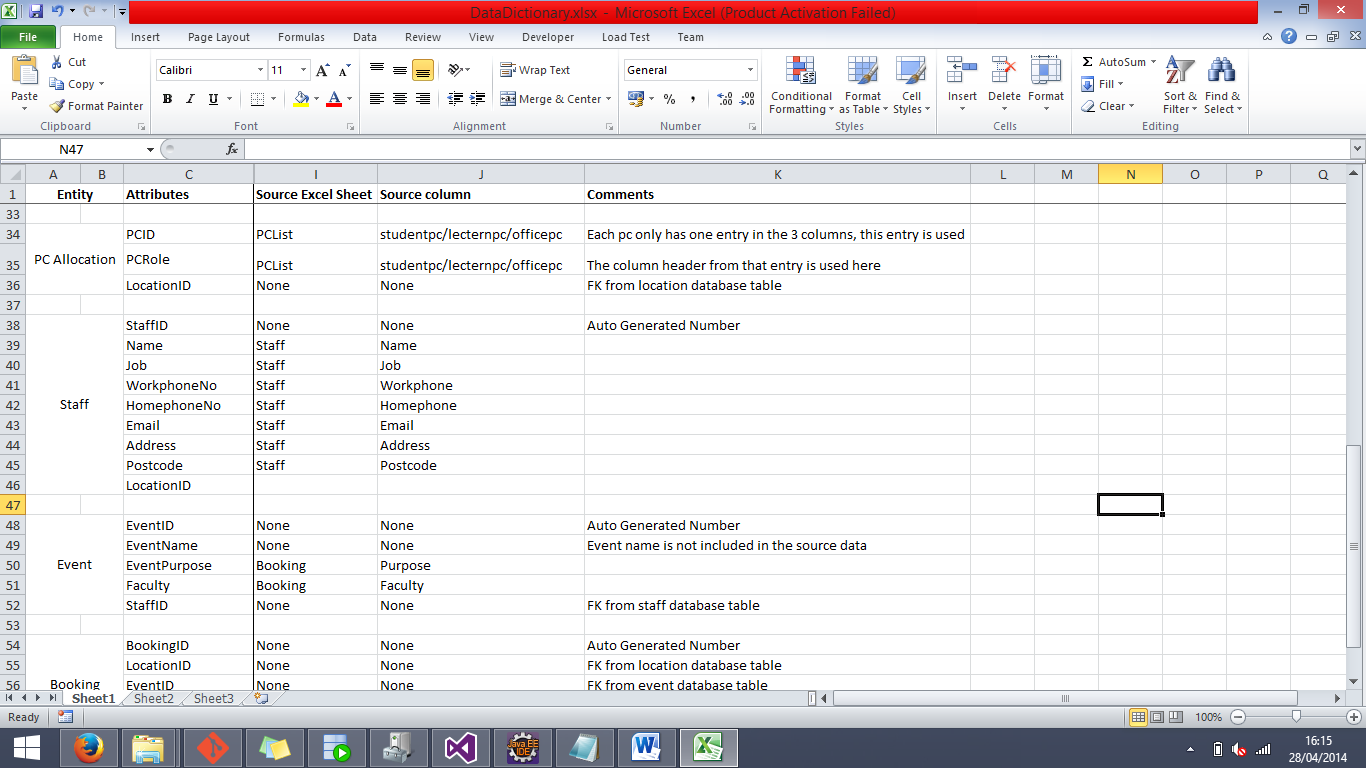
**EQUIPMENT**

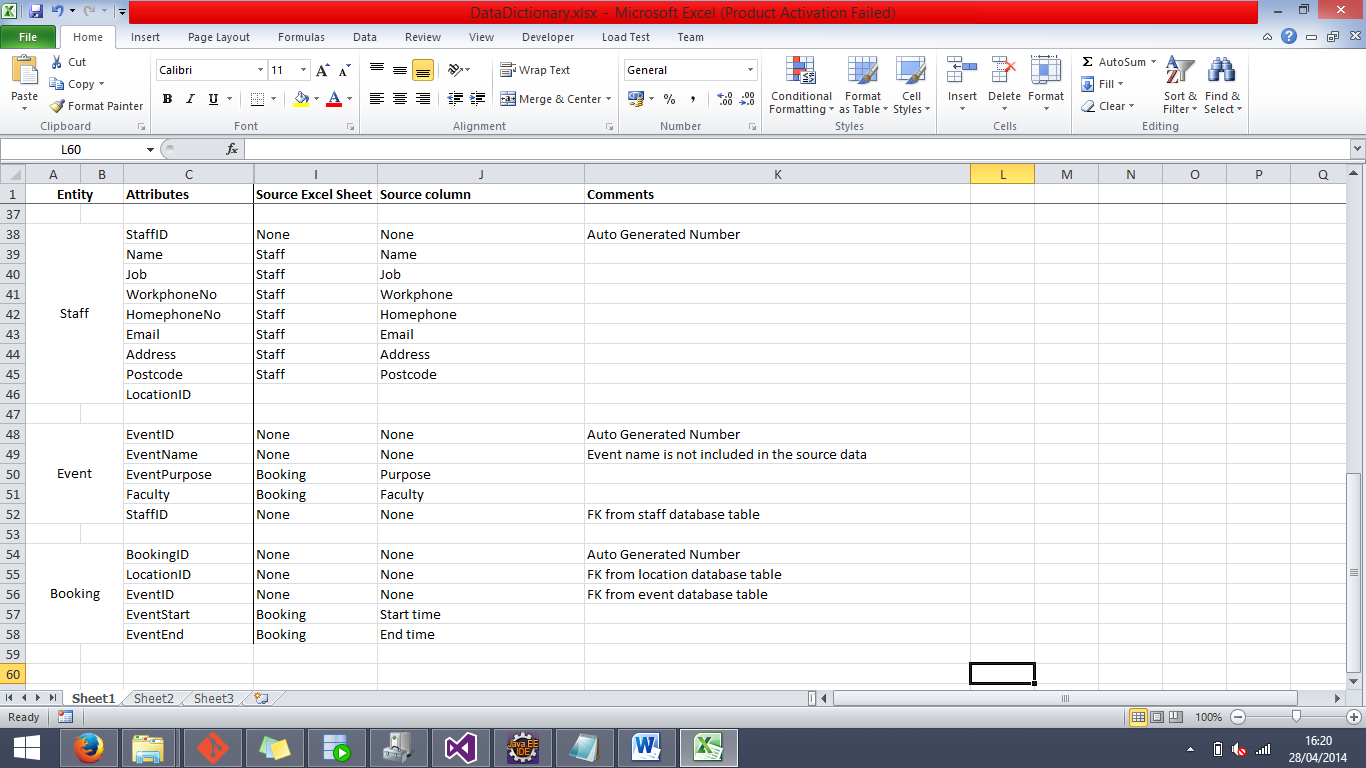
The only unique circumstance that this in the EQUIPMENT table is that the Category attribute did not draw its data from a column per se. As described in the data loading explanation, they were simple headers at the top of each section, and had to be copied down in the Excel source file to match up the data.

**EQUIPMENT ALLOCATION**

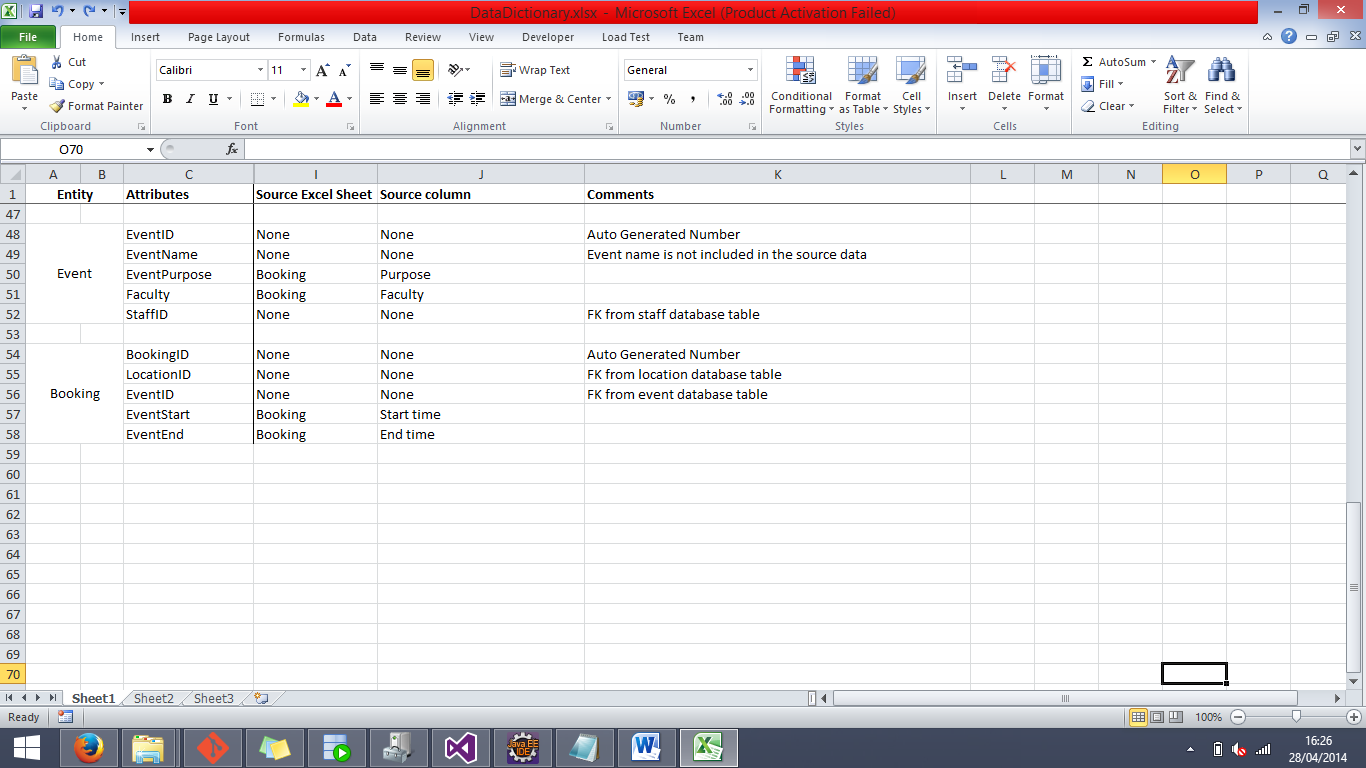
This entity features two foreign keys from other tables, making it only one attribute from this table that has a source column directly loaded from the source Excel document.

**PC ALLOCATION**

As mentioned in the comments above, the PCID attribute actually has three different source columns, but all equating it essentially the same data. Another oddity from this entity would be that the PCRole attribute did not have its own column of data in the Excel file, but rather was populated using update statements based on which numbers were being inserted into the database at that time.

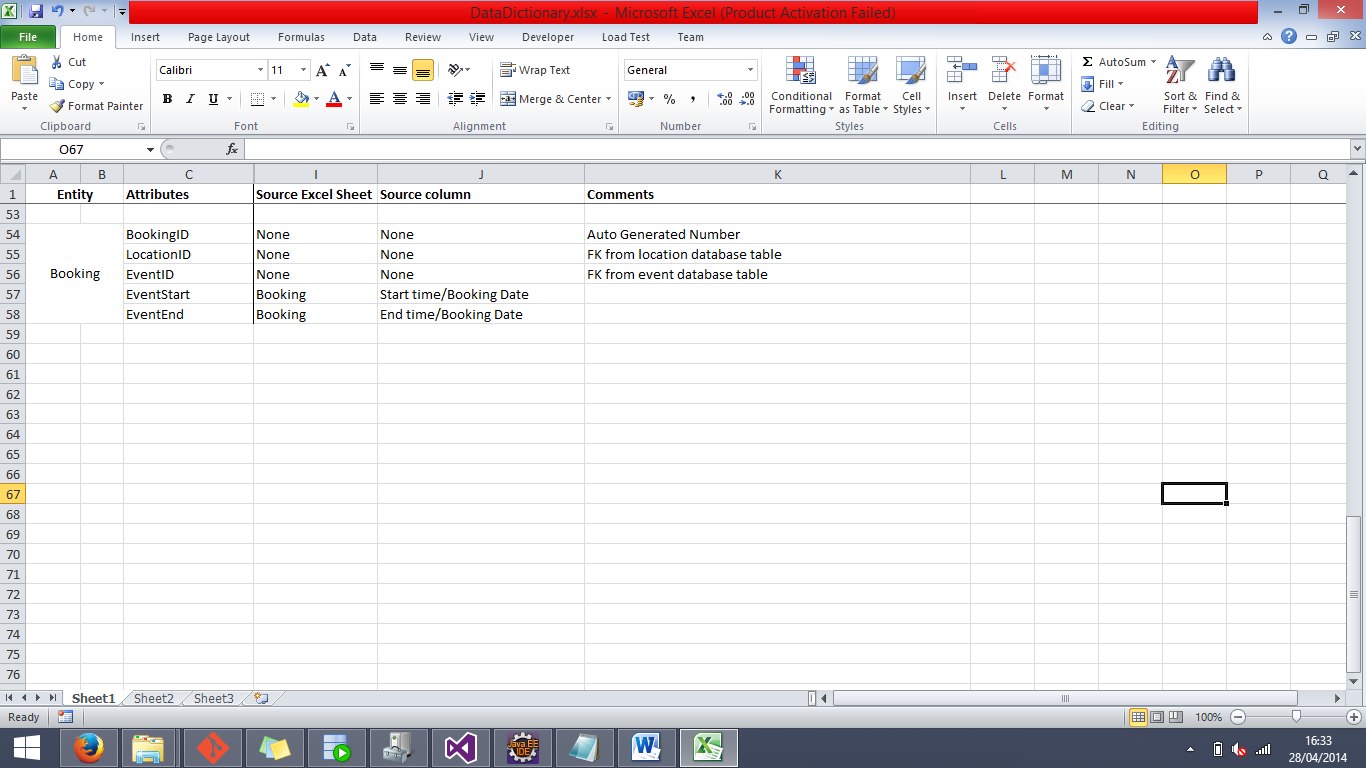
**STAFF**

Staff was a very simple table with very little complicated about it with the exception that it would need LocationID as a foreign key to derive the office or room that they work in.

**EVENT**

The only extraordinary thing about the EVENT entity was that, like mentioned just above, there is no EventName attribute listed in the data but it did not make sense to have Events without any names so it was decided to put that attribute in manually.

**BOOKING**

Again, as mentioned in the data loading explanation, the EventStart and EventEnd attributes use both the time and date to complete their data, and such they have two source data columns.

**Data Take-on Testing**

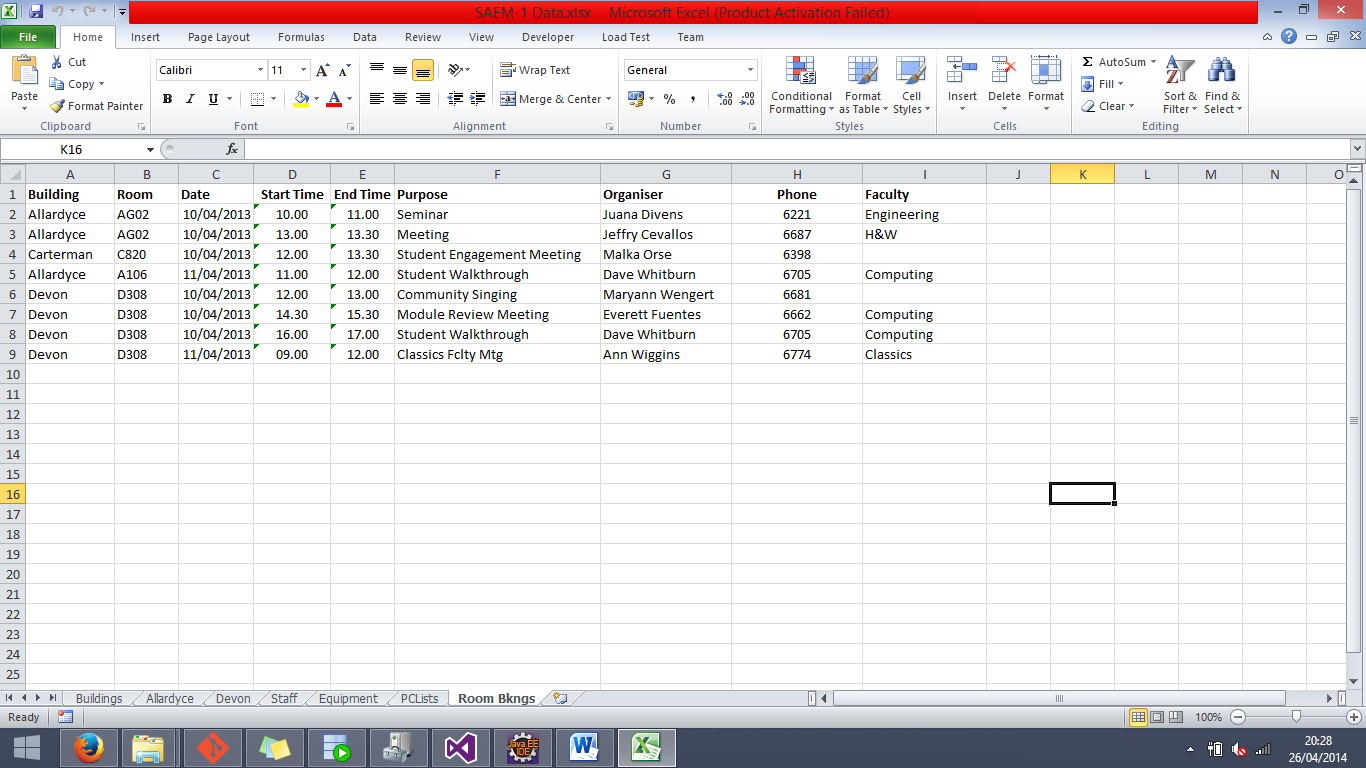
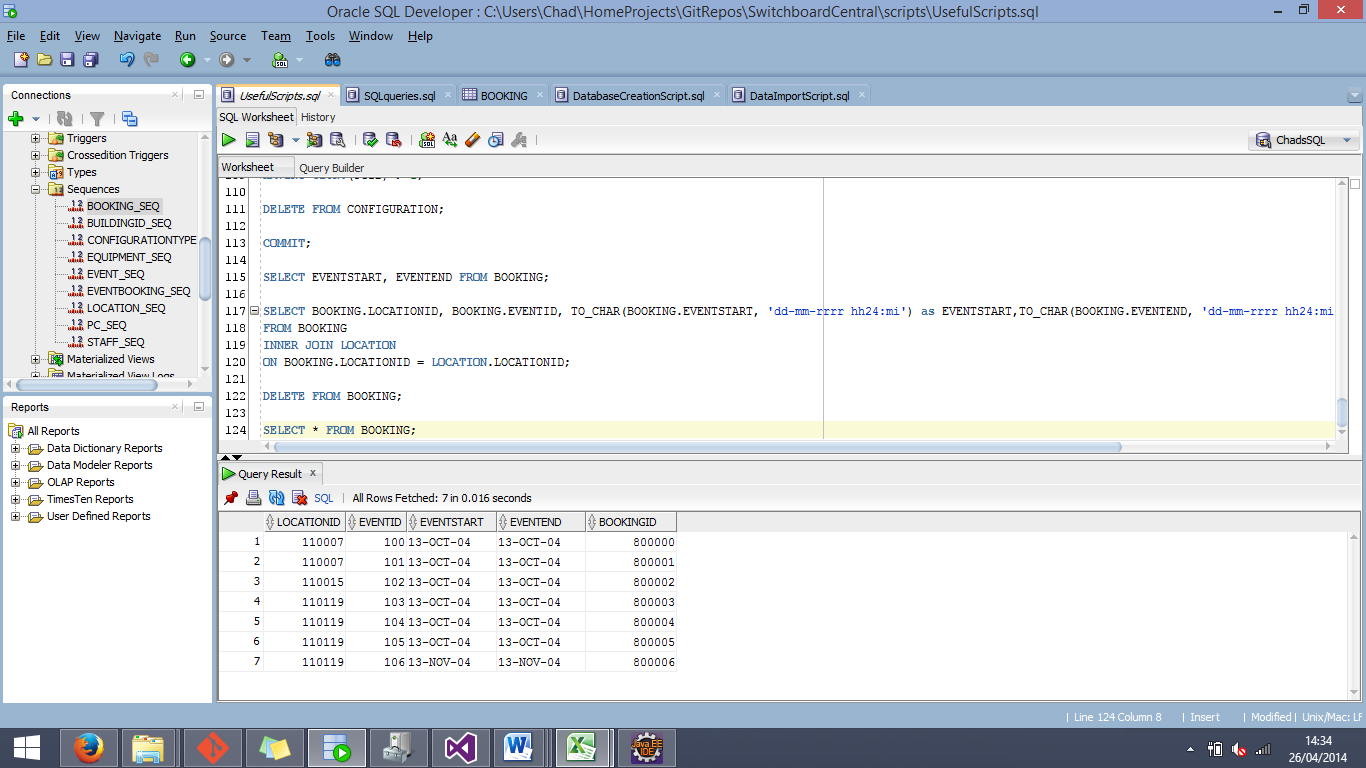
The first action we took when starting to test the data that we loaded was to look back on the source data that we were given and see if there were any aspects of it that would be able to be tested in the final database implementation to see if they were the same as they were in the source data.

As a result of that, we have compiled a series of tests that we performed on the loaded data inside of the database that could be compared to the raw data that we received in the form of an Excel document. From the results of these tests, we will be able to see if the data that we have loaded is valid and correct.

This testing will be split up into two sections, manual testing, and testing involving SQL scripts. The manual testing will largely be done by hand, performing actions such as simply checking the row counts or checking to see if a certain piece of equipment for a certain location has been loaded correctly. The SQL script testing will mostly be in the form of SQL scripts that can check things that would take a much longer time if they were to be done manually.

**Data Testing**

The first test that is performed on the loaded data is to check that the correct number of event bookings have been implemented into the database. This will be done by simply manually checking that the number of rows in the database is equal to the amount of rows in the source data.



As can be seen from the screenshots above, the amount of events in the final database is equal to the amount of data rows in the source document. There are 8 rows in the source document (9 minus the one row for column headers, minus again the one event that does not have a booking), and a total of 7 rows in the BOOKING database entity.

In an attempt to weave automated testing in with the manual testing, we made SQL scripts along the way that were able to perform the same action in the database, in order to more easily compare the data in the database with the source data.

SQL Script:

SELECT COUNT(\*) FROM EVENT;

------

The next series of manual tests revolved around making sure that the total amount of locations in the database is correct, and checking that each building had its correct number of locations

The first test was to compare the total amount of locations in the database to the cumulated amount of locations that were between the different building sheets in the Excel source file. We then compared the amount of locations that had been inserted into the database with our results, and made the table below to show the outcome of the tests.

|  |  |  |  |
| --- | --- | --- | --- |
| Buildings | Source Result | Database Result | Test Result |
| Devon | 96 | 96 | Matching Data |
| Allardyce | 36 | 36 | Matching Data |
| Total | 132 | 132 | Matching Data |

SQL Script:

SELECT COUNT(\*) FROM LOCATION WHERE BUILDINGID = (SELECT BUILDINGID FROM BUILDING WHERE BUILDINGNAME = :BName);

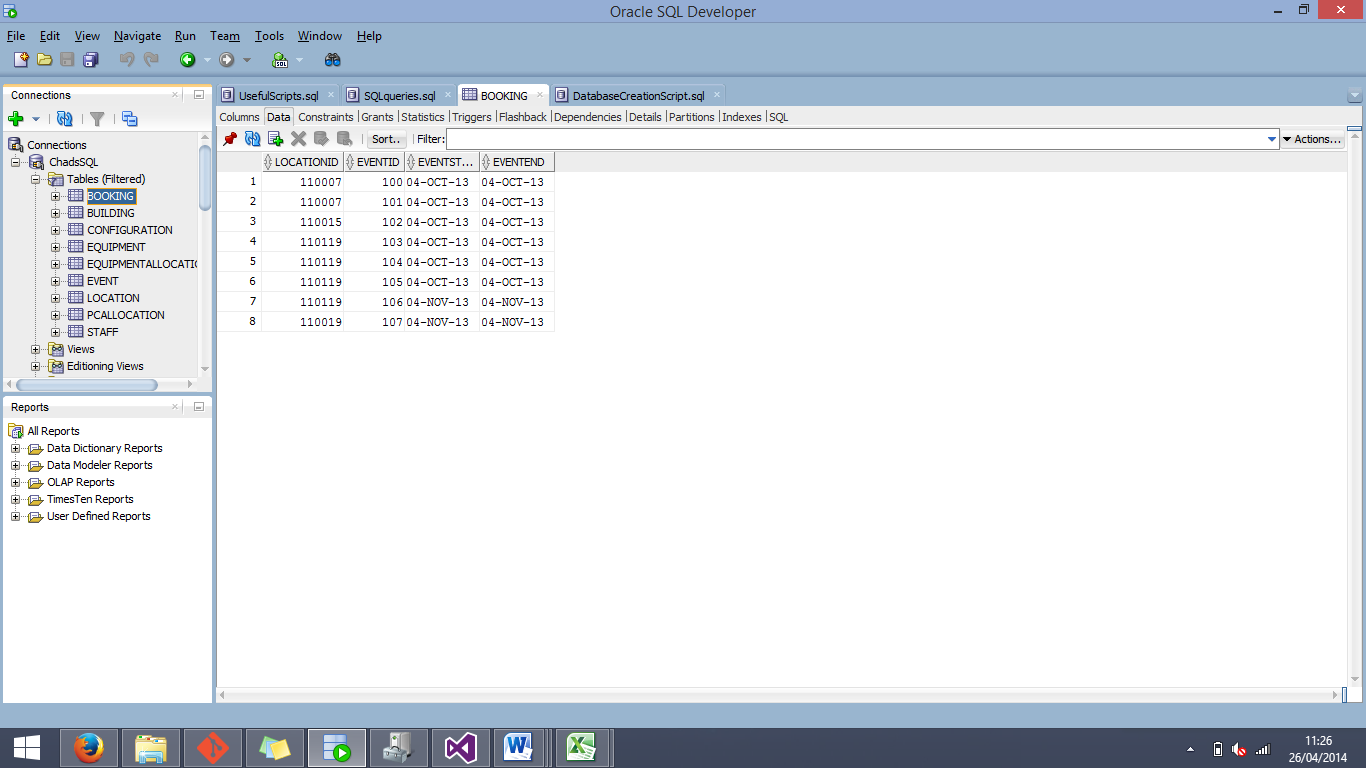
By using the bind variable to input the building name, we were able to check the amount of locations in each of the buildings, and by using this SQL script in conjunction of manual testing (counting the amount of locations in each of the sheets of the Excel file), we were able to prove that the loaded data is accurate in this scenario.

------

Another test that we performed was one exclusively on the database, and it was to check and see if it was possible to extract the date that was inserted into the database in a suitable format.

The data from the database that we used for this exercise was the EVENTSTART and EVENTEND fields from the BOOKING entity. To aid us in this test we created a script that would need to be used if we were going to be able to successfully extract the dates in a suitable format.

The default format of the date field data is shown below.

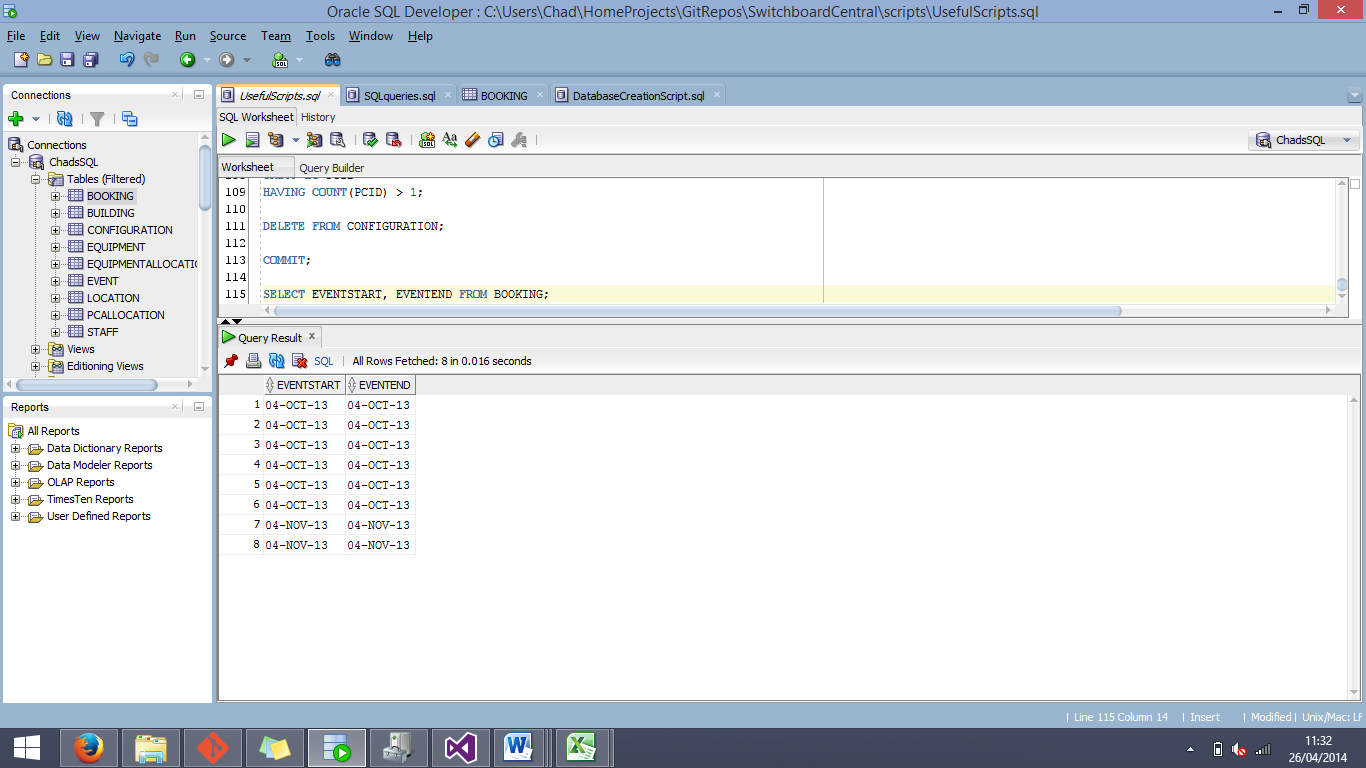


In order to see if this data would be able to show correctly by simply selecting the field, we first tested the following script:

SQL Script:

SELECT EVENTSTART, EVENTEND FROM BOOKING;

The result that was output is the following:



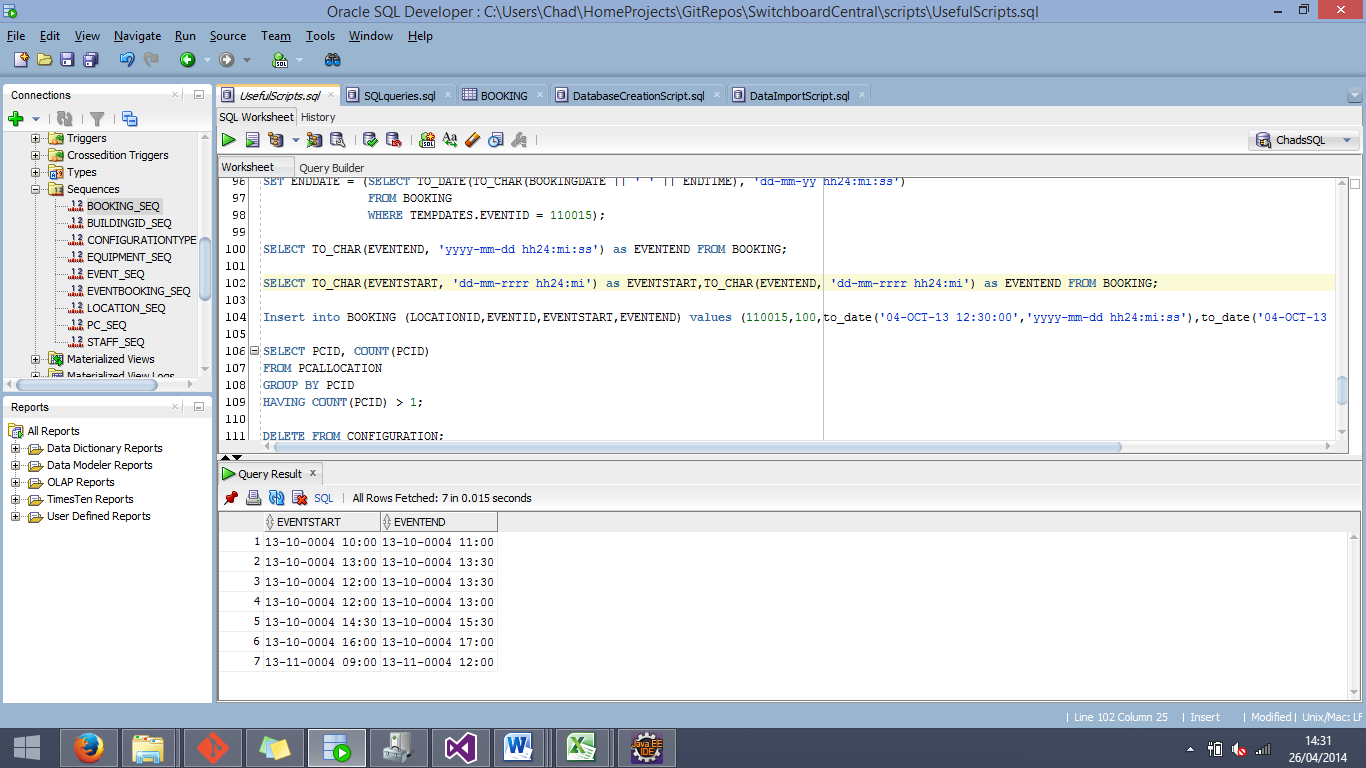
Now we know that we cannot extract the dates by normally selecting them from the BOOKING entity, we must test special methods of extracting, using functions to format the date on the way out.

To do this, we employed the TO\_CHAR() SQL function to format the outputted data to a format that would include the time as well as the date for the field.

SQL Script:

SELECT TO\_CHAR(EVENTSTART, 'dd-mm-rrrr hh24:mi') as EVENTSTART,TO\_CHAR(EVENTEND, 'dd-mm-rrrr hh24:mi') as EVENTEND FROM BOOKING;

As a result from this query, it can be seen that the data can be extrapolated in a valid format from the given date fields.



Now that it is confirmed that we are able to successfully extract the date and time from the BOOKING entity without errors, the next step will be to check and see if the data in the database correctly matches what was loaded from the source data.

To do this, we are going to combine manual checking with an SQL script to select all of the valid data from the database that would be needed to prove the authenticity of the test.

SQL Script:

SELECT BOOKINGID, EVENT.EVENTPURPOSE, TO\_CHAR(BOOKING.EVENTSTART, 'dd-mm-rrrr hh24:mi') as EVENTSTART,TO\_CHAR(BOOKING.EVENTEND, 'dd-mm-rrrr hh24:mi') as EVENTEND, LOCATION.ROOMNO

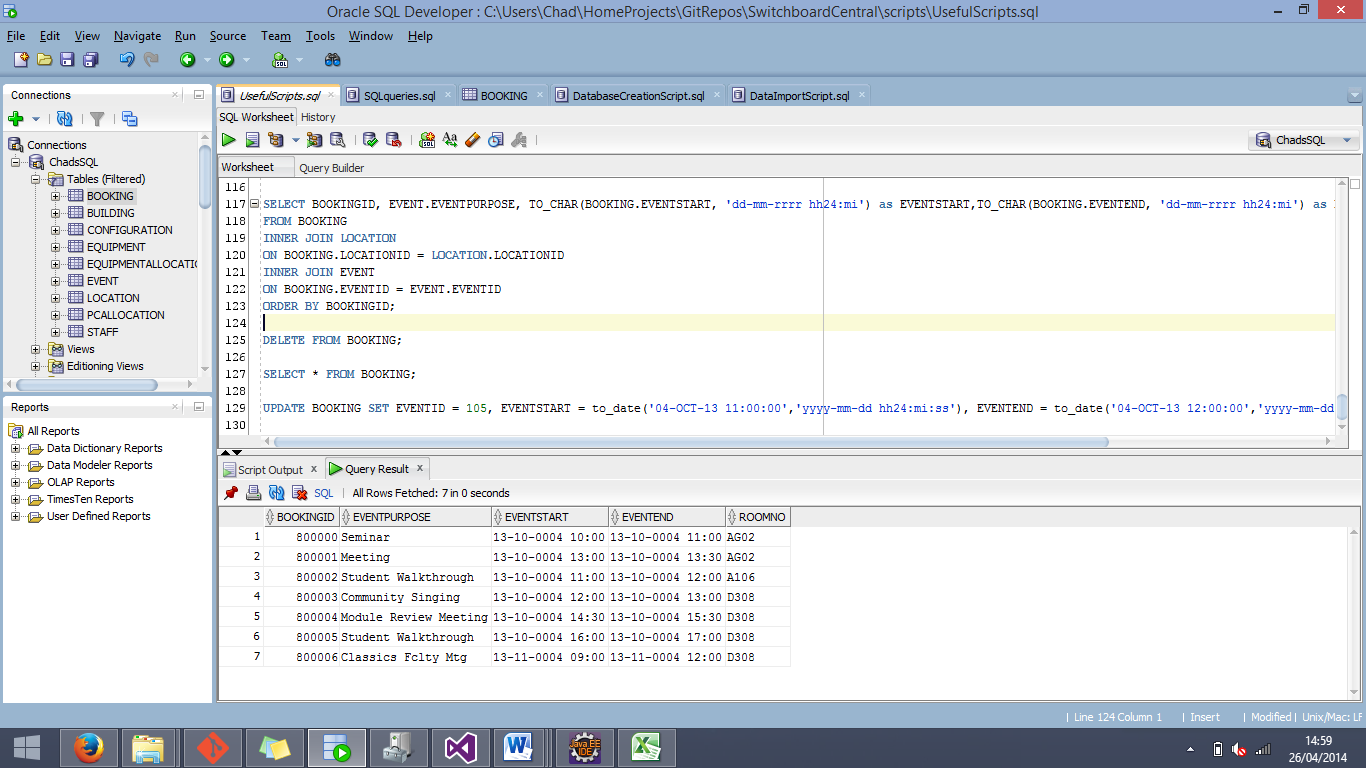
FROM BOOKING

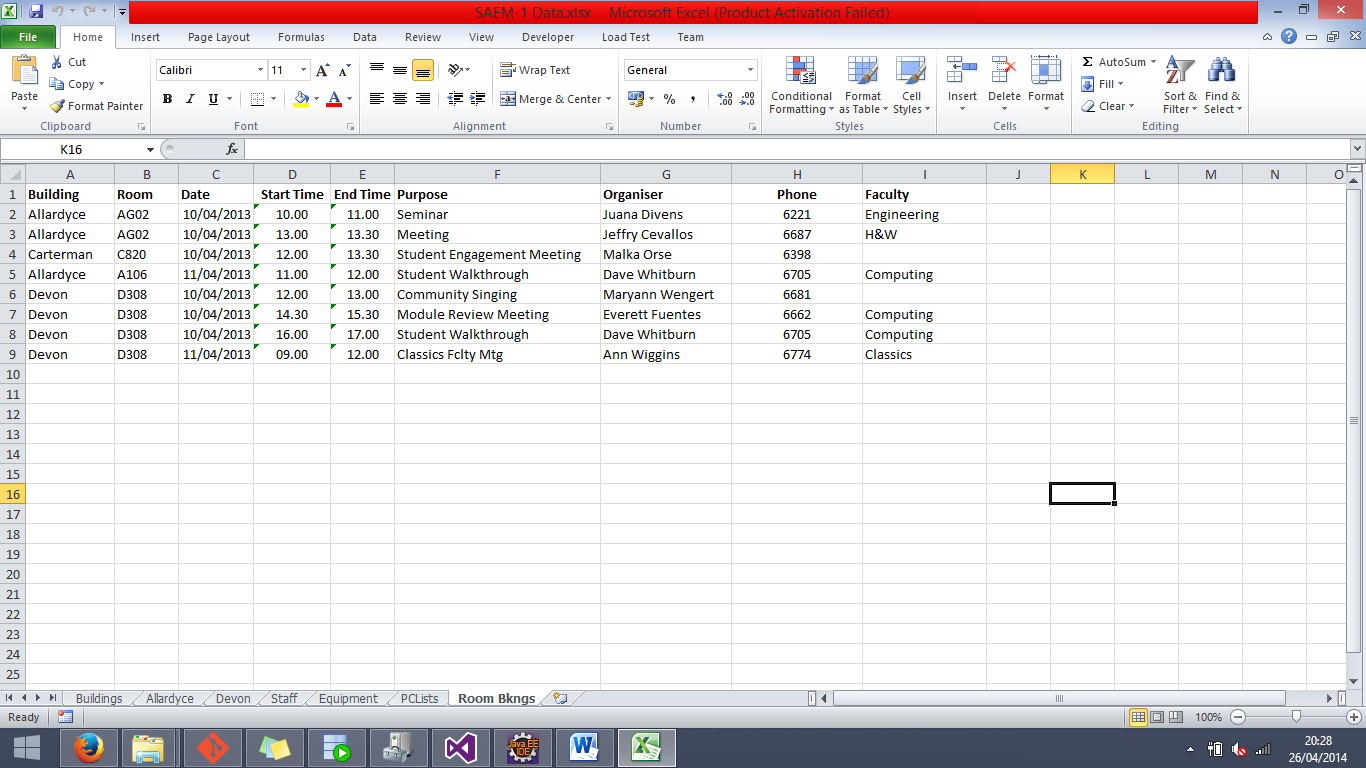
INNER JOIN LOCATION

ON BOOKING.LOCATIONID = LOCATION.LOCATIONID

INNER JOIN EVENT

ON BOOKING.EVENTID = EVENT.EVENTID;

From this script, we are able to see the ROOMNO, EVENTSTART, EVENTEND, and the IDs of the booking that each of these bookings corresponds to.



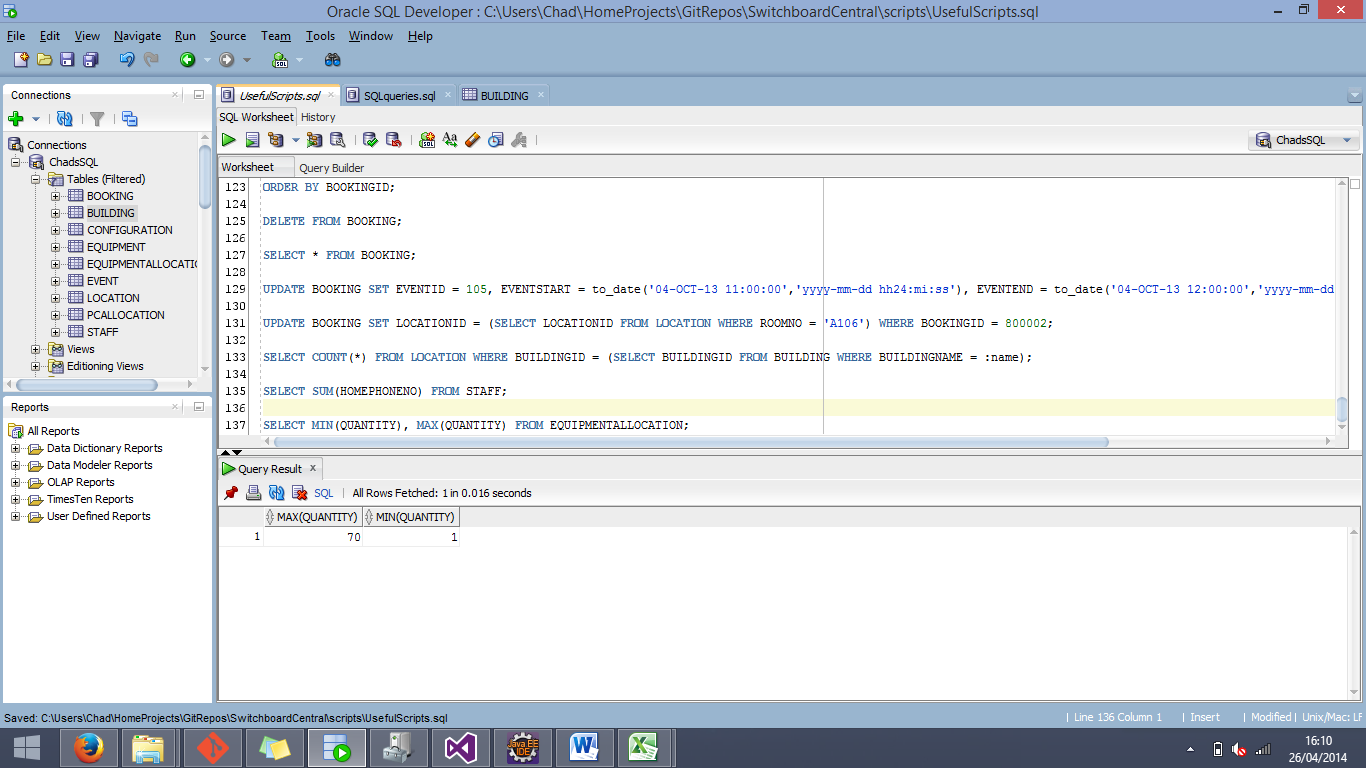
The data from the source Excel document and the data from the final database implementation is a match, proving that the loaded data is accurate in this section.

------

The next test that we scheduled to perform on the database was to further test the correctness of the data that we loaded. Our task was to check the range between the smallest amount quantity that is entered into the EQUIPMENTALLOCATION table, and the largest. To do this, we first incorporated the =MIN() and =MAX() functions inside of Excel to find the appropriate values, and then use a script inside of SQL Developer to find the corresponding values in the database.

SQL Script:

SELECT MIN(QUANTITY), MAX(QUANTITY) FROM EQUIPMENTALLOCATION;



|  |  |  |  |
| --- | --- | --- | --- |
| Source | Minimum Value | Maximum Value | Test Result |
| Source Data (Excel) | 1 | 70 | Matching Data |
| Database | 1 | 70 | Matching Data |

While admittedly this test is by no means completely conclusive as to whether or not the data inside the EQUIPMENTALLOCATION table is completely accurate, it does give at least some measure of assurance.

------

The next test that we performed on the database was one that was to make sure that any of the given CHECK constraints would perform their duty. For this test we chose the Quantity attribute of the EQUIPMENTALLOCATION entity, and attempting to insert an entry with a negative value in the given field. The expected result of this test is for the insert statement to fail due to a violated constraint.

SQL Script:

INSERT INTO EQUIPMENTALLOCATION (LOCATIONID, EQUIPMENTID, QUANTITY) VALUES (110001, 111005, -52);

The result of trying to apply the above script resulted in the following:



From this error, we can now confirm that the CHECK constraints associated with the Quantity field to prevent it from falling below 0 is effective.

The next series of tests involve performing SUM functions on both the Excel source document and the database. These tests will further enforce the assumption that we have loaded the data correctly. The test data will be shown in a series of tables below.

SUM Test 1: Staff Work Phone Numbers

|  |  |  |
| --- | --- | --- |
| Data source | WorkPhoneSUM | Test Result |
| Source File (Excel) | 118277 | Matching Data |
| Database | 118277 | Matching Data |

After comparing the SUMs of the respective columns, it can be confirmed that they are matching and that we are one step closer to proving the correctness of the loaded data.

SUM Test 2: Configuration Capacities

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Source | Horseshoe Capacity | Island Capacity | Row Capacity | Added Result | Test Result |
| Source File (Excel) | 80 | 718 | 1560 | 2358 | Matching Data |
| Database | 80 | 718 | 1560 | 2358 | Matching Data |

After comparing the SUMs of the respective columns, and the sum of those results, it can be confirmed that they are matching and that we are one step closer to proving the correctness of the loaded data.

SUM Test 3: Lengths and Widths of Locations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Source | Length | Width | Added Result | Test Result |
| Source File (Excel) | 1578.5 | 911.5 | 2490 | Matching Data |
| Database | 1578.5 | 911.5 | 2490 | Matching Data |

After comparing the SUMs of the respective columns, and the sum of those results, it can be confirmed that they are matching and that we are one step closer to proving the correctness of the loaded data.

------

The next test will once again call for us to manually check the source data that we were given with the data that is currently in the database. The purpose of this test is to check and see if the Primary Key of the STAFF entity has been correctly loaded into the EVENT entity in the form of an organiser reference.

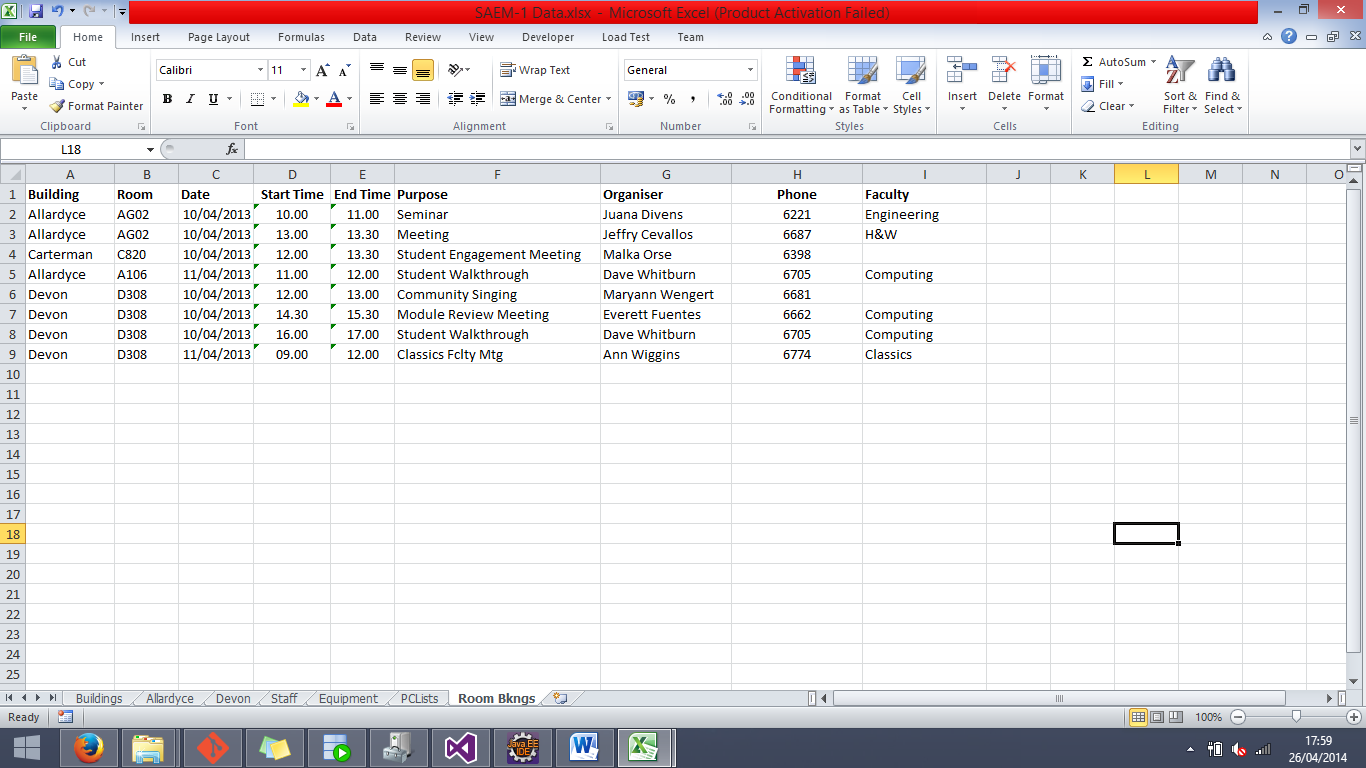
SQL Script:

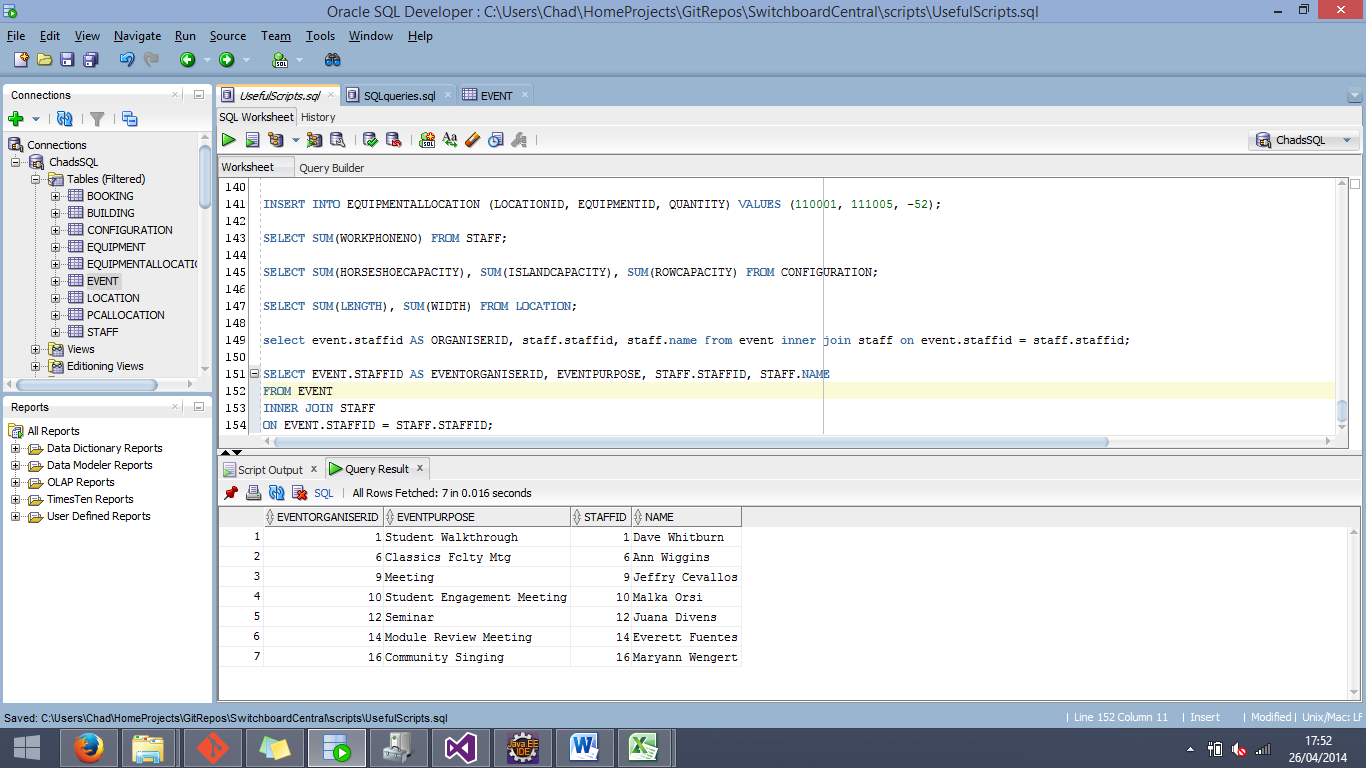
SELECT EVENT.STAFFID AS ORGANISERID, STAFF.STAFFID, STAFF.NAME

FROM EVENT

INNER JOIN STAFF

ON EVENT.STAFFID = STAFF.STAFFID;

From this SQL script we are able to see the version of StaffID that is listed in the EVENT entity, and compare it with the StaffID in the STAFF entity, along with the name of the associated staff. From this result we will be able to compare it with the source data in the Excel document to determine the accuracy of the loaded data.



As can be seen in the images above, each member of staff that is associated with an event is correctly identified by their StaffID inside of the EVENT table.

**Testing Conclusion**

Throughout all of the different tests that were performed on the loaded data inside the database, there were none that gave any evidence that the data was incorrectly loaded. So from the results of these tests, it would be possible to prove that the data loaded from the Excel source document is accurate.

**QUERY TESTING**

Query 1 Testing

Code tested:

SELECT STAFF.NAME AS STAFF\_NAME, STAFF.WORKPHONENO AS WORK\_PHONE, STAFF.HOMEPHONENO AS HOME\_PHONE, Building.buildingname AS BUILDING

FROM STAFF

INNER JOIN LOCATION

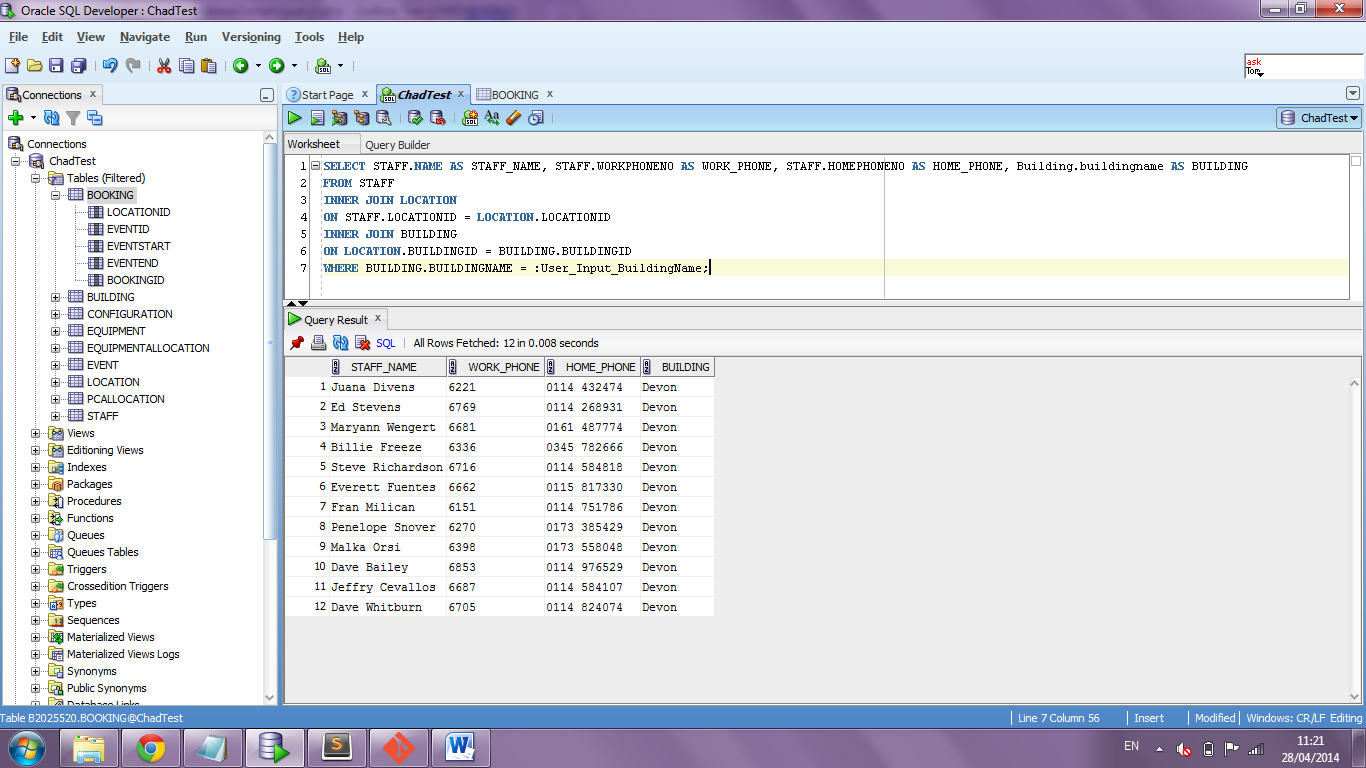
ON STAFF.LOCATIONID = LOCATION.LOCATIONID

INNER JOIN BUILDING

ON LOCATION.BUILDINGID = BUILDING.BUILDINGID

WHERE BUILDING.BUILDINGNAME = :User\_Input\_BuildingName;

:User\_Input\_BuildingName = ‘Devon’

Result:

When tested in SQL Developer the returned results are as expected meeting the critera of the query. For testing perposes the bulding name is returned, this allows incorrect data to be visible and ensure all the staff returned meet the building criteria

Code tested:

SELECT STAFF.NAME AS STAFF\_NAME, STAFF.WORKPHONENO AS WORK\_PHONE, STAFF.HOMEPHONENO AS HOME\_PHONE, Building.buildingname AS BUILDING

FROM STAFF

INNER JOIN LOCATION

ON STAFF.LOCATIONID = LOCATION.LOCATIONID

INNER JOIN BUILDING

ON LOCATION.BUILDINGID = BUILDING.BUILDINGID

WHERE BUILDING.BUILDINGNAME = 12345;

Result:

An error is returned as expected; “Invalid number”.

Query 2 Testing:

Code tested:

SELECT EQUIPMENT.NAME AS EQUIPMENT\_NAME, EQUIPMENTALLOCATION.QUANTITY, LOCATION.ROOMNO AS ROOM\_NUMBER, LOCATION.FLOOR, LOCATION.TYPE AS ROOM\_TYPE, BUILDING.BUILDINGNAME AS BUILDING

FROM EQUIPMENTALLOCATION

INNER JOIN EQUIPMENT

ON EQUIPMENTALLOCATION.EQUIPMENTID = EQUIPMENT.EQUIPMENTID

INNER JOIN LOCATION

ON EQUIPMENTALLOCATION.LOCATIONID = LOCATION.LOCATIONID

INNER JOIN BUILDING

ON LOCATION.BUILDINGID = BUILDING.BUILDINGID

WHERE BUILDING.BUILDINGNAME = :User\_Input\_BuildingName

AND

LOCATION.ROOMNO IS NOT NULL

AND

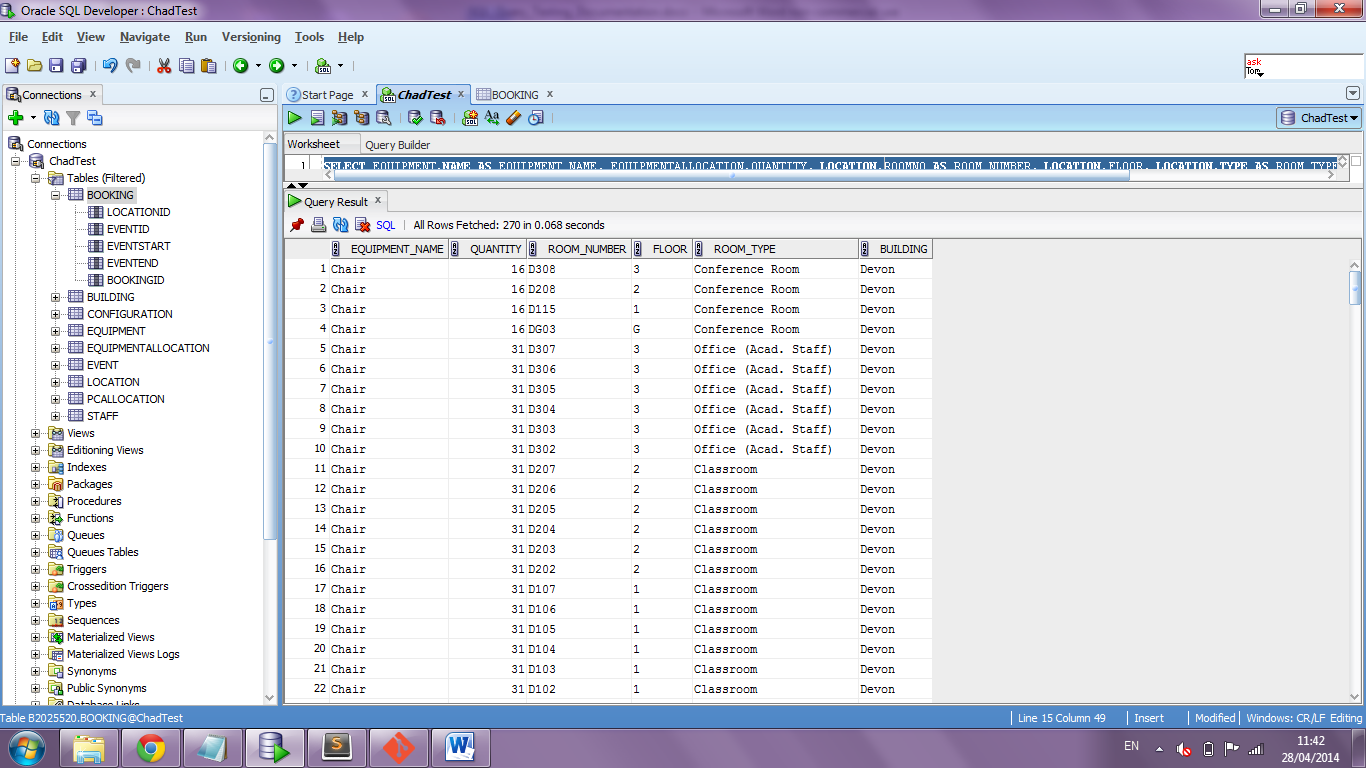
LOCATION.CURRENTCONFIG IS NOT NULL

AND

EQUIPMENTALLOCATION.LOCATIONID IS NOT NULL;

:User\_Input\_BuildingName = ‘Devon’

Result:

The data returned is correct and as expected. The buildingname column shows that the data is related to the correct building.

Query 3 Testing:

Code tested:

SELECT LOCATION.LOCATIONID, LOCATION.ROOMNO, LOCATION.TYPE, LOCATION.FLOOR, BUILDING.BUILDINGNAME, LOCATION.CURRENTCONFIG,

CONFIGURATION.ISLANDCAPACITY, CONFIGURATION.HORSESHOECAPACITY, CONFIGURATION.ROWCAPACITY

FROM LOCATION

INNER JOIN CONFIGURATION

ON LOCATION.LOCATIONID = CONFIGURATION.LOCATIONID

INNER JOIN BUILDING

ON LOCATION.BUILDINGID = BUILDING.BUILDINGID

WHERE LOCATION.LOCATIONID NOT IN

(

SELECT DISTINCT LOCATIONID

FROM BOOKING

WHERE

(

(TO\_DATE(:User\_Input\_Event\_Start, 'dd-mm-rrrr hh24:mi') BETWEEN EVENTSTART AND EVENTEND)

OR

(TO\_DATE(:User\_Input\_Event\_End, 'dd-mm-rrrr hh24:mi') BETWEEN EVENTSTART AND EVENTEND)

OR

(TO\_DATE(:User\_Input\_Event\_Start, 'dd-mm-rrrr hh24:mi') <= EVENTSTART AND TO\_DATE(:User\_Input\_Event\_End, 'dd-mm-rrrr hh24:mi') >= EVENTEND)

)

)

AND

BUILDING.BUILDINGNAME = :User\_Input\_BuildingName

AND

(

CONFIGURATION.ROWCAPACITY >= :User\_Input\_Room\_Capacity

OR

CONFIGURATION.ISLANDCAPACITY >= :User\_Input\_Room\_Capacity

OR

CONFIGURATION.HORSESHOECAPACITY >= :User\_Input\_Room\_Capacity

)

AND

LOCATION.ROOMNO IS NOT NULL

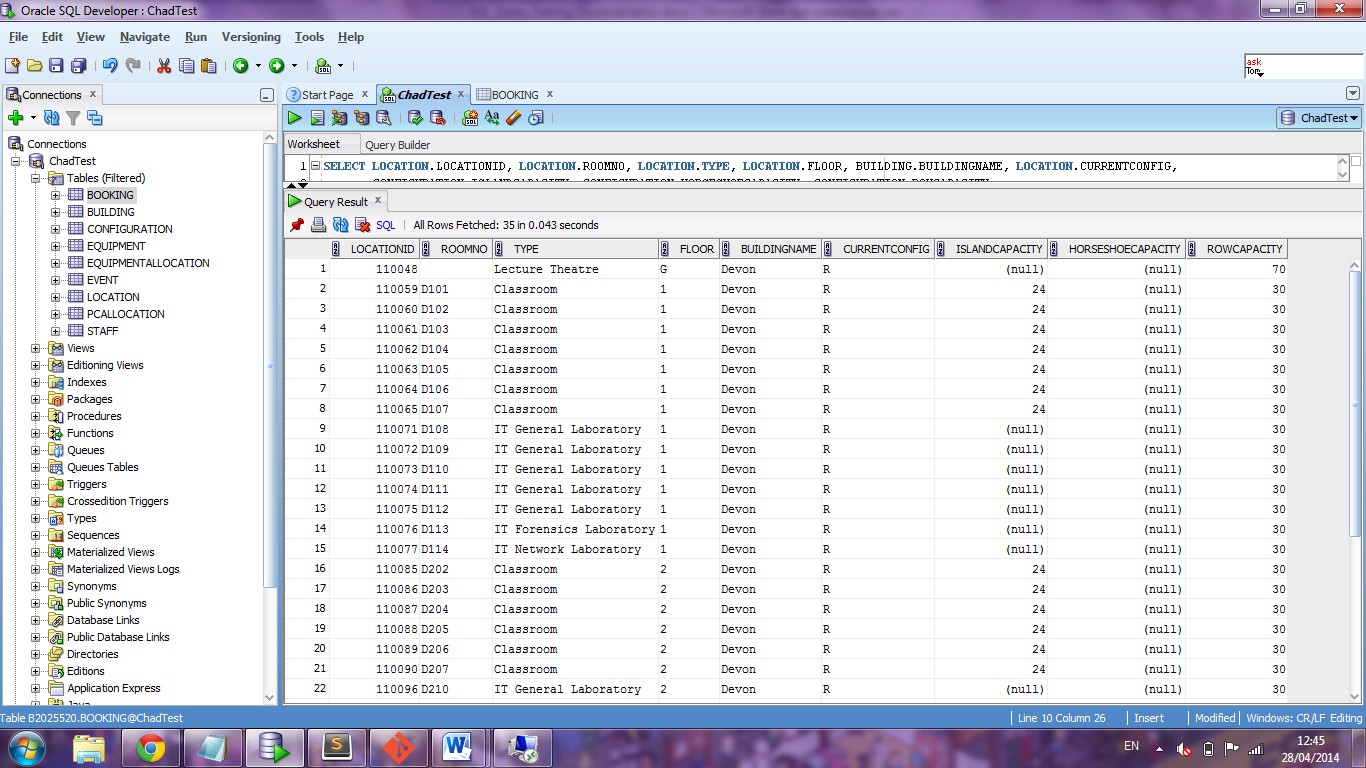
ORDER BY LOCATION.LOCATIONID

:User\_Input\_BuildingName = ‘Devon’

:User\_Input\_Room\_Capacity = 30

:User\_Input\_Event\_End = 04/10/2013 13:00

:User\_Input\_Event\_Start = 04/10/2013 12:00

Result:

The rows returned are correct fitting the critera of the requested query.

Query 4:

Code tested:

INSERT INTO EVENT (EVENTNAME, EVENTPURPOSE, STAFFID)

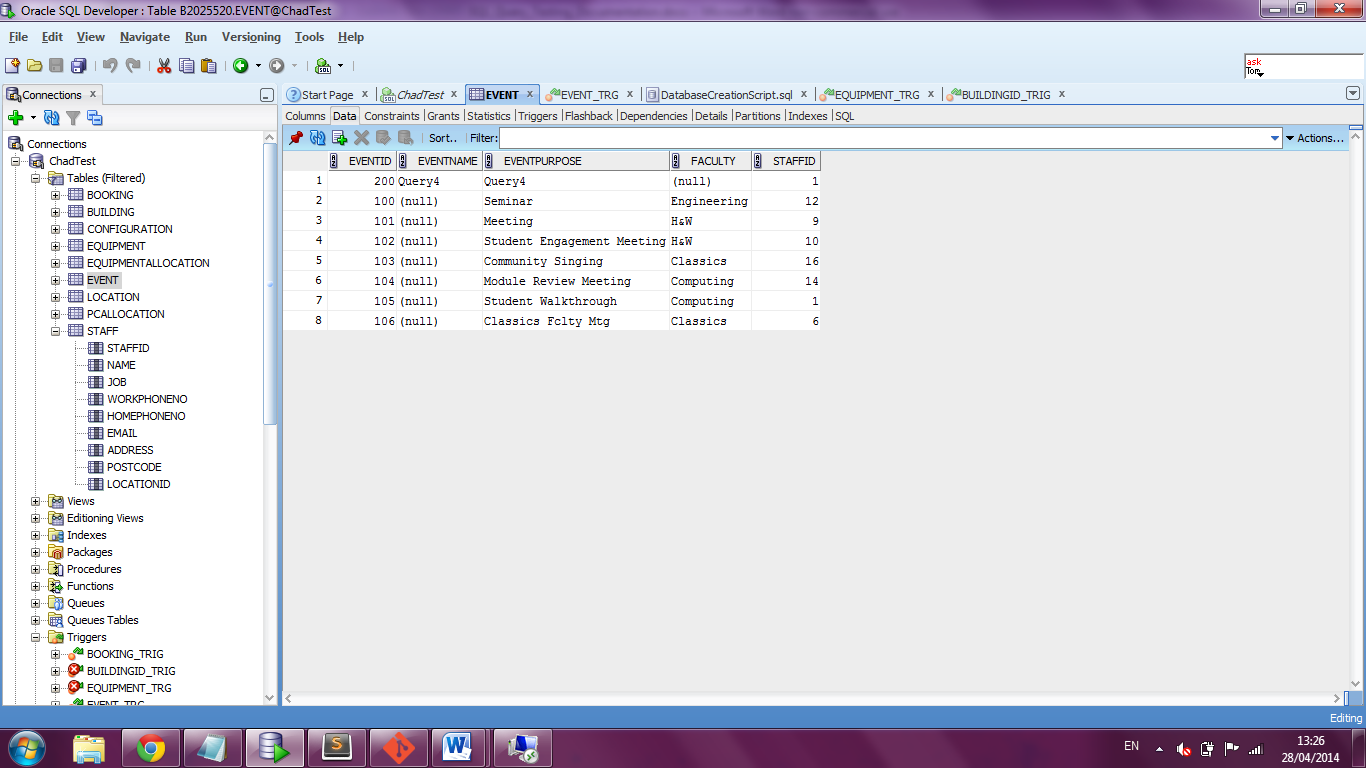
VALUES (:User\_Input\_EventName, :User\_Input\_EventPurpose, (SELECT STAFFID FROM STAFF WHERE STAFF.NAME = :User\_Input\_Organiser));

:User\_Input\_EventName = ‘Query4’

:User\_Input\_EventPurpose = ‘Query4’

:User\_Input\_Organiser = ‘Dave Whitburn’

Result:



The row has been correctly inserted and the staff id is correctly selected from the staff table.

Query 5:

Code tested:

INSERT INTO BOOKING (LOCATIONID, EVENTID, EVENTSTART, EVENTEND)

VALUES ((SELECT LOCATIONID

FROM LOCATION

WHERE ROOMNO = :User\_Input\_RoomNo), (SELECT EVENTID

FROM EVENT

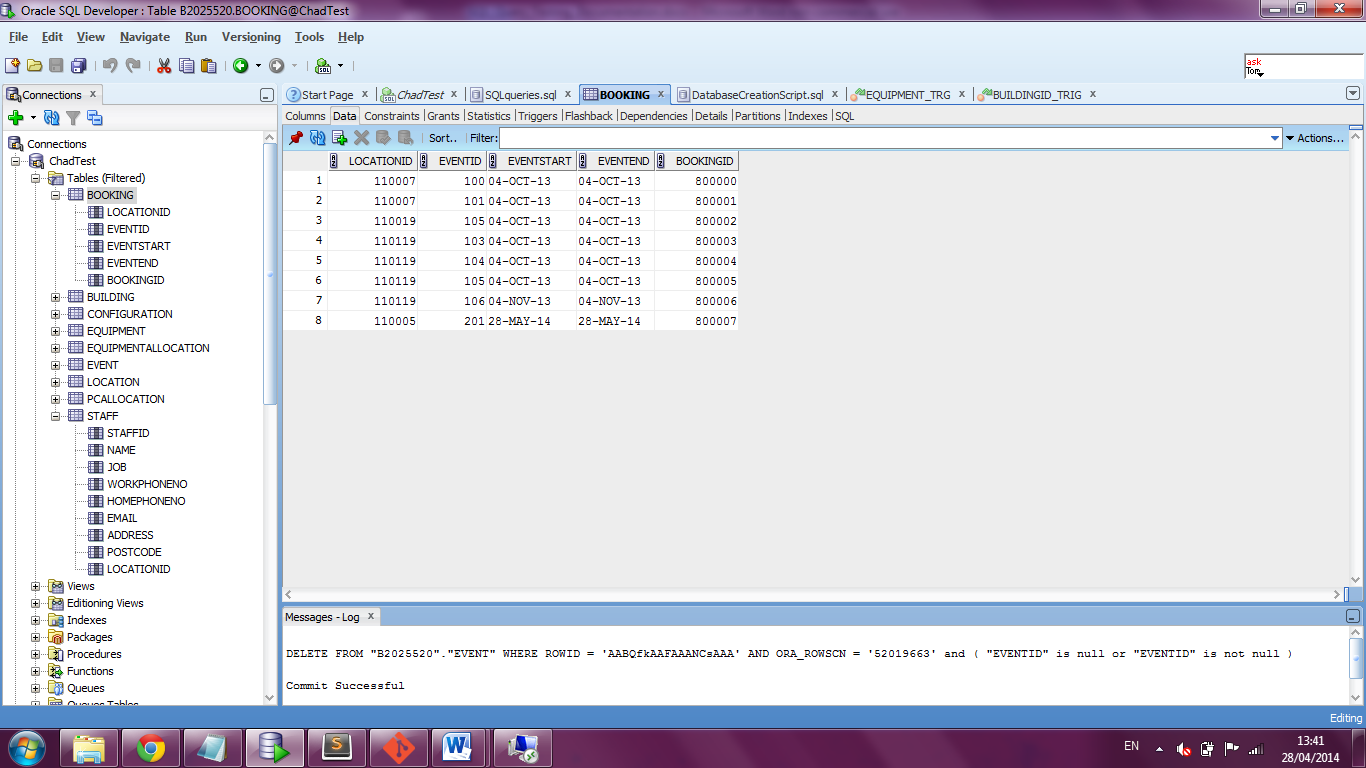
WHERE EVENTNAME = :User\_Input\_EventName), TO\_DATE(:User\_Input\_EventStart, 'dd-mm-rrrr hh24:mi'), TO\_DATE(:User\_Input\_EventEnd, 'dd-mm-rrrr hh24:mi'));

:User\_Input\_EventName = ‘Query4’

:User\_Input\_EventStart = ‘28/05/2014 12:00’

:User\_Input\_EventEnd = ‘28/05/2014 13:00’

Result:

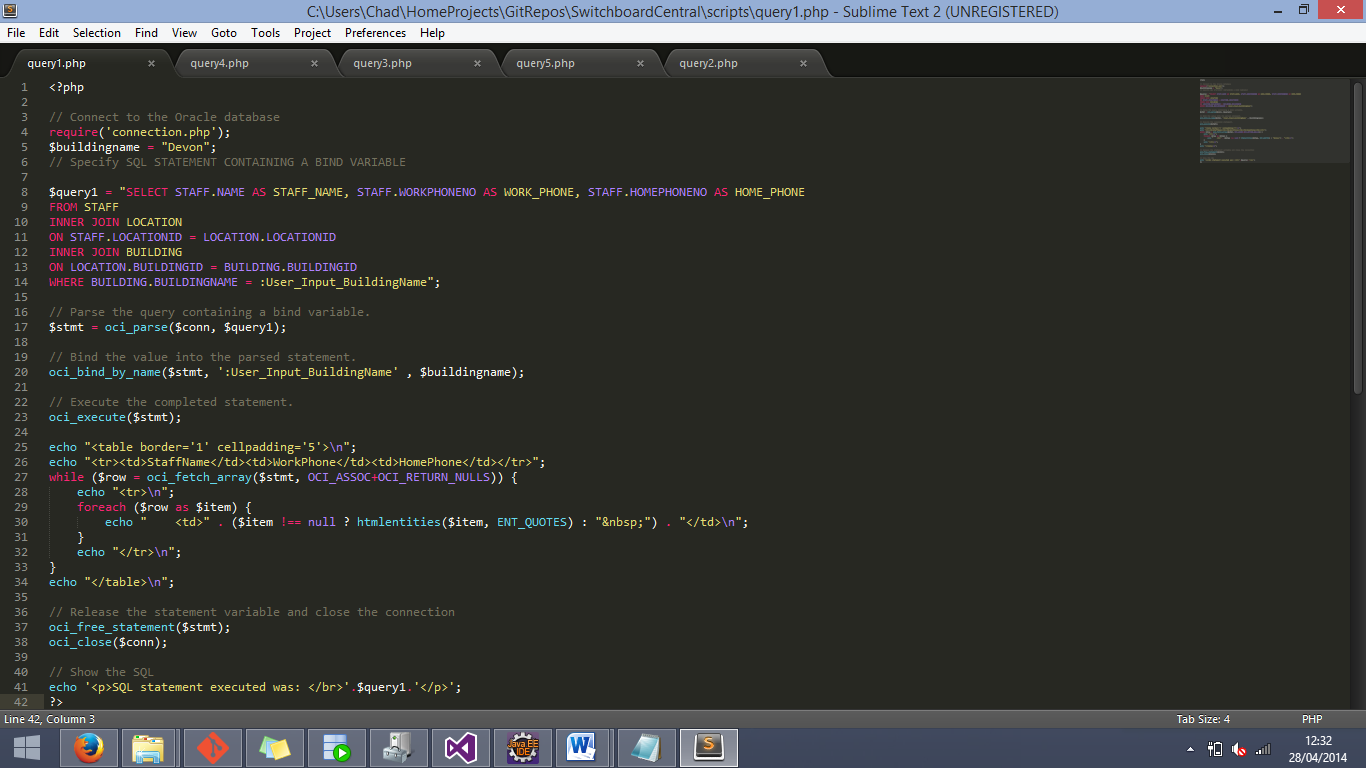


As seen on row 8 the data has been correctly inserted. The selected foreign keys from the Location and Event tables are correct.

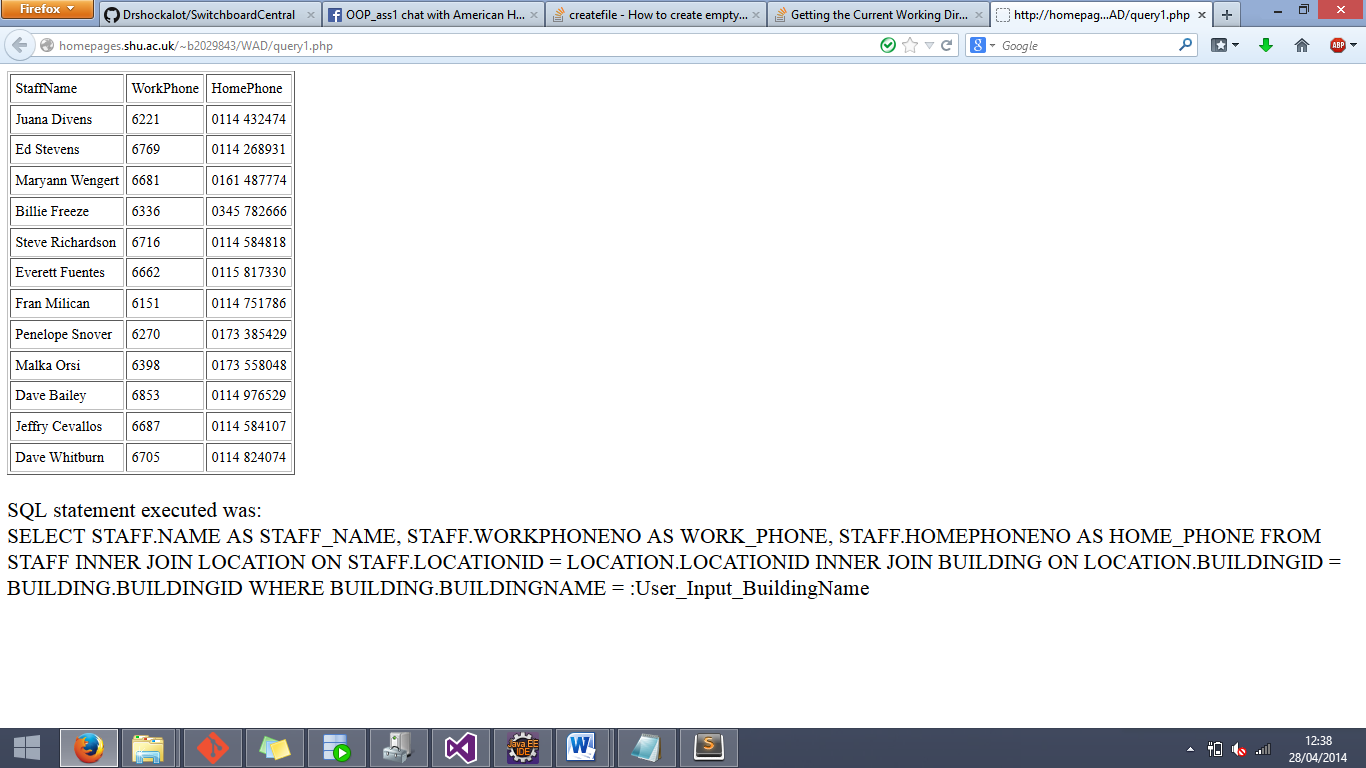
**Web Queries**

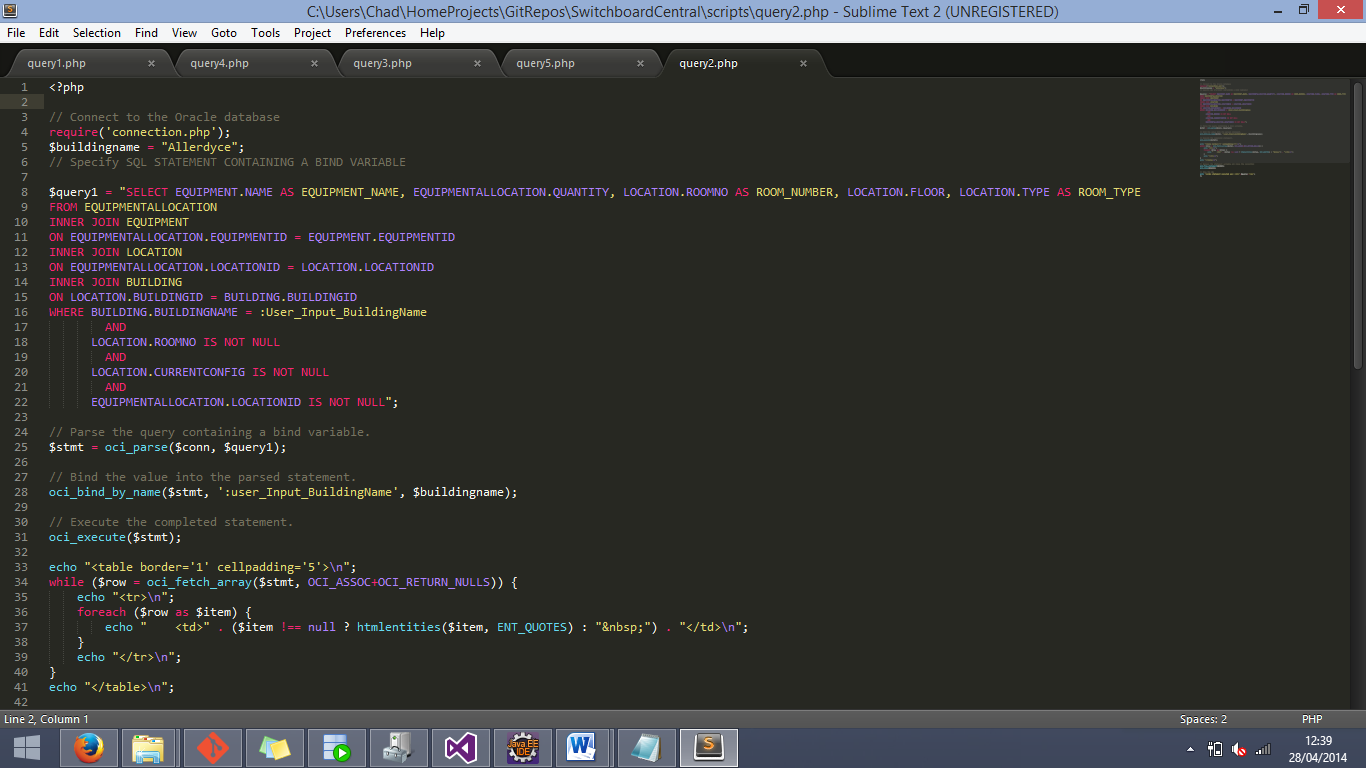
In this section we will review the SQL queries that were converted into prepared statements using the PHP scripting language. Note that there is no user input for these web pages (as it was not required), so the data for the bind variables is hard coded.

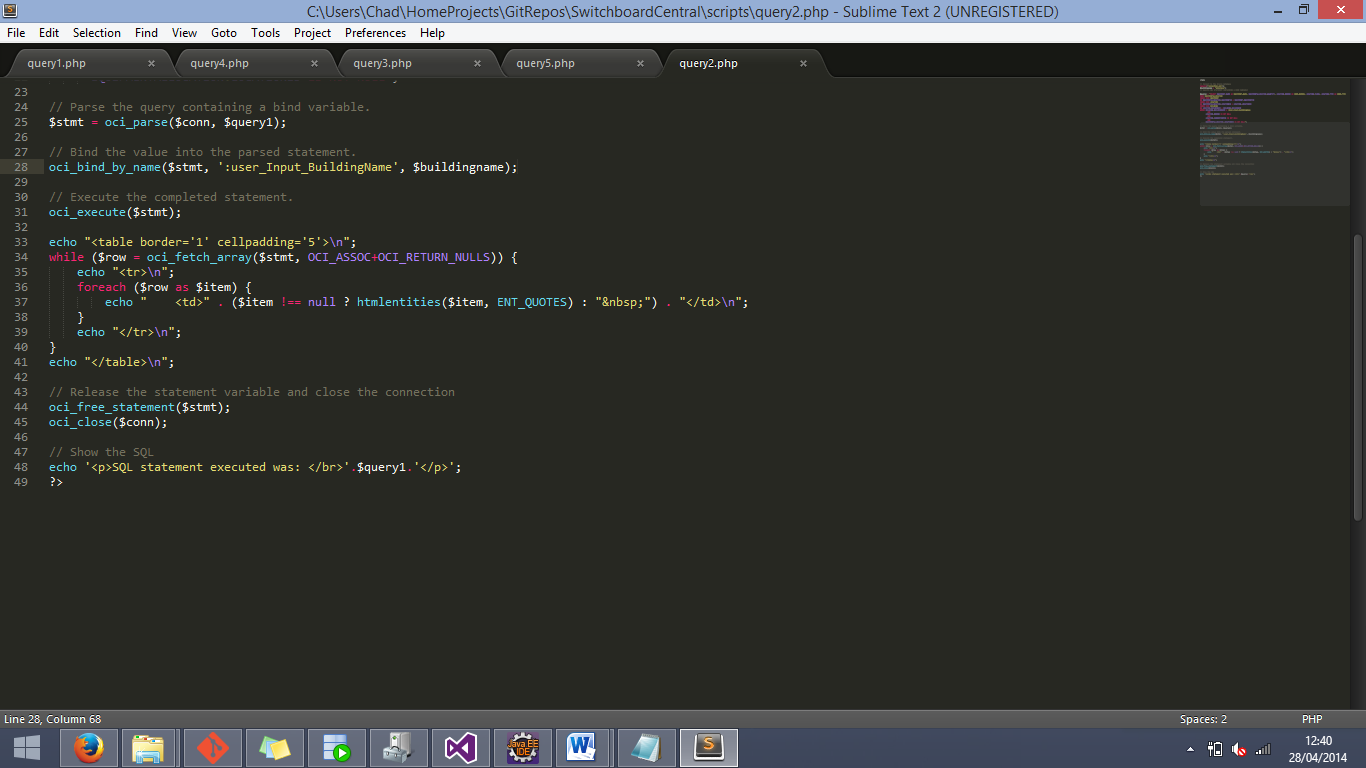
**Query 1**



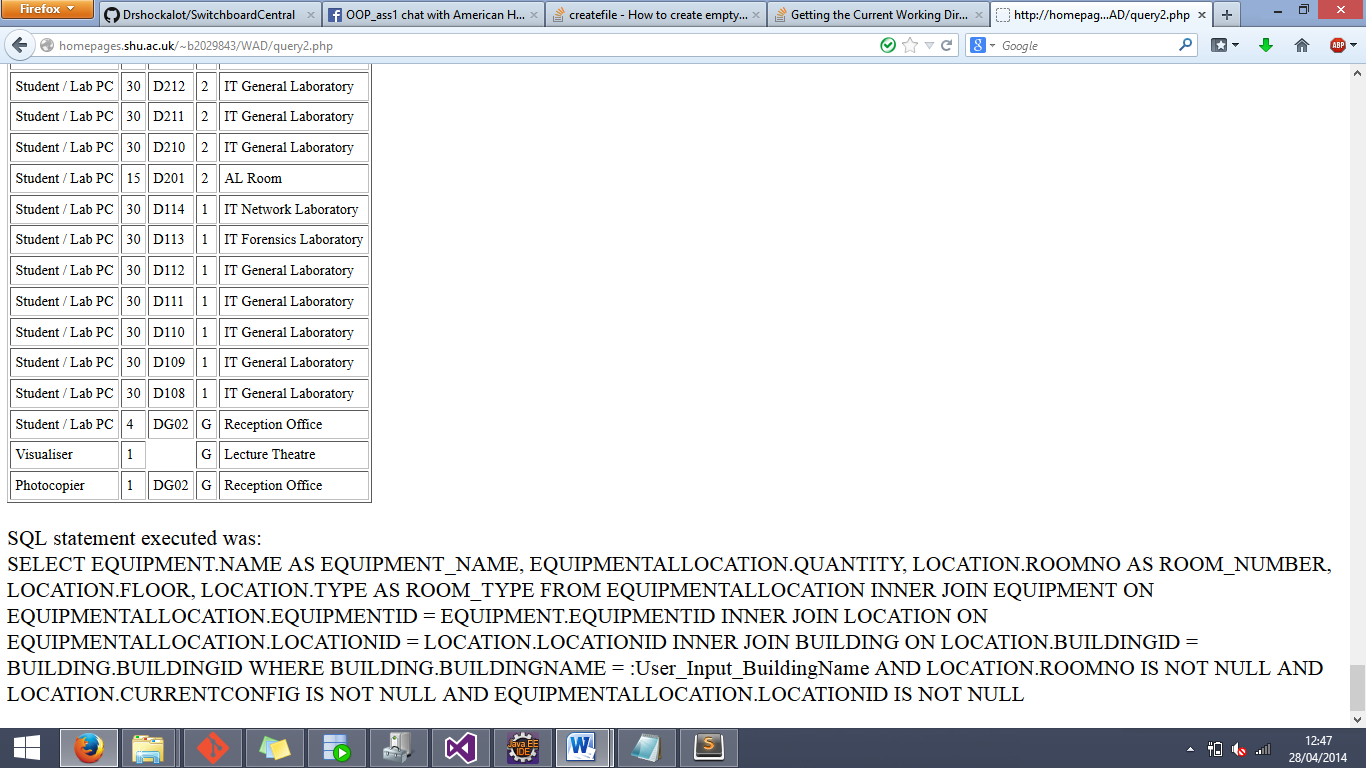
This query shows the first query, with the bind variable data being hard coded as “Devon”. The result of the above query is shown below.



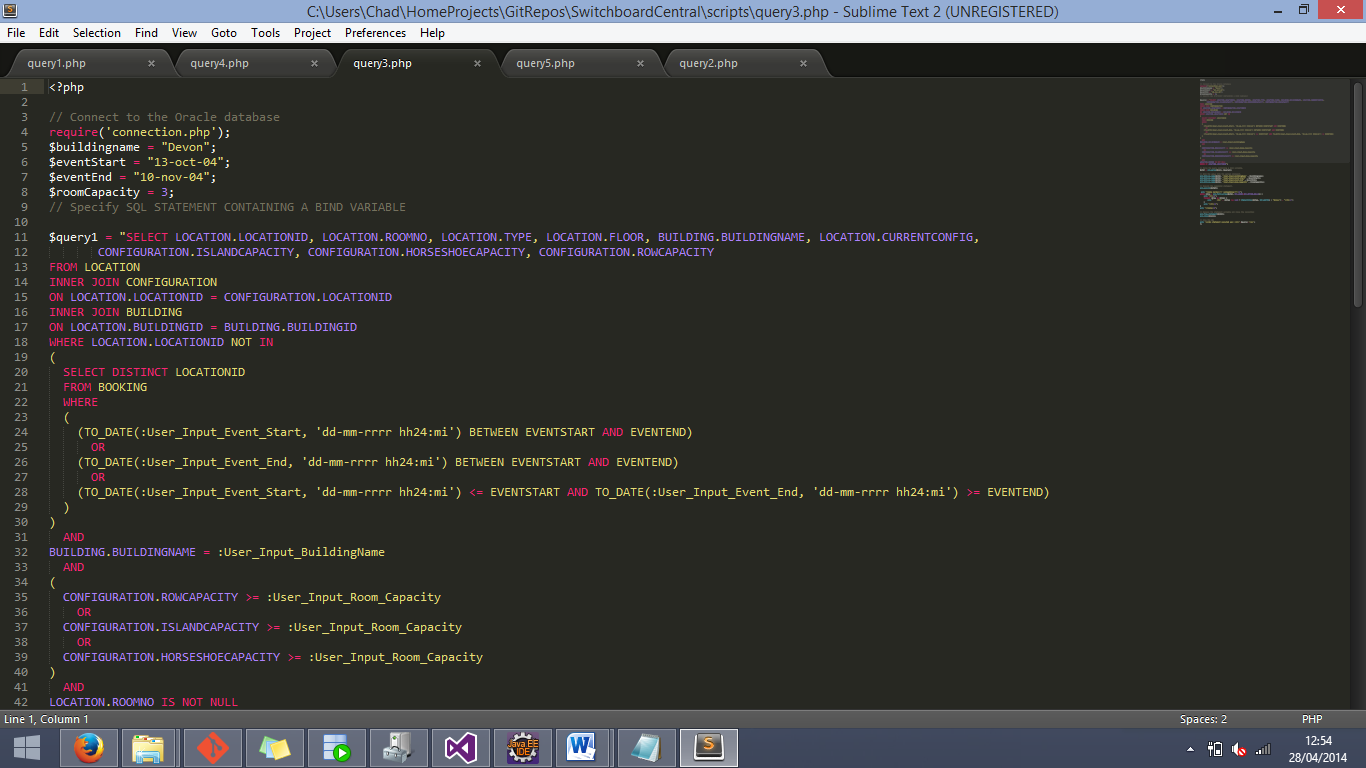
**Query 2**

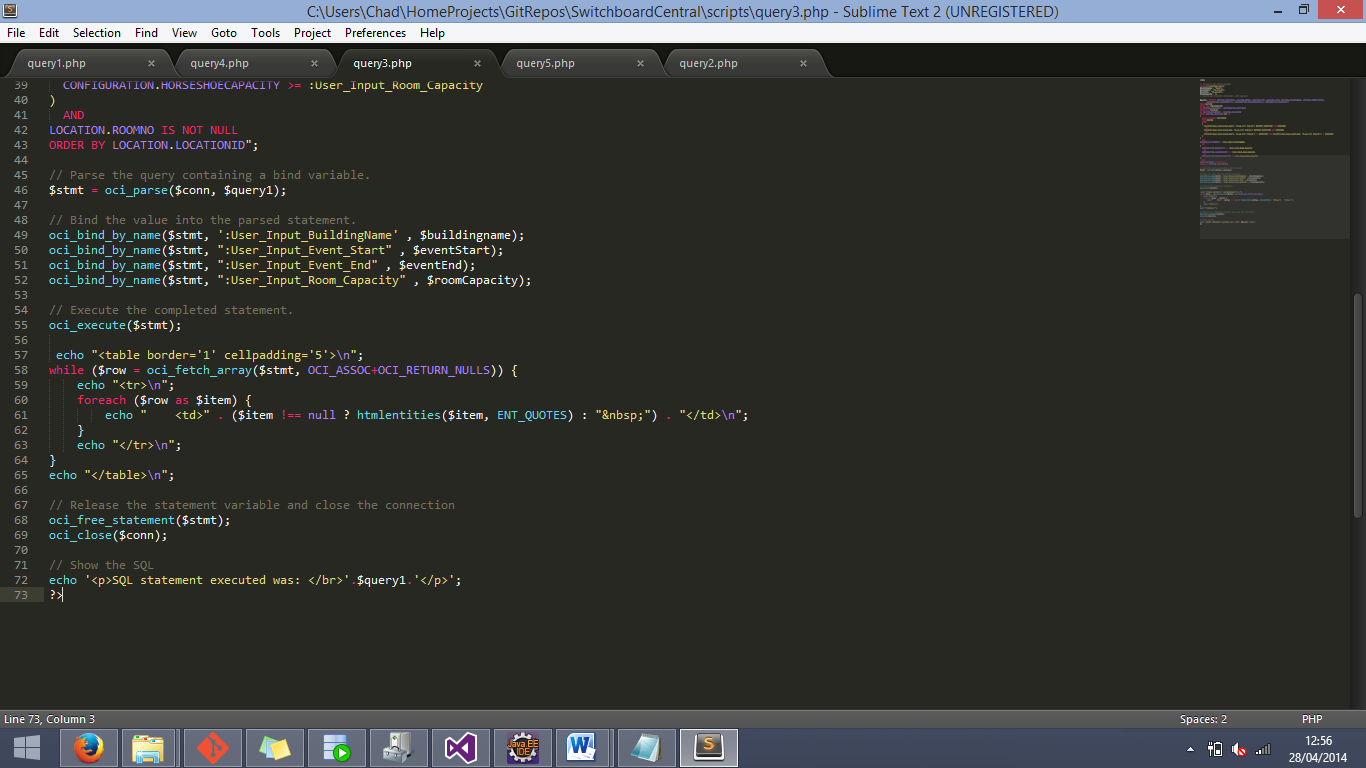


This image shows the second query, with the bind variable data being hard coded as “Allardyce”. The result of the above query is shown below.

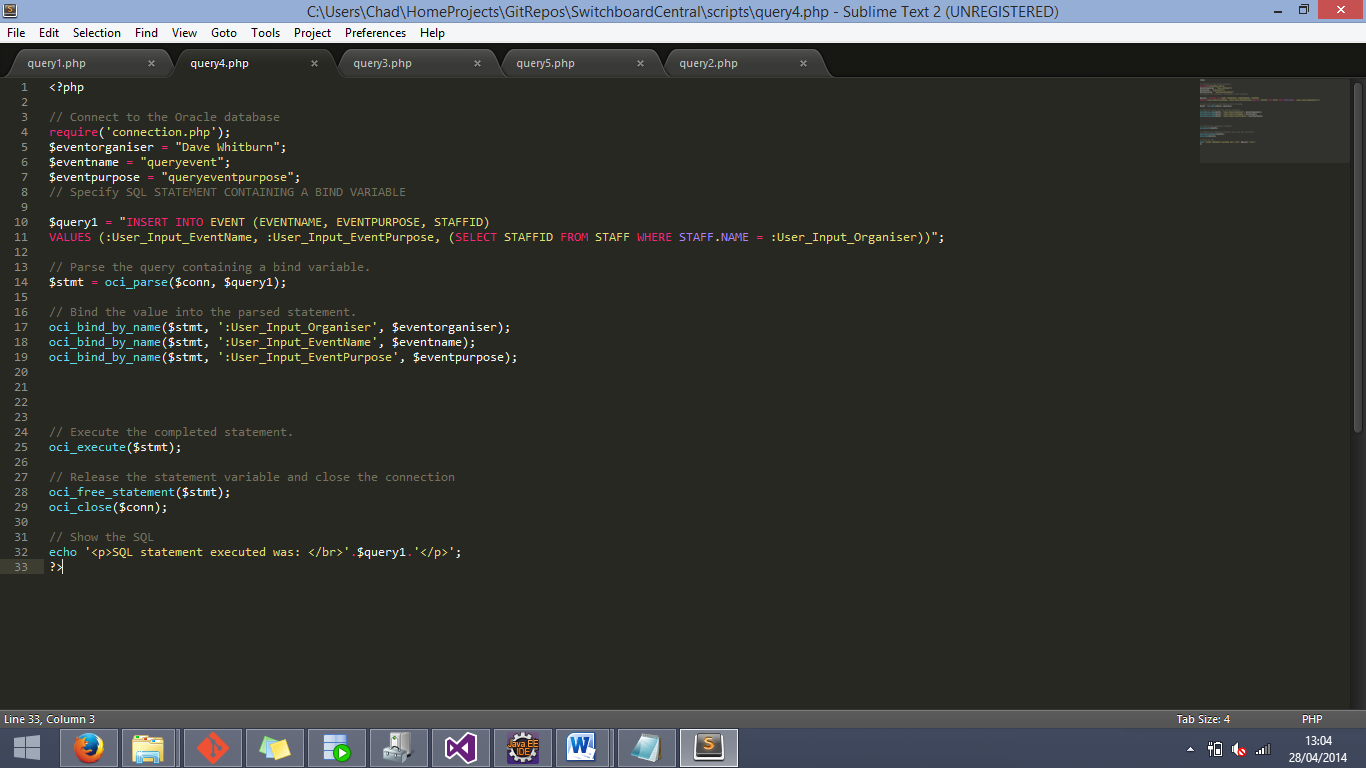
The entire result obviously cannot be shown on a single screen, so this shown only a portion of the result.

**Query 3**



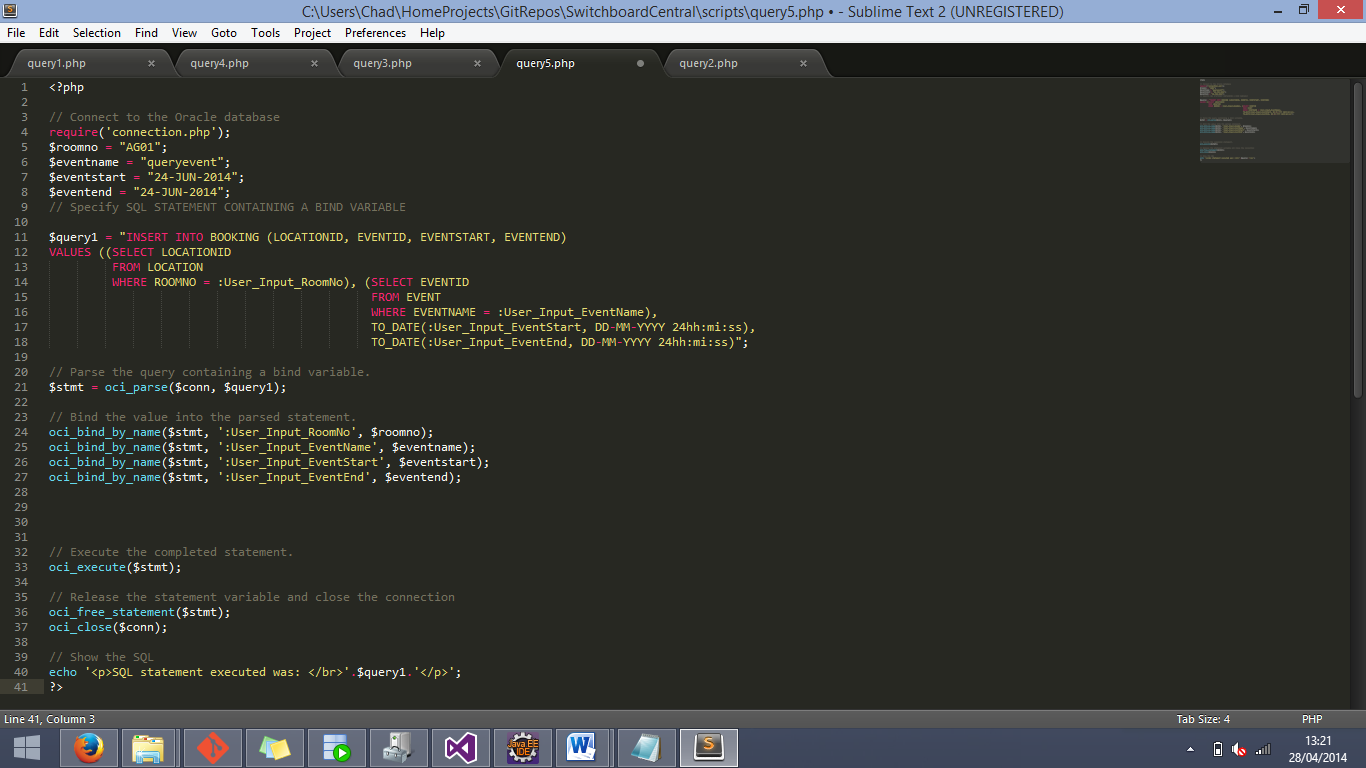
**Query 3 cont.**

The above image is of the third query, with the hardcoded data of “Devon”, “13-oct-04”, “10-nov-04”, and the number 3. The result of the query is as follows:

**Query 4**

The above image is of the fourth query, with the hardcoded data of “Dave Whitburn”, “queryevent”, and “queryeventpurpose”. As this is an insert statement there is no easy way to prove that it works as a web query (it does) via images.

**Query 5**



The above image shows the fifth and final SQL query, with the hardcoded data “AG01”, “queryevent”, “24-JUN-2014”, “24-JUN-2014”. Again, the results of this SQL statement cannot easily be shown via images so, we can only assure all those whom it concerns that it does indeed work.