SalesForce Integration with Web Server

Data Verity #2
Shane Kelley
Ebenezer Koranteng
Kevin Pham

June 16, 2017

Table of Contents

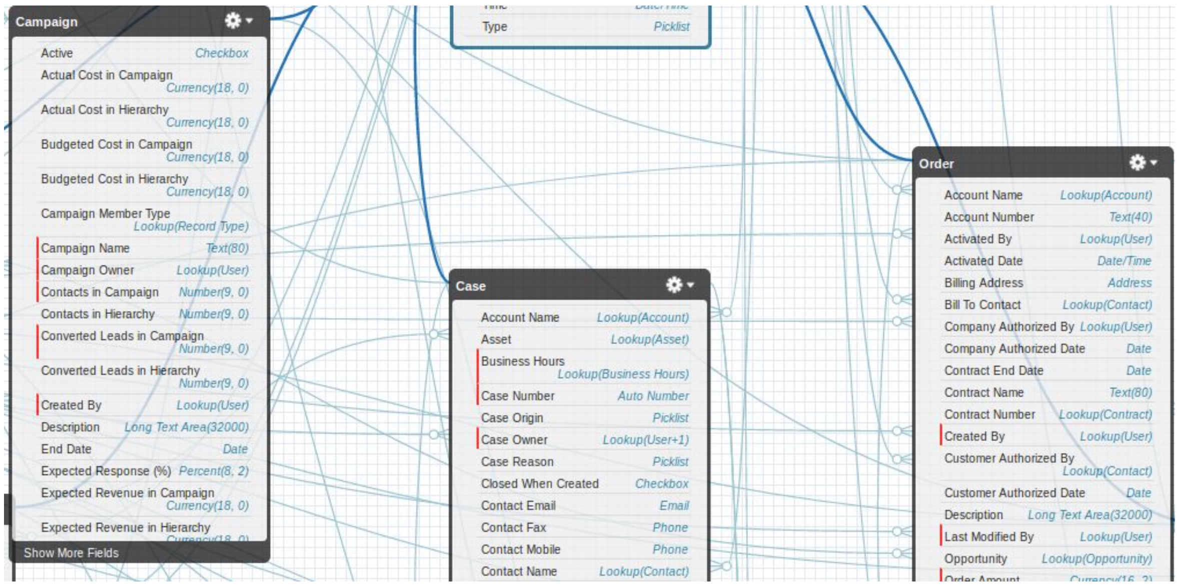
1. Introduction

1.1 Client Description

Our client, Data Verity, provides tools and software solutions to other companies that are customized to client specifications. One of Data Verity's best tools is a robust online MySQL database management and reporting tool. SalesForce.com currently holds the largest market share in customer relationship management (CRM) software usage, and has requested to integrate the Data Verity's tool into their software. SalesForce's software uses a proprietary database designed in an SQL-like language, called SOQL. Because of the differences between MySQL and SOQL, SalesForce's database is not compatible with Data Verity's database management and reporting tool. This incompatibility created the need for an integration tool to allow SalesForce's CRM software to take advantage of what Data Verity's database management and reporting tool has to offer.

1.2 Product Vision

To meet this need, the integration tool creates a copy of SalesForce's current schema into a separate, synchronized MySQL schema to be held on a Data Verity server. Synchronization between the two databases means that changes to one database will be accurately reflected in the other. This is significant because it reduces the risk of losing data for both SalesForce and Data Verity. The integration tool will allow Data Verity's existing database management and reporting tool to help SalesForce better serve their clients in the future by providing them with a more powerful software to visualize and manage their data. Figure 1 below displays 3 of the 331 objects in the SalesForce database. These objects are what was converted to Data Verity's database.



Campaign	
Active	Checkbox
Actual Cost in Campaign	Currency(18, 0)
Actual Cost in Hierarchy	Currency(18, 0)
Budgeted Cost in Campaign	Currency(18, 0)
Budgeted Cost in Hierarchy	Currency(18, 0)
Campaign Member Type	Lookup(Record Type)
Campaign Name	Text(80)
Campaign Owner	Lookup(User)
Contacts in Campaign	Number(9, 0)
Contacts in Hierarchy	Number(9, 0)
Converted Leads in Campaign	Number(9, 0)
Converted Leads in Hierarchy	Number(9, 0)
Created By	Lookup(User)
Description	Long Text Area(32000)
End Date	Date
Expected Response (%)	Percent(8, 2)
Expected Revenue in Campaign	Currency(18, 0)
Expected Revenue in Hierarchy	Currency(18, 0)

Show More Fields

Type	Picklist

Case	
Account Name	Lookup(Account)
Asset	Lookup(Asset)
Business Hours	Lookup(Business Hours)
Case Number	Auto Number
Case Origin	Picklist
Case Owner	Lookup(User+1)
Case Reason	Picklist
Closed When Created	Checkbox
Contact Email	Email
Contact Fax	Phone
Contact Mobile	Phone
Contact Name	Lookup(Contact)

Order	
Account Name	Lookup(Account)
Account Number	Text(40)
Activated By	Lookup(User)
Activated Date	Date/Time
Billing Address	Address
Bill To Contact	Lookup(Contact)
Company Authorized By	Lookup(User)
Company Authorized Date	Date
Contract End Date	Date
Contract Name	Text(80)
Contract Number	Lookup(Contract)
Created By	Lookup(User)
Customer Authorized By	Lookup(Contact)
Customer Authorized Date	Date
Description	Long Text Area(32000)
Last Modified By	Lookup(User)
Opportunity	Lookup(Opportunity)
Order Amount	Currency(16, 2)

Figure 1: SalesForce Schema Builder Tool

2. Requirements

2.1 Functional Requirements

- Convert SOQL data types, such as an integer, to a similar or the same MySQL data type.
- Convert objects from the SalesForce schema into MySQL tables. The columns of the table are known as object fields or properties.
- Create an extra MySQL table to store information about the tables. This reduces excessive queries to SalesForce's website.
- Create a method to compare when SalesForce's and Data Verity's tables were last updated.
- Create methods to migrate the information inside of the SalesForce schema into any number of MySQL tables.

2.2 Non-Functional Requirements

- The methods to both create the MySQL tables and migrate the data will be written using PHP.
- Do not break already existing implementations.
- Excessive compile times should be avoided.
- Maintain safe code standards determined by Data Verity.
- Correctly utilize SalesForce's REST API and use SOQL for queries into SalesForce's database.
- Use the same namespace for any classes made.

3. System Architecture

3.1 Web Server

The integration tool will be used to allow the web server and databases to interact with one another. Before the integration tool was implemented, the web server interacted directly with the SalesForce database. The integration tool communicates with both databases and ensures they accurately mirror one another. The red box in Figure 2 below encloses the main components that we were working with throughout this project.

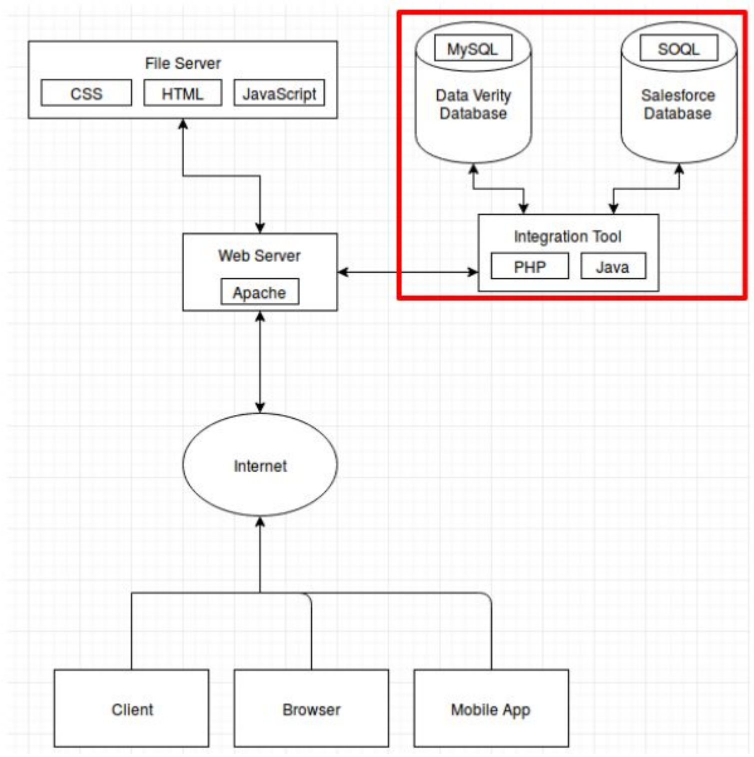


Figure 2: Overview of System Architecture

3.2 Integration Tool

The UML design in Figure 3 illustrates the connection between the classes that we had created and how we had used them to execute the task for our project.

The SObject class is designed to fetch an object from SalesForce so the team can migrate the object into Data Verity's server. To accomplish this, the team gained access to SalesForce using the Rest class. This allowed the team to obtain the object's description. The description contained the information needed to fetch the object's data. Within the SObject class is the syncObject method. This method is particularly important because it is designed to check the status of a SalesForce object, determining whether or not it should be resynced with the Data Verity server.

The main purpose of the Rest class is to get a SalesForce object's URL. The URL of the object enables access to SalesForce's database, which makes information retrieval possible. Because of this, the Rest class is used for every query or call to SalesForce's database.

The AllSObjects class grabs a list of all object names from the SalesForce database. As depicted in Figure 3, AllSObjects will have to access the SObject class to obtain information about the objects.

The ConvertSObject class is designed to create a MySQL table on the Data Verity server. It collects the name of an object as well as its fields in order to accomplish its goal. This class also accomplishes the task of determining the data types of an object's fields. The methods addMyPrimaryKey and myAutoIncrementing helped avoid duplicate records from migrating data over.

The DataFetcher class, as the name suggests, fetches data from SalesForce's tables. The data collected is stored for later migration to the Data Verity server. It also takes into account the last time the tables were updated. This will help when syncing the Data Verity server with SalesForce's server.

The PopulateMySQL class uses the information fetched using the DataFetcher class to populate Data Verity's tables. A timestamp column is also populated along side to enable comparison of the last sync time with system modified time during synchronization.

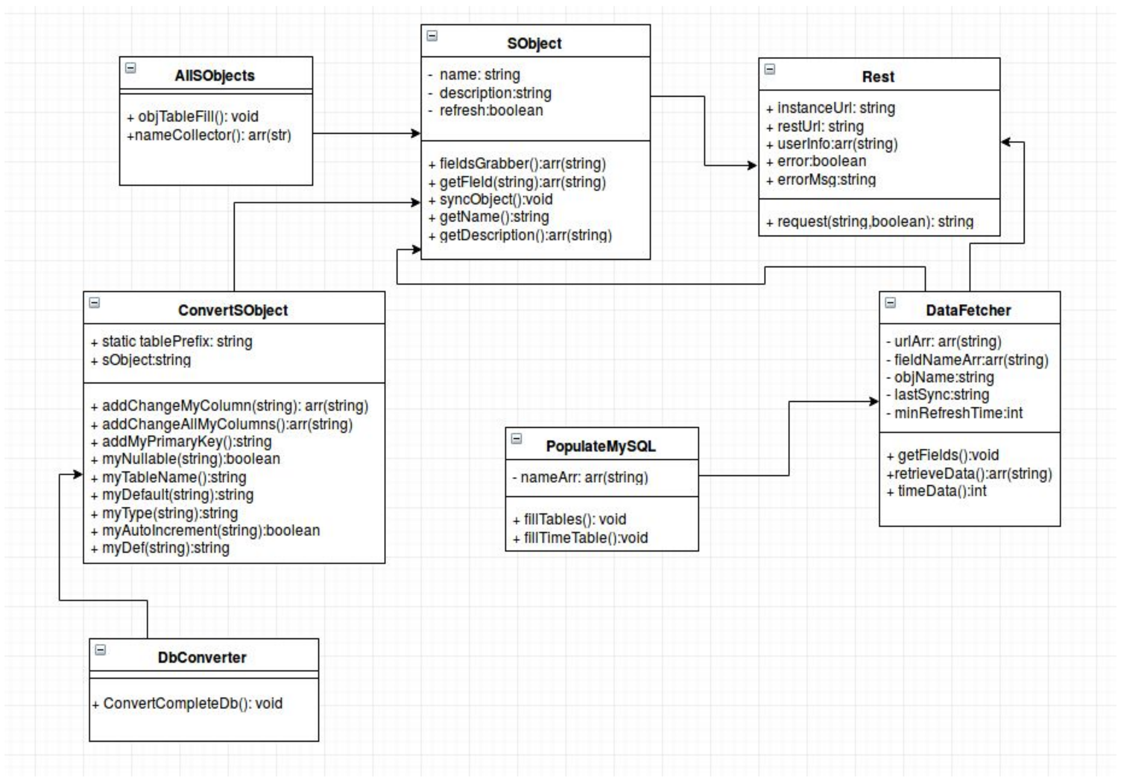


Figure 3: UML Diagram of the Integration Tool.

4. Technical Design

4.1 SOQL to MySQL Conversion

When copying data from one database to another, it is crucial to ensure no data is lost. This conversion proved to be a formidable task since SOQL has many data types that MySQL does not. Finding the most appropriate MySQL datatype to store the data was not apparent most of the time. Some of the data types that did not directly convert to MySQL includes id, description, and boolean. On the other hand, there were many data types that did conveniently convert well, such as DateTime, string, and int. Ideally, each data type would not waste any memory to store the information. For example, there is a url datatype in the SalesForce database. Knowing that urls have a maximum length of 2,000 characters helped determine what data type would be appropriate to convert it to. In MySQL, there are several text types, but the most relevant ones are TINYTEXT, TEXT, and MEDIUMTEXT. These data types hold up to 255, 65,535, and 16,777,215 characters respectively. In the case with the url data type, it logically made sense to use TEXT, since using TINYTEXT would potentially cause problem to arise when a url was greater than 255 characters in length. Also, TEXT was used instead of MEDIUMTEXT since urls can only be 2,000 characters long, TEXT would be sufficient enough to store a url even if it reached its maximum length. Using MEDIUMTEXT would require more space, and even if the difference in size between the two data types was only a few kilobytes, the impact of using one instead of the other in a large database with thousands or millions of rows would be enormous.

4.2 Avoiding Excessive Querying

One of the pitfalls for a project dealing with large databases is excessive calls and queries to a database. Excessive calls and queries, intentional or not, sparked the need for the team to come up with quicker and more structurally-sound solutions. For example, throughout the project, there were multiple instances where the sheer amount of queries and function calls led to excessive wait times. These instances are documented below:

- Instance 1: In the early phases of the project, the team would query an object one column at a time. To understand just how detrimental this is, the volume of the database will be described. Our largest object at the time, named “Account”, had 65 columns in it. With 330 total objects, most of which with similarly large amounts of columns, our program would take well over 8 minutes to complete. This was also before a custom object was added for testing purposes, which contains 3815 rows and 7 columns. An easy solution to this problem would be to use some sort of “SELECT ALL” functionality, one that all SQL-based languages have. Unfortunately, SOQL does not have any form or function that behaves similar to or completes a “SELECT ALL” style query. Because of this, a simple and custom solution was created. By obtaining all of the column names from an object, they can all be stuffed into one large SELECT query. To do this in PHP, the column names are stored in an array. This array is then iterated through, appending the column names to a SELECT query in each loop. This significantly lowered the amount of stress on the system and reduced the total wait time when querying data in an object.

- Instance 2: Our custom object of 3519 rows and 7 columns, aptly named “itsGonnaBeHUUUGE”, was designed with the intent of testing to see how large tables may create problems. Naturally, a table this large presented a major problem for the team, because of the fact that you cannot query more than 2000 rows at a time on SalesForce’s website. Thankfully, the solution is easily found on the internet, as other web developers have had questions about this issue before. When querying over 2000 rows, the query creates a field called "nextRecordsUrl". In PHP, it was easy to set a while loop that would execute as long as "nextRecordsUrl" existed.

- Instance 3: With all 331 objects queried, the next big problem was the fact that the program queried all 331 objects every time it was executed. After looking into some of the individual queries, the team found out that only about 50 of the objects actually had data inside of them. Because of this, a new MySQL table was created. This table, named "AllObjects", has 3 columns. The first column contains object names, the next column is a boolean that tells us if the object contains data or not, and the last column is a section of text containing column names for the objects. After populating this new table, the amount of tables to be queried was reduced drastically and sped our process up to about 1 minute.

- Instance 4: Only having around 50 tables queried every time was a great improvement, but importing tables with the same information as before is still inefficient. To streamline the process further, the team had to figure out a way of determining what tables do not need to be updated. After rigorous searching, the team found that the tables had a column called "SystemModstamp" within them. This column displayed the last time the table was updated in SalesForce's database, whether it was a manual or automated update. Using this information, the team created another MySQL table, named "Timestamps". This table has 3 columns, the first being the object name, the second being the last time the object was uploaded into Data Verity's tables, and the third being the last time the object was updated by SalesForce. With this last table, a very limited number of tables are updated. Because of the few updates, the wait time was reduced to just under 1 second.

5. Decisions

5.1 Programming Language

The top web development languages include Python, PHP, and JavaScript; however no team member was familiar with any of these languages. The languages that the team were familiar with were C++ and Java. Choosing a new language to learn for a web development project seemed like a formidable task, but would result in the team becoming familiar with another valuable language. On the other hand, using a language the team was already familiar with would allow the team to immediately begin to make progress towards the goal of building the integration tool. After initial discussions with the client, the team found out that the majority of the code used for Data Verity's existing tools is PHP. As a result, the team decided that learning PHP would be a good idea for the project. Once the team had a thorough understanding of PHP, the team used it for the entirety of the project.

5.2 Approach

There are two main approaches that were considered when creating the integration tool. The first option was to develop a database connector. This connector would query against the SalesForce database, similar to database connectors for other SQL variants, with a corresponding GUI. The second option was to create a synchronization tool, which will create an equivalent database structure with data in MySQL. The second option would be beneficial as Data Verity already has a GUI for MySQL. In conjunction with this fact, rewriting a GUI which is designed for MySQL to SOQL would be more difficult than copying the existing database to MySQL. For these reasons, the second option was chosen.

5.3 Object-oriented Class Creation

Each time a new task was introduced to the integration tool, a new class was created. Every new class was titled with a meaningful and easily understood name that gives developers a good idea of what the class does before opening the file. One particularly important decision during the beginning of the project was whether or not the team would have a single class to convert all SalesForce objects into MySQL tables. Doing so would eliminate the clutter of having two similar classes, but it was more advantageous to have two classes. Creating two separate classes allows the option to simply convert an item one at a time in the future while also supporting the ability to convert all objects in one run. Only having one class for conversion would eventually lead to wasting time if someone wanted only one object to be updated and had to wait on all objects every time.

6. Results

6.1 Features Implemented

There were three main tasks the integration tool needed to be able to fulfill to be considered complete. The tasks were to copy the SalesForce schema to a separate MySQL schema, push changes made from SalesForce to the MySQL schema, and push changes made from the MySQL schema to SalesForce. The integration tool at the time of this report has the capability to convert a database with hundreds of objects that can each have thousands of entries.

6.2 Features Not Implemented

The last main component of the integration tool to push changes from the MySQL tables on the Data Verity side to SalesForce's schema has not been implemented. The amount of time allotted for the team to work on the project proved to be insufficient to complete a robust implementation of this task. Instead of working on it with the potential for it to be incomplete by the time the course was over, the team instead worked on optimizing and further testing the existing implementations.

6.3 Performance Testing Results

Throughout the course of working on this project, the team had to constantly consider the length of time their implementations would take to run. There were several times during the team's sprints where runtimes were lasting tens of minutes. Ultimately, the goal was to keep the team's runtimes minimized. After the functionality of a task was achieved, the team would review the process and make changes to optimize the runtime if possible. By the end of the last sprint, all tasks that were implemented had acceptable runtimes for the client.

6.4 Summary of Testing

Queries against the SalesForce schema could only return 2,000 rows at a time. To ensure the integration tool could convert objects with over 2,000 rows, an object with 3,519 rows was created in order to test whether or not the tool would correctly handle the object. Another test was created to ensure all objects in the SalesForce schema were successfully copied to a MySQL schema. Lastly, testing was done to confirm whether or not the comparison of timestamps between when an entry was last modified and when the databases were last synced was correct. This last test is crucial to the integration tool. This is because it is vital that the two schemas are always exact copies of one another so that the reporting tool can provide accurate reports of the information stored in the database.

6.5 Future Work

In order for the integration tool to be complete, a third component must be added. This last component is to push changes in the MySQL tables to the SalesForce schema. After that component is complete, further development can be made to expand the tools that can be used to analyze the data in the SalesForce database.

6.6 Lessons Learned

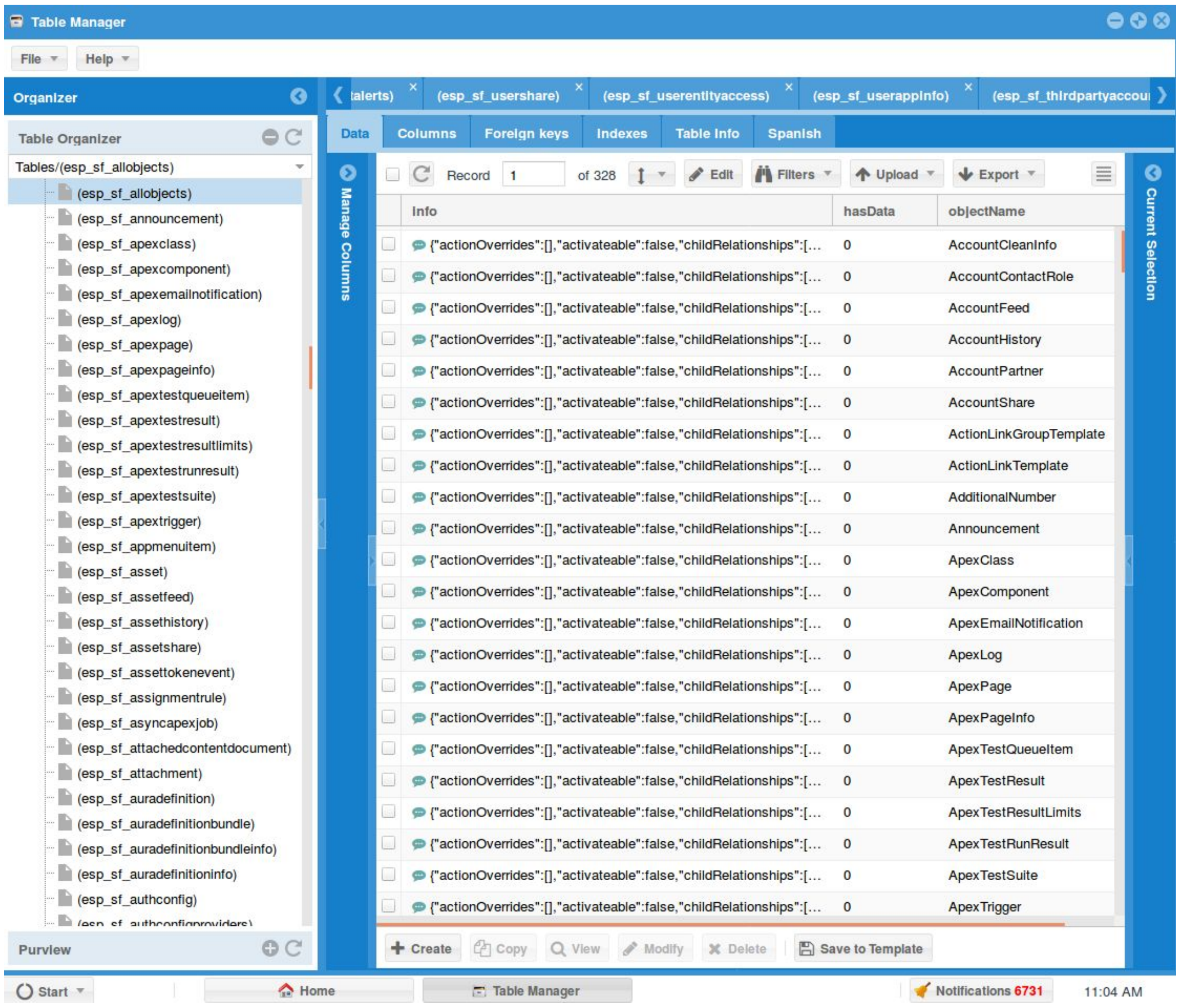
Throughout the course of the project, there were countless opportunities where the team was stumped because the problems that spawned were unfamiliar to team members. Each of these encounters was an opportunity for the team to learn something new. Some of the lessons that were learned are listed below:

- Creating a cache of objects on Data Verity server prevents the need to query SalesForce multiple times and drastically improves efficiency.
- Object-oriented PHP is easy to implement and feels a lot like C++ and Java. This helped vastly, especially when expanding the integration tool.
- It is difficult to develop foolproof software that caters to every need, both in the present and future.
- Learned the importance of having a system administrator available at all times since server can be overloaded quickly with some simple errors, particularly infinite loops.

Appendices

A1. Table Manager

Table Manager is one of the tools created by Data Verity. This tool allows the user to browse through the tables located on the database.



(esp_sf_allobjects)
(esp_sf_announcement)
(esp_sf_apexclass)
(esp_sf_apexcomponent)
(esp_sf_apexemailnotification)
(esp_sf_apexlog)
(esp_sf_apexpage)
(esp_sf_apexpageinfo)
(esp_sf_apextestqueueitem)
(esp_sf_apextestresult)
(esp_sf_apextestresultlimits)
(esp_sf_apextestrunresult)
(esp_sf_apextestsuite)
(esp_sf_apextrigger)
(esp_sf_appmenuitem)
(esp_sf_asset)
(esp_sf_assetfeed)
(esp_sf_assethistory)
(esp_sf_assetshare)
(esp_sf_assettokenevent)
(esp_sf_assignmentrule)
(esp_sf_asyncapexjob)
(esp_sf_attachedcontentdocument)
(esp_sf_attachment)
(esp_sf_auradefinition)
(esp_sf_auradefinitionbundle)
(esp_sf_auradefinitionbundleinfo)
(esp_sf_auradefinitioninfo)
(esp_sf_authconfig)

Info	hasData	objectName
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	AccountCleanInfo
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	AccountContactRole
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	AccountFeed
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	AccountHistory
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	AccountPartner
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	AccountShare
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ActionLinkGroupTemplate
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ActionLinkTemplate
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	AdditionalNumber
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	Announcement
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexClass
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexComponent
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexEmailNotification
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexLog
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexPage
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexPageInfo
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexTestQueueItem
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexTestResult
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexTestResultLimits
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexTestRunResult
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexTestSuite
{"actionOverrides":[],"activateable":false,"childRelationships":[…	0	ApexTrigger