Salesforce Jar Publisher

CSCI370
Summer I 2017

Adrien Perinet
Cody Watters
Garrett Daly

I. Introduction

A. Client

Our client, Salesforce is the largest Customer Relationship Management (CRM) provider in the world. Salesforce software is cloud-based, allowing easy data management and set up regardless of size; as such, its customers include businesses and partners of all types and industries. Salesforce creates better interaction with customers through fast access to data and by striving to understand customer preferences, predicting customer needs, and facilitating mobile business management. As one of the top 50 traffic driving websites in the world, Salesforce consistently helps lead the way in the creation of next generation technologies to help businesses grow.

B. Product Vision

This project is meant to ease the use of the SOAP (Simple Object Access Protocol) interface for both customers and developers. Currently, Salesforce APIs (Application Program Interfaces) can be implemented with either REST (Representational State Transfer) or SOAP interfaces. The vast majority of consumers use the REST interface because it is both easier and more standardized; however, there are a handful of customers that still prefer to implement Salesforce APIs using SOAP. To use the SOAP interface, Salesforce provides a WSDL (Web Services Description Language) file that can be compiled into a jar file. The jar file can be used as a dependency in a client's code base.

The purpose of this project is to reduce the barrier between a client and using the SOAP interface. The customer currently uses SOAP to log into their Salesforce Org, download a WSDL, compile a jar with a Salesforce compiler, and push the jar into their code base as a dependency. With this product, the customer will have the ability to automatically do any or all of these steps via a command line tool. In addition, the jar will be automatically pushed to the Maven Central Repository in order to make the dependency more easily accessible. The only component that will remain manual for the user is setting up their Maven Central settings, which should be different for each user.

II. Requirements

A. Functional Requirements

The Jar Publisher is primarily an API tool for Salesforce customers. Therefore it facilitates the technologies that customers wish to use in the following way:

1. Many customers access Salesforce features using SOAP interface instead of more common RESTful interfaces. As such the Jar Publisher is capable of interpreting requests to handle customer needs for the SOAP interface.

2. In response to a customer request, the Jar Publisher produces a WSDL file that describes the API the customers require.

    a. This WSDL file is used to generate a .jar file containing appropriate code corresponding to the WSDL. This .jar file pushed to a public repository for customers to access via Maven dependencies.

    b. The WSDL file is also used to generate java stubs, which are basically the decompiled java functions from the developed .jar file.

3. The Jar Publisher provides basic authentication functionality, such that the identity of a customer is verified before services are provided.

4. The Jar Publisher executes in the background. That is, the Jar Publisher should have no GUI; it should execute only in response to customer calls.

B. Non-Functional Requirements

1. Programming Languages

    a. The main component can be done in any programming language that we like, although Salesforce primarily operates with Java, Bash, with some teams using Ruby and Scala. For this project deliberated between using Java, Bash, or Python.

    b. Python, Maven, Bash, XML

2. Standalone

    a. This project exists in a public repository, and should be standalone. It should not interact with Salesforce code directly, but instead uses their APIs.

3. Source control - Git via GitHub

4. Public Repository - Nexus / Maven Central

C. Potential Risks

Working with code that will directly be hitting client's environments posed a few obvious risks including any bugs that would open up their software to vulnerabilities. However, the risks in this project involved interacting with Salesforce owned environments. Testing this product was tricky because any production level test requires being pushed to the official repository. Using actual username and passwords was relatively easy, but we were limited in the number of IP addresses we could test from. In addition, because the scope of the project involved making a standalone application that can be used for different APIs in the future, the deliverable had to be robust enough to properly and dynamically handle a range of dependencies and functionalities. To overcome some of the potential risks, we structured the program incrementally as a command-line-tool. Every step of the tool requires authentication, and without a session key requires a username and password.

D. Definition of Done

Salesforce allows clients to use both SOAP and REST to access their multitude of APIs. The process for using SOAP is currently incomplete and fairly undocumented. Salesforce only provides a WSDL file for their clients to compile and incorporate into their code; however, their client is then responsible for being able to compile this to a jar file that can be used in the dependencies of their application. The scope of the project is to dynamically generate a WSDL file and then create a jar that is accessible to the client. In this sense, the project is done when a client can checkout an automatically generated dependency file from a Salesforce owned repository.

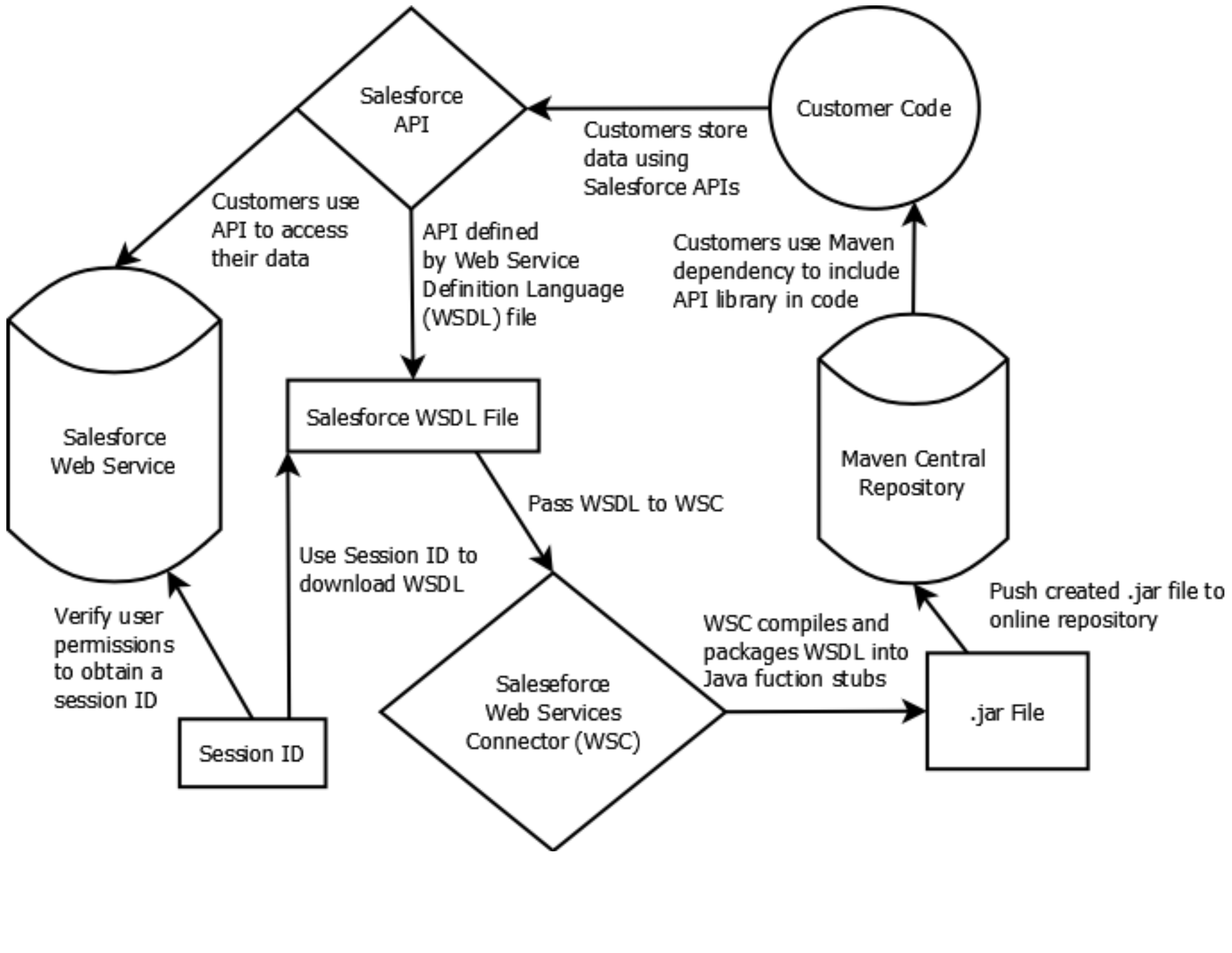
III. System Architecture

The Salesforce Jar Publisher consists of a Command Line Interface (CLI). CLIs are used to allow programmers to access specific functions in a program one at a time. The Jar Publisher has multiple possible access points, as it can be used for a variety of purposes depending on the user's needs. Standard usage involves a strict execution sequence, but it should be noted that the user can exit wherever they please (see Figure 2).

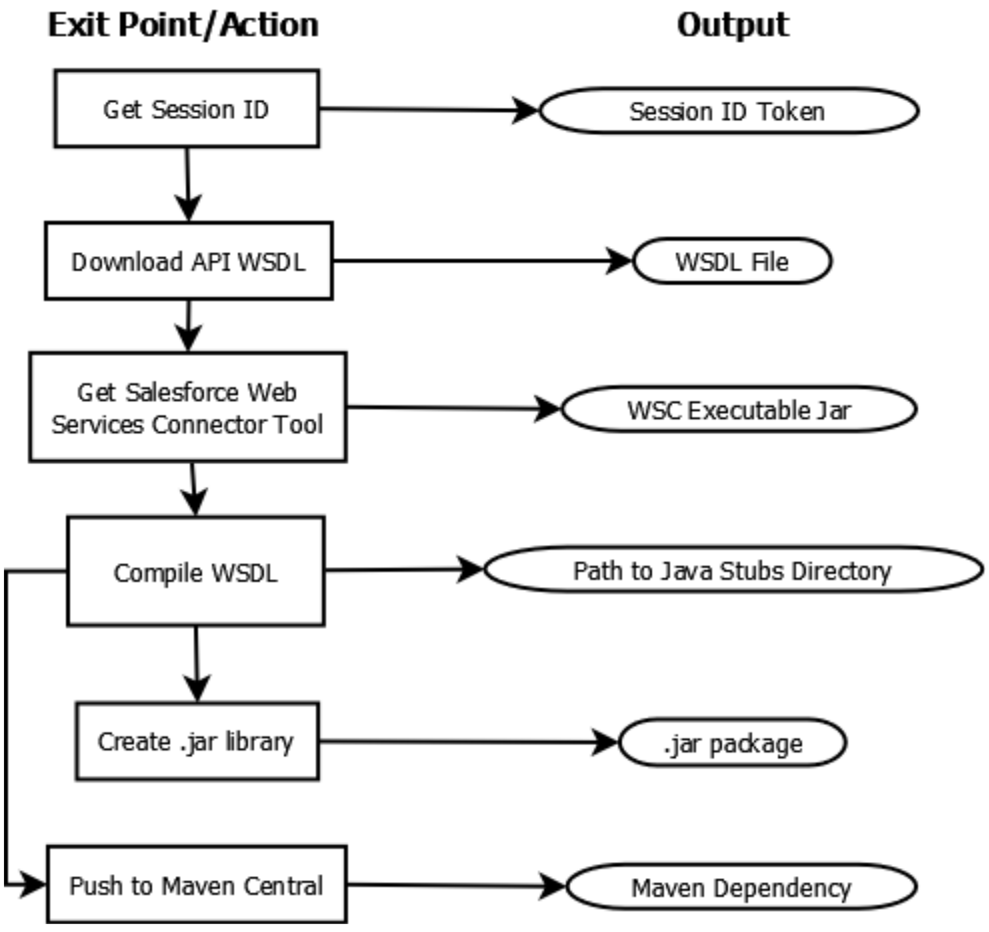
Execution flow is described in Figure 1 and consists of the following: The user must first authenticate their identity with the Salesforce web service (in order to verify which APIs they have permission to use) by providing their Salesforce credentials. This yields a Session ID, which is then used to download a Web Service Definition Language (WSDL) file from Salesforce, which describes the functionality of the specified API. This WSDL is subsequently interpreted by the Salesforce Web Services Connector (WSC) tool, which compiles and packages the WSDL into Java function stubs within a .jar package. These stubs can then be used directly by the user, or they can be pushed to the Maven Central Repository. The latter allows programmers to automate the inclusion of such stubs using the Apache Maven tool.

As stated above, the user can exit at any of the above described points. For example, if they only wish to obtain a WSDL file, they can opt to end execution before pushing to Maven Central. Alternatively, they might wish to run every step. This is demonstrated in Figure 2.

Components include the following:


Salesforce API
Customer Code
Customers store data using Salesforce APIs
Customers use API to access their data
API defined by Web Service Definition Language (WSDL) file
Customers use Maven dependency to include API library in code
Salesforce Web Service
Salesforce WSDL File
Maven Central Repository
Pass WSDL to WSC
Use Session ID to download WSDL
Verify user permissions to obtain a session ID
Push created .jar file to online repository
WSC compiles and packages WSDL into Java fuction stubs
Saleseforce Web Services Connector (WSC)
.jar File
Session ID

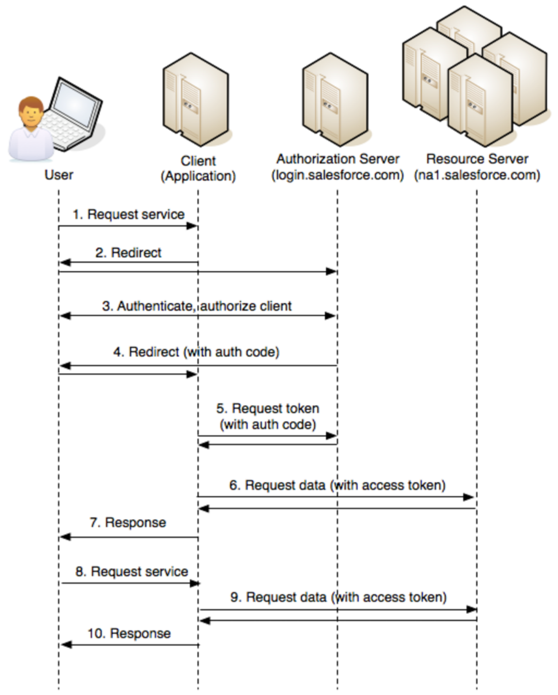
- *Salesforce Web Service:* black box representing the various data management services offered by Salesforce
- *Session ID*: used as a token to authenticate users
- *WSDL file*: Web Service Definition Language file; XML document that describes the functionality of an API.
- *Salesforce WSC Tool:* Web Services Connector tool; used to compile Salesforce WSDL files into Java function stubs.
- *Apache Maven:* Project building tool; used to automate the inclusion of 3rd party libraries.
- *Maven Central Repository:* Online repository that stores code for usage in Maven projects.



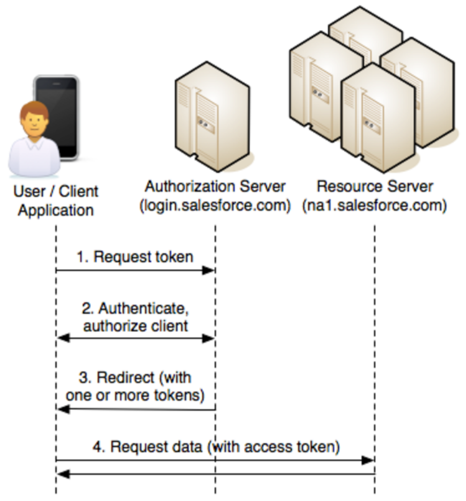
IV. Technical Design

A. Obtaining a Session ID

Initially, there was not a quick and easy way to obtain a session ID from Salesforce using only a username and password. The early cURLs (Client URL Request Library) would need to use a client_id and client_secret key [1]. These required developers to log onto the Salesforce website and get their keys. These keys were based on what company and domain a Salesforce client was using, and required some searching within the Salesforce website. This slowed the process and made automating login procedures nearly impossible.


User
Client (Application)
Authorization Server (login.salesforce.com)
Resource Server (na1.salesforce.com)
1. Request service
2. Redirect
3. Authenticate, authorize client
4. Redirect (with auth code)
5. Request token (with auth code)
6. Request data (with access token)
7. Response
8. Request service
9. Request data (with access token)
10. Response

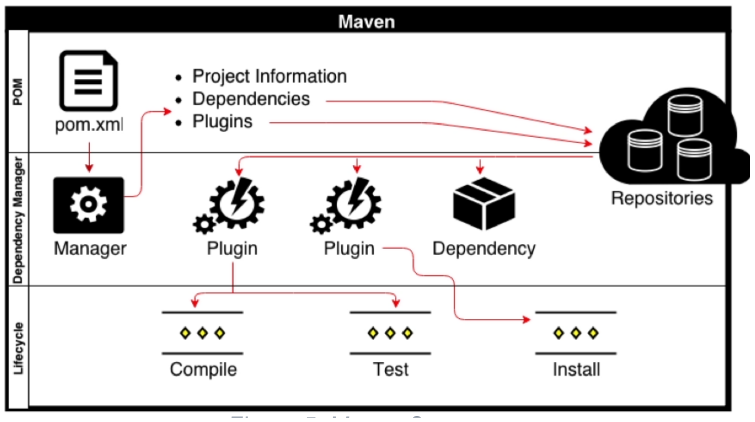
Using a Web Server Flow was another option, as it would grant a session ID that could be used to download from Salesforce servers. The Web Server Flow was not a viable solution, because it too required using the client id. This would take information from the client to the authorization server, then pass information to and from the resource server. This would then generate an access token, and give information back to the client. The decision was made that using simply a username password combination was the best choice for gaining a session ID, because we wanted direct communication between the client and resource, without having to wait on the resource server. The Web Server Flow is demonstrated in Figure 3.


User / Client Application
Authorization Server (login.salesforce.com)
Resource Server (na1.salesforce.com)
1. Request token
2. Authenticate, authorize client
3. Redirect (with one or more tokens)
4. Request data (with access token)

After looking through an older discussion page, there was an example of a SOAP request that returned a session ID using one of the earlier versions of the API. This took in two fields, simply a username and password, and returned a SOAP response that contained a large amount of information, including a session ID. After parsing the session ID, cURLs become much easier to use. This allowed for much of the automation to take place, and will allow for ease of use later on. Simply using the same session ID will allow developers to work faster. An example of the User-Agent flow is shown in Figure 4.

B. Pushing to Maven Central

The Salesforce Jar Publisher relies heavily on Apache Maven, a tool designed to automate project builds. One of the primary functions of Maven is the inclusion of third party libraries. Such libraries are name “dependencies”, and are declared inside an XML document named “pom.xml”, which describes the build characteristics of a Maven project (see Figure 5) [3]. For example, if a user wishes to use a 3D graphics library, they need only include the dependency in their pom file, allowing Maven to automatically locate and download the necessary code at compile time. Furthermore, Maven can be used to push a user’s code to an online repository named Maven Central, which allows *other* programmers to easily incorporate the user’s code.



This has several significant implications for the Salesforce Jar Publisher. First, Maven Central’s ability to easily distribute code makes it the ideal platform for hosting Salesforce code, as it allows customers to easily find and incorporate the Java code necessary to access their data. Second, Maven itself can be used to automate pushing code to Maven Central, making it a powerful tool for uploading dynamically generated Salesforce code (such as the code generated from WSDL files). As such, Maven consists of an essential utility for the Salesforce Jar Publisher.

Maven Central has very strict requirements in order to prevent the distribution of malicious code. Such requirements include declaring an owned “group ID” (typically this follows package naming conventions such as “edu.mines.csci370”) and a dedicated GitHub repository. While such requirements are necessary and ultimately beneficial, this caused significant difficulties due to the more dynamic nature of our project. For example, we briefly considered automating the creation of GitHub repositories for each API. This proved both excessively wasteful and difficult to implement; as such we chose to create a single “dummy” repository to host each release. In order to prevent merge issues and other such conflicts, the repository contains no code, and simply exists in order to satisfy the Maven Central requirements. Additionally, we had to implement code to dynamically create a pom file based off the characteristics of the code being pushed.

Pushing to Maven Central required a myriad of file management actions such as copying and moving the files created during earlier steps in the Salesforce Jar Publisher’s execution. It became clear that the steps required to deploy to Maven Central were better accomplished using a Bash script, rather than the Python we used for the remained of the project. As such, we were forced to shift the Maven deployment lifecycle over to a separate, standalone script. Admittedly this recourse was not ideal, but given our limited timeframe, we were forced to settle with such a solution.

V. Decisions

A. Language

	Versatility	Redability	Libraries	Error Handling	Use at Salesforce	Total Score
Bash	3	2	3	3	2	13
Python	1	1	1	2	3	8
Java	2	3	2	1	1	9

We were given complete freedom in selecting a language to use for the Salesforce Jar Publisher; as such we had a variety of options to choose from. Knowing a bit about how the project was going to develop going forward, the first step was narrowing down the language options to one that would work well with Salesforce and the structure of the project. The language needed to be one that was relatively quick at processing and executing commands on a machine as well as able to handle various errors that might arise when sending HTTP requests. The languages that were the best for this team's skillset as well as the problem are Bash, Python, and Java.

The decision ultimately fit well into five categories. The first, versatility, is fairly straightforward given the three options. Bash doesn't have a lot of options other than executing a script style set of instructions without putting some serious time into making a complex application. Java is versatile but can be tedious when class structure isn't a top priority. This left Python as the best option. From a readability standpoint, Python was again the best decision as the script could be ran through a linter to match common Python standards. The third category is no surprise as Python takes pride in a community of library architects that allow for versatile use cases. Python, however, is not the best of these three at error handling, where Java is easily the better option. In addition, Java has the final upside due to being the primary language at Salesforce. The decision matrix above shows that, while Java is used more frequently at Salesforce and would be a decent language for the job, Python has a slight advantage. Given these factors, the project took roots in Bash for proof of concept scripts and evolved to Python shortly after.

B. WSDL Compilation Tool

Web Service Description Language (WSDL) files are XML documents used to define the various API functions offered by a website or web service. The information provided in a WSDL can be used to automatically generate function stubs (e.g. C++ prototypes, Java method stubs) that allow users to incorporate API function calls in their code, regardless of language. The process of converting a WSDL into stubs is known as *WSDL compilation*, and consists of a primary step in the Salesforce Jar Publisher's lifecycle. Several WSDL compilation tools exist; those most appropriate to our project are Apache Axis (the most popular solution) and the Salesforce WSC tool.

The following table illustrates the advantages and disadvantages of both tools.

Salesforce WSC (Web Services Connector)	Apache Axis (Apache Extensible Interaction System)	Both
● Used primarily to expose Salesforce APIs. ● Owned by Salesforce (no license required) ● Used primarily for Java applications ● Can use both strictly (Enterprise) and loosely (Partner) typed WSDLs ● Uses Metadata APIs for managing sandbox "customizations"	● Generalized tool; used for a variety of purposes and APIs ● Owned by Apache (requires license) ● Used primarily for C++ and Java applications ● Uses strictly typed WSDLs	● Code generation tool used to compile client side code from **WSDLs**. ● Uses the SOAP API for managing data ● Written in Java ● Used to facilitate web service deployment ● Used to simplify client side code generation

The Salesforce Jar Publisher team elected to use the Salesforce WSC tool for two key reasons. First, the WSC tool is owned by Salesforce, such that its incorporation into a Salesforce project would require no additional juggling of licenses. This greatly reduces administrative overhead, allowing the team to focus more on development and less on potential legal issues. Second, the WSC is specifically designed for use with *Salesforce* WSDLs and APIs, a fact that also holds true for the Salesforce Jar Publisher. The WSC tool is constantly maintained in order to guarantee proper compilation of Salesforce WSDLs; therefore the tool is almost certain to function properly, a luxury that is not available when using Apache Axis. As such, the Salesforce WSC tool is the clear choice for compiling WSDLs.

C. Interface

Initial Possible Options:

- Desktop Application
- Singular Script / Program
- Web Application
- CLI / CLT(Command Line Interface / Command Line Tool)

We initially did not know how the final outcome of our project would operate. There needed to be a simple way for a developer to login to salesforce and quickly download dependencies for a given WSDL. For this reason, we considered the four possible options listed above.

Both the desktop application and web application were great options in terms of usability. A web application would have far greater access than a desktop application, because there would be no need to download an application each time a user wanted to run the program; however, a web application would need to integrate with the Salesforce site, of which we have limited access. For a project of this magnitude, there seemed to be no need to have a full-blown GUI (Graphic User Interface) because of the limited options. Creating a desktop or web application seemed unnecessary, and seemed to limit the scope and growth of where the project could expand.

We then decided to run the project as something directly from the command line. We initially went with a single script in Bash to get everything up and running, and then re-wrote the script in Python to allow for ease of testing / error handling. Having a single script limited the project somewhat, as each time a developer accessed the script they would perform a single action and gain a single dependency. For flexibility, we decided to modify our tool to become a CLI. This way, a developer could use the CLI to login, gain a session key, gain a WSDL, generate stubs for a WSDL, or gain dependencies all separately. This way if a developer needed to perform only one action they would be able to do so easily. Creating a CLI also allows for easier changes down the road, as single actions would need to be swapped in and out as Salesforce code changes.

VI. Results

A. Results

The group successfully met each of the requirements set forth by our Salesforce contact. There are no features that we failed to implement, such that our product represents a complete, standalone tool that will greatly facilitate the incorporation of Salesforce SOAP APIs into customer projects. Despite this, there remain areas to improve if future development occurs. For example, the current iteration of the Jar Publisher functions only as a Command Line Interface (CLI); future versions could benefit from a lightweight GUI to facilitate the tool's usage for programmers less familiar with the command line environment. Furthermore, the tool is geared primarily for Salesforce engineers, despite the fact that many of its features (such as WSDL retrieval) could greatly benefit Salesforce customers. As such, it may be beneficial to create a public version of the tool specifically for non-Salesforce use.

During development, we maintained an extensive set of tests to help guarantee the overall functionality of the Jar Publisher. Certain tests, however, were slightly hindered by the fact that we lacked full access to certain Salesforce resources. For example, when deploying .jars to Maven Central, we were unable to push Salesforce code due to permission issues. We were forced to test the code's functionality indirectly by creating dummy files instead of the true Salesforce files that the tool will eventually use.

B. Lessons Learned

This project presented a variety of important lessons and problems that we are certain to continue to face throughout our careers. For example, the permissions example above helped illustrate a problem frequently faced by external contractors; often times they must work around their client's policy in order to address hurdles that might not otherwise exist. Furthermore, as a group we discovered that an overwhelming majority of our time was spent researching and learning rather than writing code. This embodies the fact that a software engineer must constantly expand their knowledge base, a lesson that we a likely to face with every new job and with every paradigm shift in computer science.

In addition, our team met each day for a daily scrum. In these meetings, we learned the value of discussing what each member was working on. On multiple occasions, these meetings would spawn a conversation that ended up solving a problem causing a block. Additionally, the group met with our client an average of once every four days to discuss current, new, and old stories in each of our sprints. This helped facilitate progress on the project and kept the project focused as it moved through multiple different iterations that had significant differences. As mentioned, the team worked consistently through sprints which consisted of 2-4 stories. At Salesforce, these stories are labeled "spikes". Each spike broke up a large idea of the project direction into executables. In terms of the magnitude of the spikes, each one could be classified as a 3 on the Fibonacci scale where a trivial task and 8 is usually the largest task in a sprint.

The agile lifecycle is supported by many third party applications. These applications are used to track progress on small and large scale pieces of software. Amongst these are recognizable names such as Jira by Atlassian and Pivotal Tracker. For a project of this scale, we chose to go with a smaller kanban board hosted via the web application Trello. Trello offers light scale cards that can be moved from one section of a kanban board to another column. This is useful for tracking the lifecycle of a particular spike. In our case, we tracked New, In Progress, and Completed spikes. Each card containing a spike can be assigned to a member and commented on as well as updated. For burndowns, we manually tracked the completion of spike points and ended up getting a fairly consistent sprint velocity. In order to facilitate communication in an easier and more accessible way than email, we set up a Slack with our client. Slack is a business tool used for intra-company communication. This allowed for quick questions between the client and ourselves.

VII. Appendices

[1] Patterson, Pat. (August 2016) "Digging Deeper into OAuth 2.0 on Force.com" https://developer.salesforce.com/page/Digging_Deeper_into_OAuth_2.0_on_Force.com

[2] Multiple Authors. "Obtaining Access Token using cURL? (October, 2010) "https://developer.salesforce.com/forums/?id=906F00000009CYaIAM

[3] Apache Software Foundation. “Project Summary” (November 2015) “http://maven.apache.org/ref/3.3.9/project-summary.html”