Symplified RESTful Web Services

CSCI 370 Field Session - Summer 2012

Ryan Beverly
Alex Broom
Travis Bybee
Sampson Miller

Symplified is a cloud based security company which offers its customers a simple way to integrate enterprise networks with the cloud in a secure fashion. In addition, they offer various tools for the customer including access management and auditing. These services are used in preventing “side-door” access to apps, single-sign on, and central authentication of users.

Customers choose Symplified as a way to manage their employees’ access to both external and internal cloud services. By routing all logins through the Identity Router, employees are only responsible for one login. Managing all of the other logins through the Identity Router allows the customer to easily add and remove cloud services from individual employees and ensures that if an employee of theirs leaves, that employee will not have access to the accounts the customer has set up for them.

Symplified’s system is composed of two main parts, an Identity Router and an Administration Server. The Identity Router is a tool meant to provide access management for certain users, such as employees at a company. This access can be strictly controlled by company administrators. Symplified's proxy based architecture prevents “side door access”. “Side door access” is a method of gaining access to a computer or protected application by bypassing normal authentication while attempting to remain undetected. Users have to access applications through a company ribbon or portal, which prevents unauthorized connections. As a means to provide this access management, Symplified’s router will request credential validation from a user store. There are several types of user stores, each with their own set of properties, but they are essentially databases with sets of information, typically containing usernames and passwords for authenticating application access. When a user attempts to log in to a protected web application the user store is accessed, establishes authorization and connects the user. This user store information is never replicated by the router, as this would only create more information to manage.

The Administration Server is the centralized administrative interface where administrators can set user authorizations for web applications and create a connection with the administrators’ clients’ user stores. The Administration Server manages access control from user store to applications. Fields on the server are set to control everything that is accessible by an employee.

Our product is an application to connect to REST web services, written in four different programming languages: Ruby, PHP, Java, and .NET. Each final application demonstrates to Symplified's customers how to send HyperText Transfer Protocol(HTTP) requests to their Identity Router and Administration Server. In addition, each program also has a command line interface (CLI) or a graphical user interface (GUI) to allow the customers to quickly see how each request works. The purpose of having very similar applications but in four different languages is to give customers the option of understanding how requests work in the different languages and providing customers a base code to work from in their preferred language.

REQUIREMENTS

The requirements of the client were as follows. For each of the four languages (Ruby, PHP, .NET and Java) the application needs the ability to send a GET, DELETE, POST, or PUT

HTTP request to the servers. In order for the requests to be properly authenticated, each request requires four HTML headers: accept, content-type, Authorization, and Date. The Authorization header contains a digital signature which uses specific methods of encoding and hashing so the server will properly authenticate the request. These requests use the indicated resource on the server in different ways depending on what request is sent. An example of a resource would be a UserProfile or an Application which exist as data on the server.

Of the server commands the most basic of these is the HTTP GET request. This request sends a query to the server, which responds by returning the resource information in either Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format. This is the simplest request required. The DELETE request is similar to the GET request and also requires minimal effort to implement. This request resembles the GET request in all but functionality. The DELETE request removes the indicated resource from the server.

The POST and PUT requests require additional steps to be taken while creating the signature and headers. The POST method provides the server with an XML or JSON string, and the server adds the resource to the server's resource database. The PUT method also gives the server a string and an explicit path. Instead of adding a new resource to the server, PUT modifies an existing resource according to the information sent with the request.

The final functional requirement we were given was to use the HTTP request functionality to create either a CLI or a GUI; this was left open to our creativity and varied from language to language. We implemented CLIs first and then GUIs and web pages for our applications.

Outside of the functional requirements for the product, there are also three necessary features each application needed. Each language requires clear documentation for installation. The main reason for creating these applications is to provide example base code for Symplified's customers. In order for their customers to effectively use the base code, installation needs to be straightforward and recreatable. The clear instructions allow customers to quickly set up and run the base code, allowing their teams to quickly develop their own code.

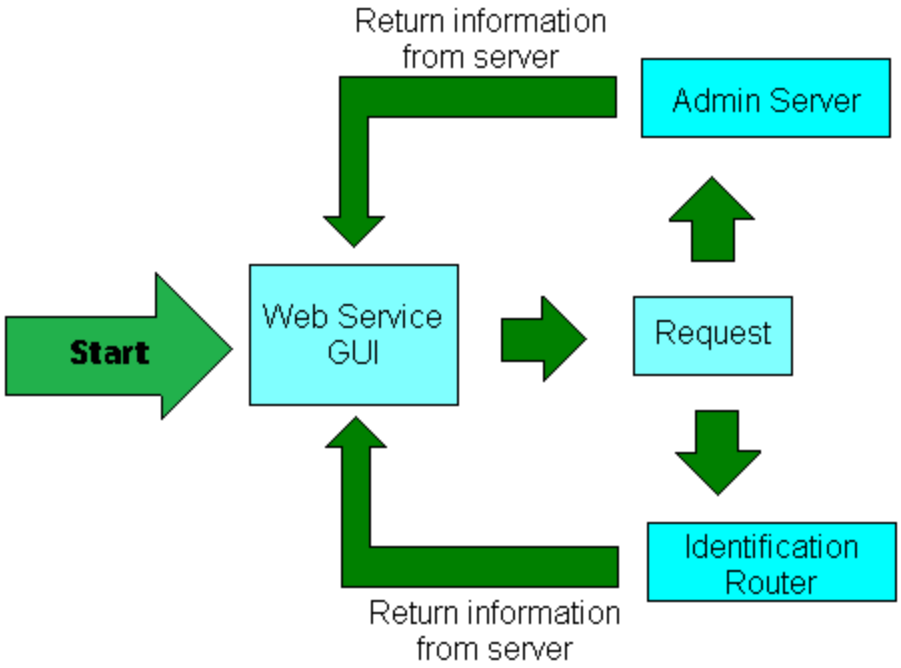
Secondly, the code must be easily readable. This involves the use of multiple functions and ample, clear comments. Even if the customers could quickly install the base code, it would be useless if they did not understand how the code works and why aspects are implemented the way they are. The use of functions and comments provides additional instruction on how the entire code comes together and prevents frustration when examining the code.

Finally, the output from the server in the user interfaces must be simple and show a meaningful view of the data. When viewing the demos, customers want to understand the data output once parsed. By formatting the parsed output, it shows to customers that data is quickly grabbed from the servers and can be easily formatted for human readability.

ARCHITECTURE

A general representation of how the program will execute in each of the four languages is shown in Figure 1.

Figure 1 - Blueprint:



The user will be presented with an interface and have the ability to send requests to either the Administration Server or the Identity Router. This request will go to the server or the router depending on what the request was, and if the request is valid, content will be returned to the user, along with an HTTP response code. If the request is not valid, an HTTP error code will be returned with a description of what failed and the reason why.

The Identity Router and the Administration Server are the two primary systems used in the Symplified product. These interact with each other to provide the security and authorization for users connecting to both internal and external cloud services used by the company.

TECHNICAL DESIGN

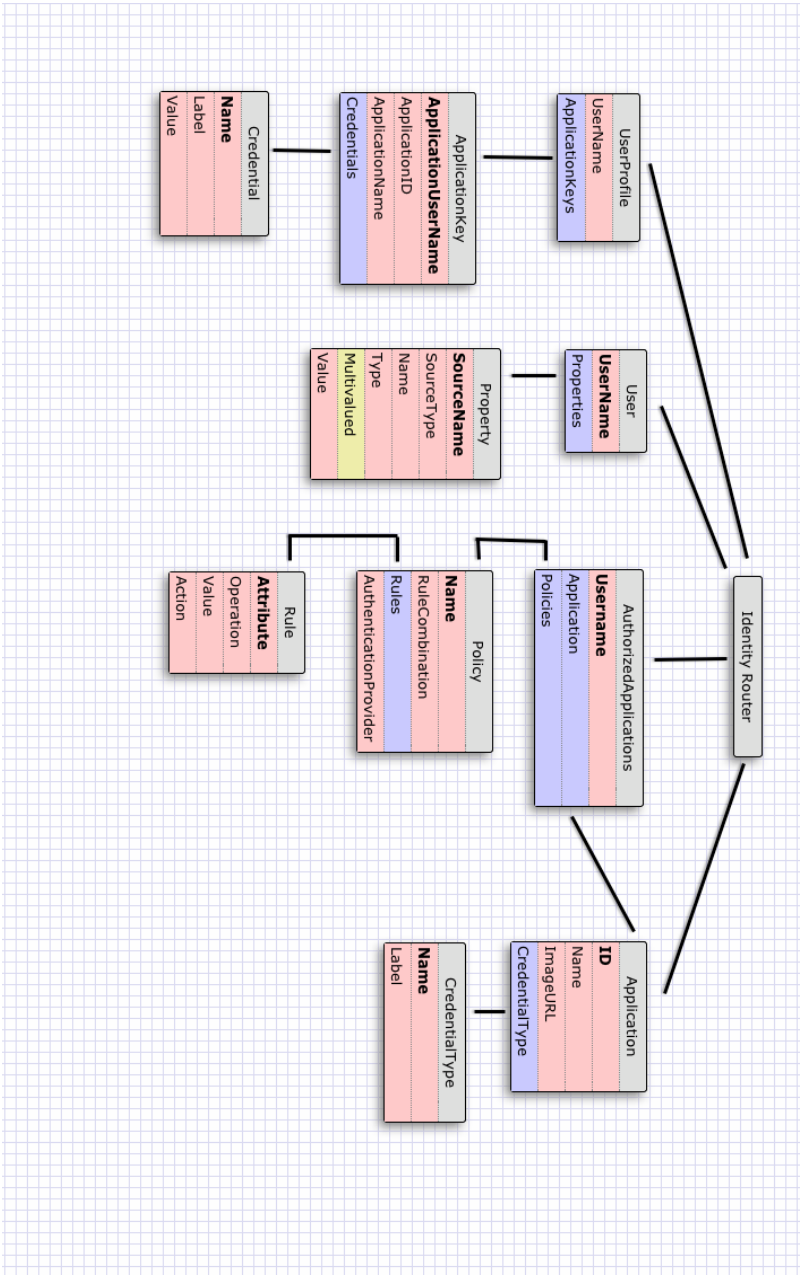
When designing our applications, it was decided that multiple resources would be placed into classes, as well as placing the code for the server requests into a class. Placing resources into classes was used in order to help facilitate further expansions of the applications. After placing a resource into a class, it became much easier to take certain parts of that class's data to output to the user. It also simplified POST and PUT operations, making generation of XML and JSON files or strings easier to implement and understand; multiple generation libraries use a class to generate the XML or JSON string.

Placing the server requests into a class also allowed for future implementation of additional functions. Keeping the request in the class allowed data to be stored within it and allowed further functionality when making requests. Keeping the request as a class and storing the data within it allows multiple different forms of user interfaces to easily use the class.

Keeping the server request in a function would require each user interface to create new methods of how to access or modify the data received from the server.

The Identity Router holds all the information relating to each individual user of the system. As seen in Figure 2A below, each user has a “UserProfile”, containing all the information about which applications the user has set up, along with the credentials associated with those applications. Users also have a “User”, which holds the information relating to their account creation date, account expiration date, display names, and the database, the UserStore, where the user is located. Each user contains a list of authorized applications, applications on the Identity Router that can be added to their UserProfile, this is a static list of applications.

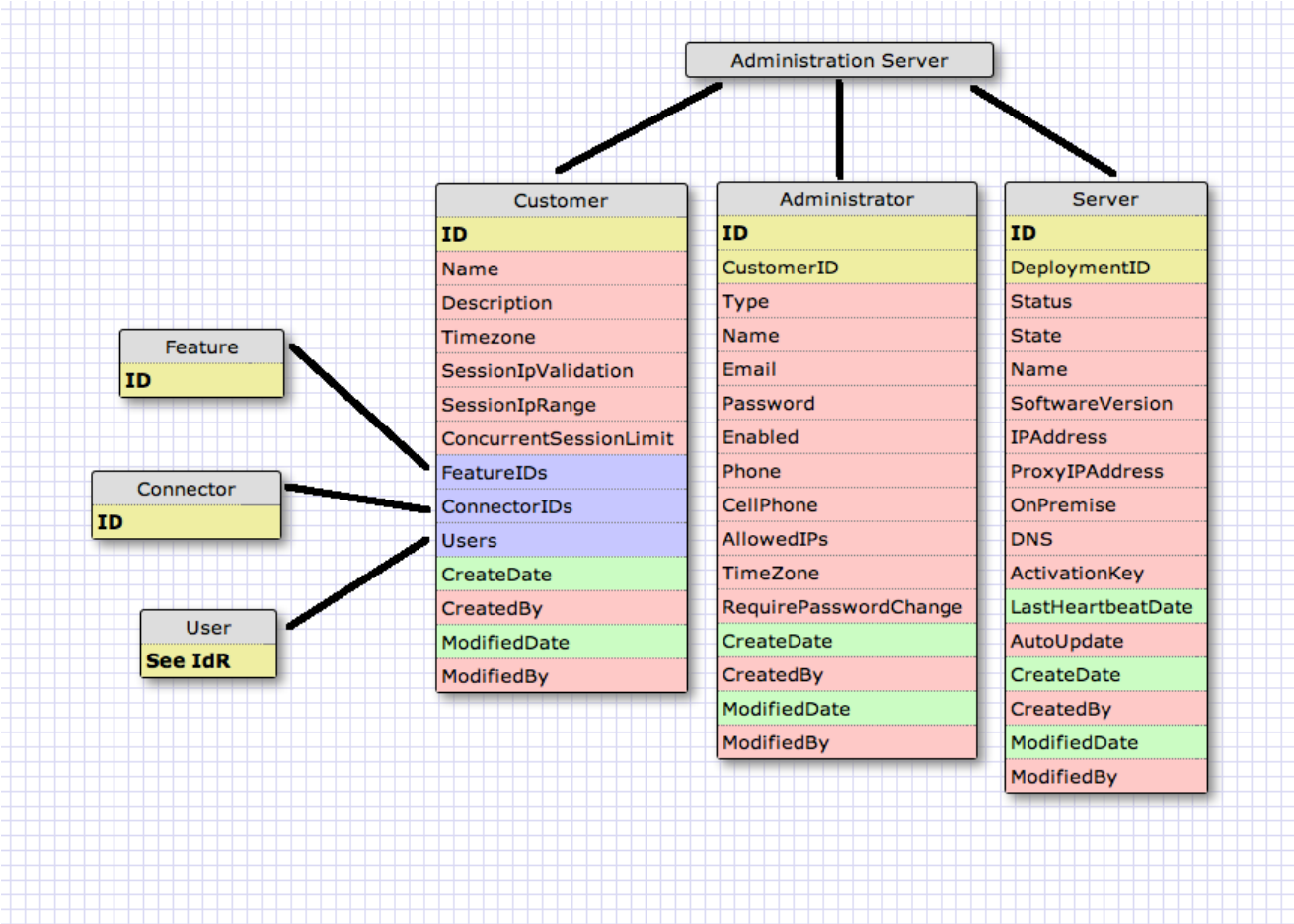
Figure 2A - Identity Router Database Schema:



*There are other resources on the Identity Router, our project only dealt with the resources in the above diagram.

The Administration Server houses a list of the administrators, the servers being used, and a list of the customers. Administrators have rights to modify the customers, add or remove servers, and modify UserProfiles on the Identity Router. The database schema in the Technical Design section shows all the information associated with the items in the Administration Server.

Figure 2B - Administration Server Database Schema:



*There are other resources on the Administration Server, our project only dealt with the resources in the above diagram.

The software we wrote was designed to interact with the Identity Router and Administration Server, to be used as a means to modify the resources existing on them. The software sends a request to the server, then the server returns the information which is then parsed, organized, and outputted by the software. Every element on the server is accessible to the software through a GET HTTP call. The documentation for the Identity Router and the Administration Server provided by Symplified outlines which resources can be created with a

POST or modified with a PUT. This basic functionality was required to exist in Ruby, Java, .NET and PHP, and then addition features were built on top of this to make this basic set of features more powerful and accessible to the user.

Our software interfaced with the Identity Router and the Administration Server through HTTP requests. Authorization, server side commands, and data to be posted were encoded and encrypted, then sent in headers attached to the HTTP requests to the server. The headers attached to the HTTP requests determined whether the inputted or outputted data would be formatted in XML or JSON. The headers also provided the authorization to access the server as well as the current date. The server then returned a response, which was parsed to produce an easily readable output for the user.

Figure 3 is a State Diagram demonstrating the flow of an application from the perspective of the user. This behavior is common to all of the applications we created.

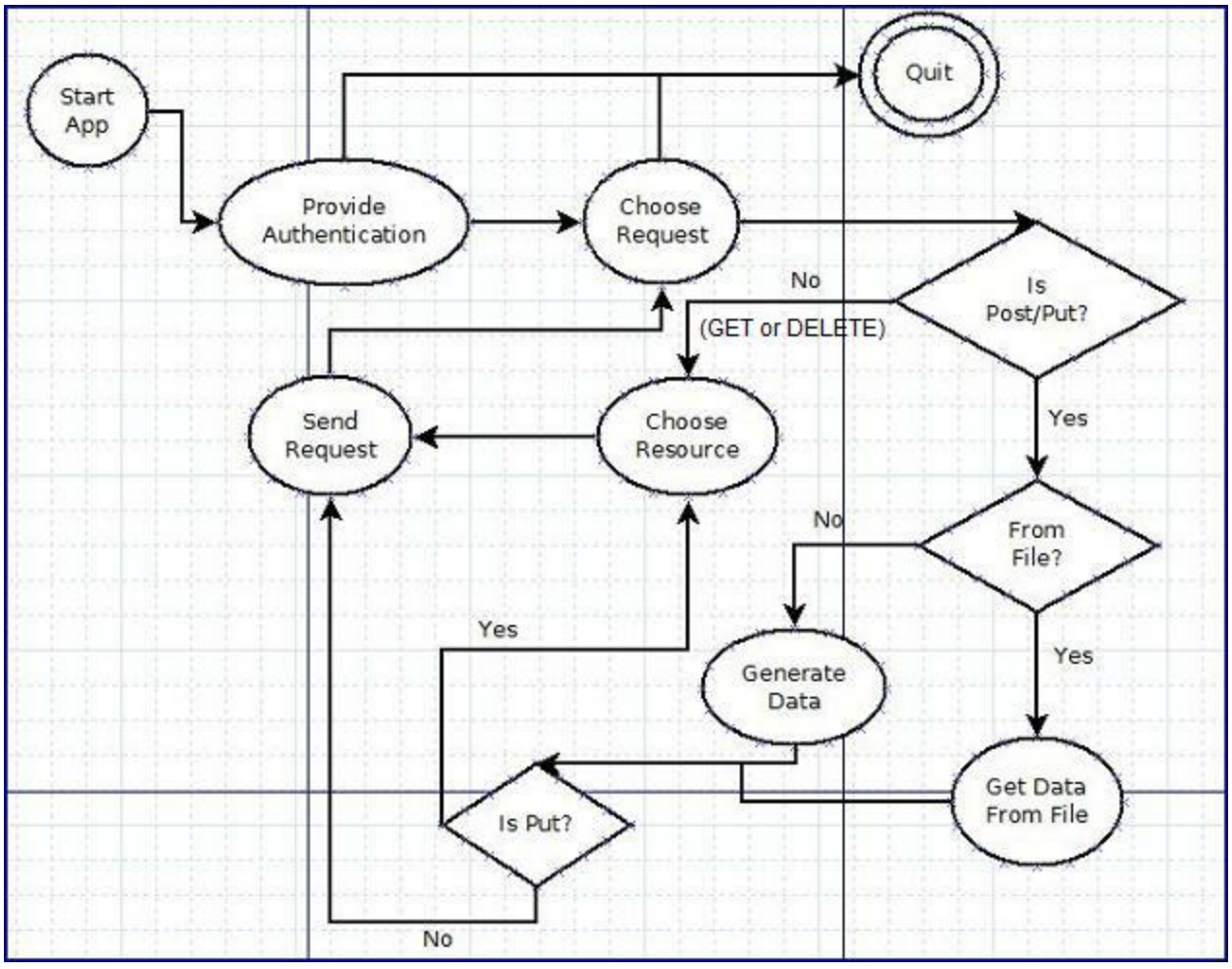
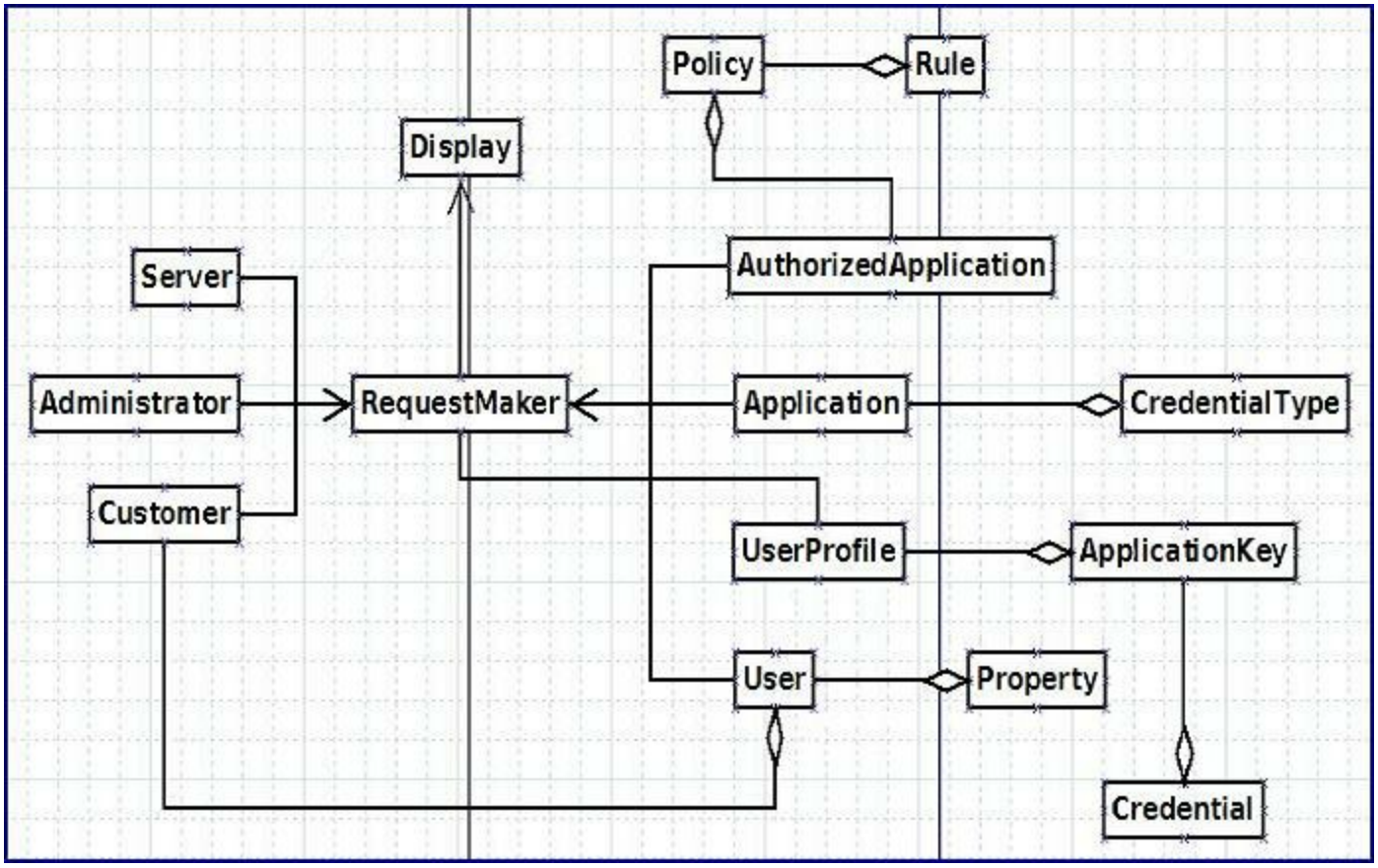


Figure 4 is a UML describing the general layout of all of the applications we wrote; this representation varies slightly between applications.



DESIGN & IMPLEMENTATION DECISIONS

For each language a variety of libraries were examined to decide which best fit the needs of software.

Ruby

Rest-Client library was used to send HTTP requests in Ruby. The HTTP-Client library was originally tried, but the Rest-Client library was built with RESTful web services in mind and made connecting to the servers very easy. With Rest-Client attaching headers and starting sessions was straightforward, and well documented.

Multiple XML and JSON parsing libraries exist in ruby, and the libraries that were chosen for this, LibXML-Ruby and json, had good examples that were easily built off of to support the data that we retrieved from the servers.

For the graphical user interface in Ruby, Ruby Tk was selected. Ruby Tk is cross platform; existing on Windows, Mac OS X and Linux. The example program we created could easily be run in all three environments. The installation of the Ruby Tk library in Linux was simple and the library is simple to use with large amounts of quality documentation available

online.

Java

In order to parse and generate the XML strings, Apache commons codec v1.6 was used. This is an external library that was chosen because it is the most commonly used library for parsing XML. The process for extracting the data from a file was simple to reproduce. Unfortunately, its simplicity also resulted in repetition of code across each parser. Generation of an XML string was also simple and did not have much repeated code.

PHP

To send the HTTP requests to the server, the Client URL (cURL) library was used. This is an external library, part of the libcurl package. cURL was used instead of some of the other libraries available in PHP, such as the basic “file_get_contents” or “http_get”, because it is easy to set all the needed parameters for the HTTP request and change from one request to another. The documentation behind this library was clear and easy to understand as well.

All the libraries used for parsing and generating XML and JSON data, encoding and encrypting the signature, and creating a graphical user interface, were, by default, built into PHP.

.NET

In order to parse and generate the JSON strings, the Json.NET library was used. This external library was used because the majority of the community supports it for more advanced JSON parsing. This library works best when working with JSON code that conforms to convention; extra code was written in the parsing functions in order to extract data without failing for the JSON data retrieved from the servers. The library also used an extremely simple JSON serializer that takes a class and makes a JSON string from it. For .NET, an additional class had to be made in order to create the proper JSON output, but minimal code was used to build the JSON string from the new class.

ROADBLOCKS

Throughout the course of the project there were several problems encountered that were more significant and time consuming than others. These problems were more significant than learning new syntax for a language or being unfamiliar with some a convention, they were subtle issues that significantly prevented overall progress.

A problem occurred in .NET when trying to set the date header in the signature. In some versions of .NET there is a method to set the date in the HttpWebRequest method by typing “.Date” after the method name. No method like this exists in .NET 3.5, so a workaround was found, which led to the use of the TcpClient Class. TcpClient would not normally be used for setting the date header, but part of its functionality allows the server to see the date, and allowed for the completion of the signature.

Throughout the project we discovered errors within the company’s server system. Some of the first errors we encountered were discrepancies between the company’s documentation

and the actual server content. For example, the field in Administrator Server's documentation reads "RequirePasswordChange", but the response from the server is "RequiresPasswordChange" with an "s". These caused some confusion when implementing parsing and generation functionality. In addition there was functionality that we were asked to implement that was not correctly implemented on the company server. Some of the POST commands were later omitted from the project when it was discovered that the server was not handling them appropriately. The only other error we encountered on the server was when a GET request specifying the return of JSON data was returned, the data was incorrectly formatted for a few resources. This caused problems when parsing the data.

RESULTS

At the end of the project, we have code to access and modify the Identity Router and the Administration Server in four different languages; Ruby, PHP, Java, and .NET. Each of these languages has their own unique graphical user interface that takes advantage of each language's unique features. Not all functionality has been added to each user interface due to time constraints, but every feature is represented in at least one language's interface and can be used as an example for the other languages.

Using the code our team produced, Symplified could expand on it, adding a full set of features to each of the four languages. They could also use the code as a building block for a more complex application that interacts with the Identity Router and the Administration Server in advanced ways, such as dynamically viewing connected users, POSTing more objects to the server, and modifying user access in real time . All the basic functionality already exists, the generation of signatures to connect to the server, code for sending and receiving server requests, and examples of parsing the data into usable class objects, lessening the development time for an application with these more advanced features.