Universal AJAX Interface Generation

Alex Boettger, Jared Martin, and Randy Campbell,
Department of Mathematics
Morningside College
Sioux City, IA 51106
{ alb019, jam021,campbell}@morningside.edu

Abstract

There are many web client-side frameworks and many web server-side frameworks. However, there are few frameworks designed to facilitate AJAX communication between client and server that provide support for both sides of communication. Those that do exist generally provide a compiler for the client-side language that generates JavaScript code (such as Pyjs for Python). We have developed a proof of concept application for generating AJAX interface code for both client and server. The interface code ensures that there can be no mismatch between what is sent by one end and expected by the other. On each side, data is sent via a simple function call. The software is designed to enable creation and integration of new modules for server-side languages, allowing it to become (as new modules are added) language agnostic on the server-side.

1 Introduction

Anyone that has ever created an AJAX interface knows that the process can be long and error prone. It would be much easier to create these interfaces with a framework that would generate the code for all the functions needed to package, send, and receive data. Then a developer would need to only worry about writing code that performed the desired task.

Most web frameworks focus either on the client-side or the browser-side, but not both. Those that do try to address both generally do so via some sort of template scheme or a compiler that converts code from the server-side language to JavaScript.

This project focuses on generating custom AJAX communication code for the browser and the server in a way that readily allows code generation modules for server-side languages to be added. This allows the system to become server-side language agnostic. The project has passed the "proof of concept" phase and is in process of becoming an easily usable tool.

We gratefully acknowledge Morningside College's funding of this project via a Summer Undergraduate Research Program (SURP) grant.

2 Goal of Project

The goal of the WinG project is to create a framework that creates custom AJAX interface code for both the server-side language and for JavaScript on the client side in a way that allows the framework to be be easily extended by adding generators for server-side languages. There are several advantages to generating the AJAX interface code for browser and server. One is the time saved and programming errors averted by removing from the developer the burden of writing this code. In fact, the only code the developer needs to write is code that does the specific task that a server-side script is supposed to perform given data from the browser or that a JavaScript function is supposed to perform with data it receives from the server. It also ensures that there can be no data mismatch errors from browser to server or server to browser.

3 Project Overview

The system provides AJAX interactions via function calls.

JavaScript code accesses the server via a simple function call, passing in data to be sent as parameters. Each such function on the JavaScript side has a corresponding function of the same name and having the same parameters on the server that does server-side processing of the data.

When the server-side script is ready to send data back to browser, it does so with a function call. The data is directed to a JavaScript function of the same name and having

the same parameters as the function called to send back the data. The JavaScript function then does the desired task, using the returned data.

Figure 1 illustrates the underlying architecture used to mediate AJAX communication, using a login interaction as an example. Rectangles represent components whose code is entirely generated by the system. Boxes with angled upper right corners represent components for which a function skeleton is generated. The code that needs to be added to the skeleton is the just the body of a function.

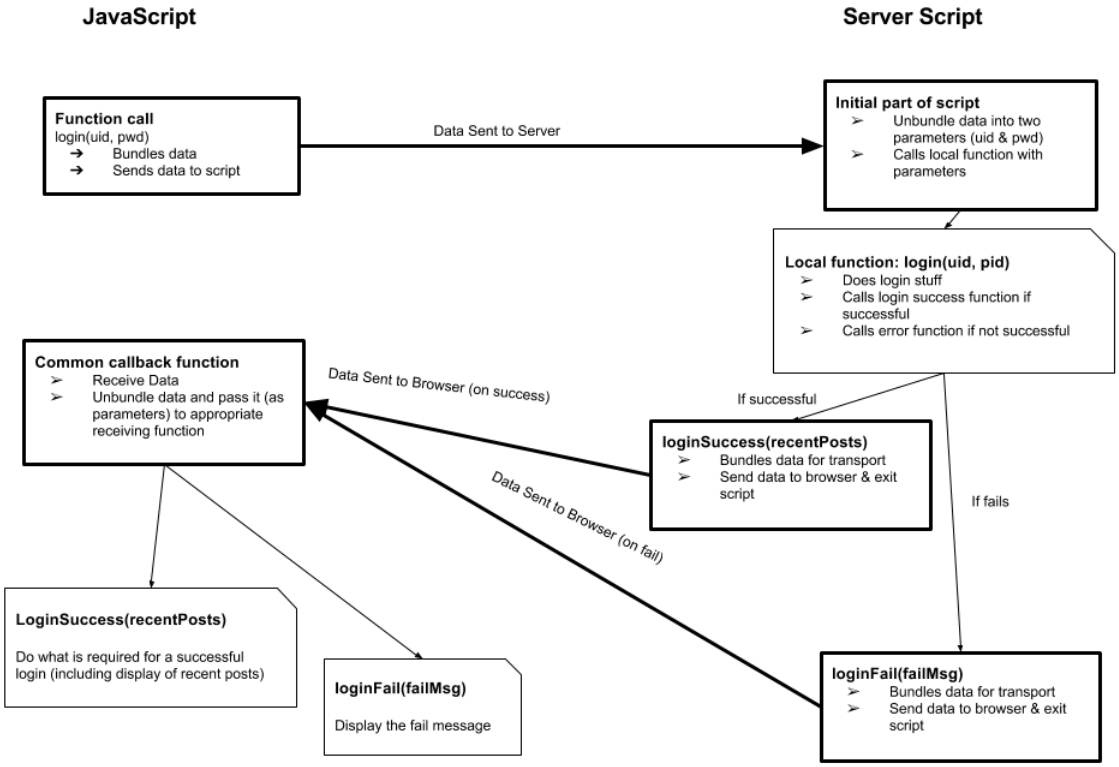

Figure 1: Basic Communication and Functionality of Interface

To ensure that the system could be easily extended to various server-side languages, we split the project into two main categories: client-side development and server-side development. Splitting the project into these two parts allowed us to focus on what each side needed to do in order to ensure accurate communication. It also enabled us to simplify the work of creating code generation modules for different server-side languages.

The client (browser) part is universal because the code generated for it is JavaScript. The code generated for the server must be specific to the language used on the server. Thus, there must be a code generation module available for the desired server-side language. At present, there is a module for C++ and a module for Python.

Figure 2 shows the architecture of the system.

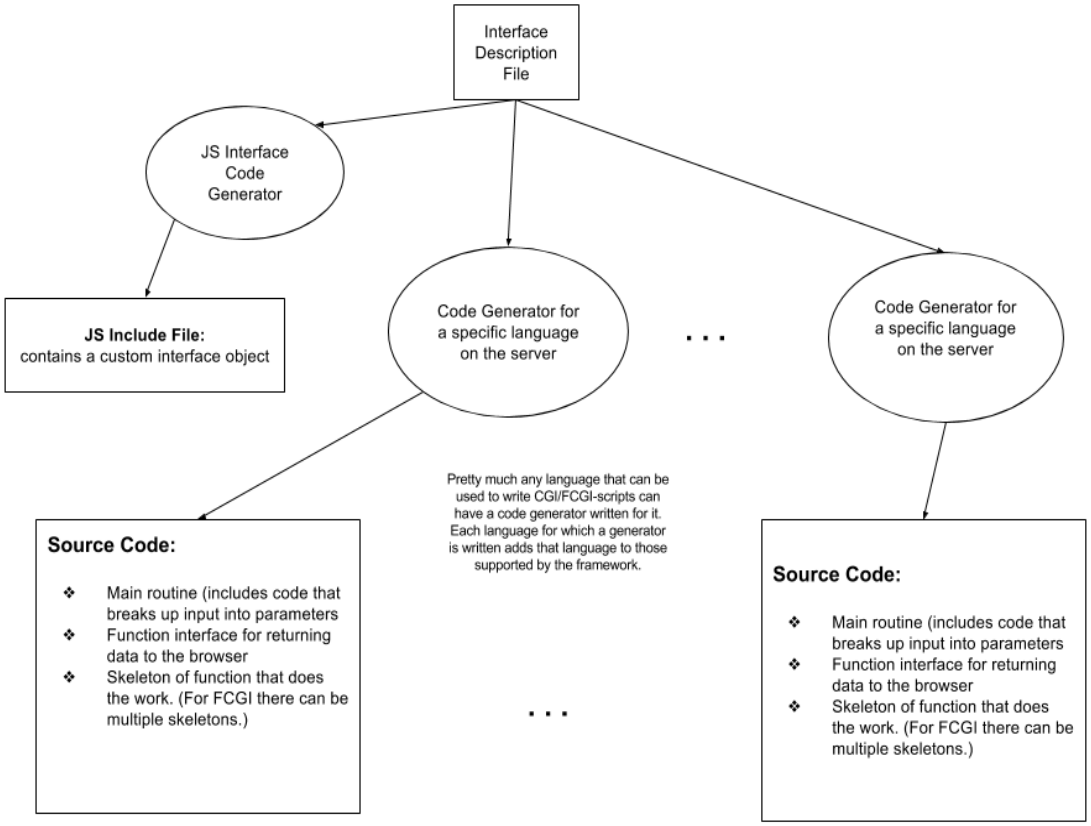

Figure 2: System Architecture

To begin development of the interface we first had to develop a description file syntax. Description file information provides the information necessary to generate code for browser and server. It provides the server domain name and path on the server to where the server-side scripts will be located. It also defines the function names, parameter names, parameter types, the AJAX request type (post or get), and whether or not a session key should be sent to the server for server-side functions. It identifies functions as being in cgi or fast-cgi scripts. In addition, it provides for server-side functions intended to receive uploaded files and server-side functions intended to download files. It also defines the function names, parameter names, and parameter types of functions that receive data from the server.

3.1 Description File Syntax

We use the example description file in Figure 3 to illustrate some of the description file syntax.

```
# filex description file

PATH: https://gimli.csci.morningside.edu/~username/cgi-bin/filex/

CGI:1
{
register(STRING uid, STRING pwd) POST;

login(STRING uid, STRING pwd) POST;

logout() POST: SKEY;

delete(STRING fkey) POST: SKEY;

upload() UPLOAD[*P]: SKEY;

download(STRING fkey) DOWNLOAD: SKEY;
}

RECEIVERS:
{
popMsg(STRING message);

loginSuccess(STRING skey);

delSuccess(STRING fkey);

fileInfo(STRING[] fdata);
}
```

Figure 3: Basic Description File

The line named Path contains the online location of the cgi or fcgi files. Following that is the section that contains all of the information for constructing the cgi functions (tag 1). Description files can define CGI or FCGI scripts (or both if the project demands it).

Tags 2-6 show elements of function descriptions. The first word represented by tag 2 is the name of the function. Tags 3 and 4 provide a parameter declaration. STRING refers to the type that is expected for this parameter and uid, is the name of the parameter.

Types currently provided by the system are string, int, real, string array, int array, and real array. (JSON can be sent in a string parameter.) A function can have zero or more parameters.

Some additional information follows a function declaration's parameter. First is request type (GET or POST) or a file upload/download specifier. The options are: GET, POST, DOWNLOAD, UPLOAD[1], UPLOAD[1P] UPLOAD[*], UPLOAD[*P]. Because file uploads require POST (and we default to POST for downloads) a request type is not needed for functions doing file upload and download.

File handlers defined with a 1 in the square brackets handles one file upload while the file handlers defined with a * handle multiple files. The P in the file handling designators are used to tell the interface that the browser to track the progress of the upload or download.

A request type or file upload/download specifier may be followed by a session key flag, as seen in the item tagged 6. This directs the system to include session key handling in the communication. This can be useful for session handling without using cookies.

The item tagged 7 provides declarations for JavaScript functions that process data received from the server.

3.2 Generating Javascript

There are two parts to the Javascript code for the interface. One is a library of general functions used by the system, which is included in the HTML file. The other is a JavaScript file containing the custom AJAX interface code. This is generated by the system for the application and is also included in the HTML file. The name of this generated JavaScript file is selected by the user at the time that the file is generated.

Suppose that a generated JavaScript file is given *jader.js* as its name. This file will contain a definition for an object named *jader* that contains all of the JavaScript functions necessary to satisfy the requirements given by the description file. The jader object organizes functions in sub-objects within itself. All functions for calling a cgi-script will be members of the *jader.cgi* object. All functions for calling an fcgi-script will be members of the *jader.fcgi* object. All receiver functions will be members of the *jader.skel* object.

The *jader* object will also contain a universal callback function that receives data from the server, determines which receiver function should receive it, splits up the data received into appropriate parameters for the receiver function, and calls the receiver function with the parameters.

The functions that are members of *jader.cgi* and *jader.fcgi* are fully generated. For functions that are members of *jader.skel*, the system generates function skeletons where code in the body of the function is created by a developer.

3.2 Generation of Server-Side

The server-side aspect of the interface requires a module for each server-side language used. Currently, the interface supports cgi and fcgi scripts written in C++ and Python cgi and wsgi scripts. The modularity of the interface allows for easy adaptability to future languages. Even though Python and C++ are two very different languages, it was easy and efficient to add the module to allow for Python based programming after the C++ module was done.

Although the code generated for the server language includes some common boilerplate, most of it is customized code for the application. It includes a main program and functions that match, in name and parameters, the JavaScript functions used to initiate an AJAX call. The main function processes the data sent by the browser (including conversions from strings to integers or floating point numbers where needed) and then calls the appropriate function. This function performs the required task, and then calls a function to send data back to the browser. The name of this function, and its parameters, match those of the JavaScript receiver function that will process the data when it reaches the browser.

The encapsulation of AJAX communication via function calls removes all tasks from the developer that can be removed. The only thing left for the developer to do, relative to AJAX communication, is to write code that processes data received by the server and code that processes data sent to the browser. Everything else is automated.

4 Future Directions

The WinG project was a proof of concept project. It was successful. It can be used as a development tool now. However, there are several ways it can be made more useful. Among them are:

1. Currently every module has the description parser embedded in it. Decoupling code generators from the parser by using an intermediate file would make it easier to create code generation modules for different server-side languages.
2. Tracking changes so that only description file items that are new or modified need to be generated would greatly simplify the incremental construction of web apps using the system. An intermediate file (as described in the previous item) would greatly facilitate this.
3. Adding communication with websocket servers (including the option of automatically generating a skeleton for a websocket server) would significantly expand the types of apps that could be created more quickly by using this system.
4. Adding modules for more server-side languages would also greatly increase the usability of this system.

5 Conclusion

Through the summer the students developed a proof of concept system for generating AJAX interface code. The system significantly decreases the time required to create web applications by generating much of the required code. Experience with the test applications we created often had 50% or more of the code generated by the system. The code generated by the system also precludes data mismatch and other sorts of errors that can result from handcrafted AJAX communication code. Having proved the viability of this approach, the next step is to convert the prototype into a convenient and complete tool.