Software Design Document

February 8th, 2017
Document version: 1.1

Team Name:
Website Automated Testing for Enterprise Reliability

Project Sponsor:
Jonathan E. Haynes

Faculty Mentor:
Mohamed Medhat Elwakil

Team Members:
John Sadie, Max Wason, Jason Le, and Peter Haschke

Table Of Contents

1. Introduction

The field of software testing has been exploded in recent years. One factor leading to this is the need to provide consistently reliable websites. Websites must not only be enticing to users, but must function exactly as intended. A single moment of downtime, or a click leading to the wrong page and the user getting frustrated, represents issues that results in the loss of extremely large sums of money if not dealt with properly. The omnipresence of modern websites means this particular issue is very large in scale and application.

Choice Hotels is a prime example of a company that relies heavily on their website for business. They are an American hospitality holding corporation founded in 1939 that currently franchises more than 6,300 hotels across 35 countries. Some of Choice Hotels well known franchises include Comfort Inn, Quality Inn, Sleep Inn, Econo Lodge, and many more. The company reportedly made upwards of 758 million dollars in the fiscal year ending in April 2016. In the hospitality business, and particularly with the size of a corporation such as Choice Hotels, an easy and error free way to find and book hotel rooms is vital. Without a well tested website Choice could potentially lose millions of dollars. Making sure every component of their web presence has been subjected to rigorous testing is necessary if they wish to run a profitable company.

This is however, more easily said than done. Choice Hotels manages over over 14 websites (one for each chain in the parent company, plus the main website), and each must be running smoothly at all times, a task that is not trivial even for a corporation of this magnitude. To do so reliably, the only answer is automation. Automating every step of the website creation and testing process guarantees that errors are dealt with before they reach the customer. Because of this, Choice Hotels has wisely decided to invest in automated testing, saving them millions in lost profits if an error was encountered in a public, live website. One such aspect of automated website testing is the initial step of gathering all the desired interactible elements in a given webpage, so that information about each element can be plugged into a testing suite to be run automatically. Currently this is a manual process; we at Team W.A.T.E.R. have been contracted to provide a solution.

We have decided to solve this problem via a custom Chrome plugin that will pull all interactable elements from a given page and output them in the desired file format(s). This gives a lot of powerful yet straightforward functionality to the end user, while still conforming to the technical requirements and limitations. The details of our planned implementation can be found throughout this document.

2. Implementation Overview

There are many tools that exist to test web applications but they all currently have the substantial downside of needing manual intervention at specific stages of the process (to account for the huge amount of diversity in modern websites). Our project aims to automate an aspect of this costly manual process by creating a standalone Chrome browser extension to pull selected elements of a webpage automatically. By using our Chrome extension (easily accessible via the Chrome web store), a user would be able to navigate to the page which they would like to test, open the extension box, and simply click a button. The file (or files) output the user has chosen will automatically download on Chrome. By doing so, users will have a simple and efficient access point for generating test cases for automated testing of these webpages, tremendously expediting the once manual and slow process. The technical details of this are discussed below.

2.1. Overall approach

As mentioned above, we have decided to use a Chrome browser extension for the project. Everything for the solution will be packaged within the extension. We wanted to make sure the extension has no outside dependencies, as this would introduce unneeded complexity. This means we will be primarily using Javascript for the logic of the programing and leveraging the Chrome API. The Chrome API is a prebuilt library of tools that make many things possible within an extension. We will mostly be using the messaging API within the plugin and the download API. Beyond this, HTML and CSS will be used to make the interface aesthetically pleasing. We will always aim at modularizing each part of program so new parts can be added at later times increasing functionality.

The files outputted will be in multiple formats. The basic one will be a standardized, but generic style, such as XML. The option for custom formats, such as a .java file for Selenium, will also be possible, as that will eliminate the additional step of transferring the elements from XML format to one suited for the test suite language (in this example: from XML to Java for testing in Selenium). By having this option, the final workflow will consist of the user navigating to the desired page, using our plugin to select the chosen interactive elements to be tested, and then simply clicking a single button to generate a file containing all of the elements. This file could then ideally just be dropped into the test suite, references to it could be added by the tester, and then the automatic test(s) could be run. This is much cleaner than the original error-prone and extremely slow solution of manually inspecting the page and transposing *each and every* element onto the testing suite by hand. Because of all this, our approach to the end product will save Choice Hotels, an incredible amount of time and money.

3. Architectural Overview

Since we are using a Chrome extension, the high-level architecture is already defined. A Chrome extension has three areas of execution of Javascript code. The first is within the extension's popup box. This area has access to any elements inside of the popup box and has very limited support for using Chrome API tools. The second area is the actual page the browser has loaded. This area has access to all elements on the DOM and also has limited access to Chrome API tools. The final area of execution is a backend space, that every plugin has. This area is hidden from view of all users. It has full access to all Chrome API tools. To send data between all three areas of execution the Chrome API messaging tool is required.

We will be using all three execution spaces in our implementation and utilize the Chrome messaging API in order to pass data between each. The frontend is represented by the first "popup" space will be used to get data from the UI on the popup box and send this to the "middleware" space that exists on the current webpage. The UI data will include the user selected output file types and the elements the user would like to parse the page for. This data will be retrieved, error checking will be done, and either error messages will be sent back to the UI for the user to see or the data will be send to the "middleware" execution environment via a message.

Once this information is received using Chrome API messaging, the next environment of execution or the "middleware" will be responsible for accessing the current page's DOM and pulling all requested UI elements, packaging this data along with the user selected output file types from the popup ,and sending it via API messaging to the backend. Using this environment allows us to be able to access the DOM and actually pull the elements.

The backend portion of the code is where the packaged element data is parsed, made into a file or files that the user has selected, and is downloaded into the user's browser using the Chrome download API. We must use the backend to do the heavy lifting and the downloads because it has the most access to Chrome APIs, mainly the download API.

This separation of environments allows us to make a very modularized product that separates each task into their own environments.

The following figure in the next page, figure 1: is a simple workflow architecture: The frontend is represented by the plugin GUI that takes user input and sends it to the middleware. This is then followed by the parsing of elements. The backend then takes over by writing to the output file.

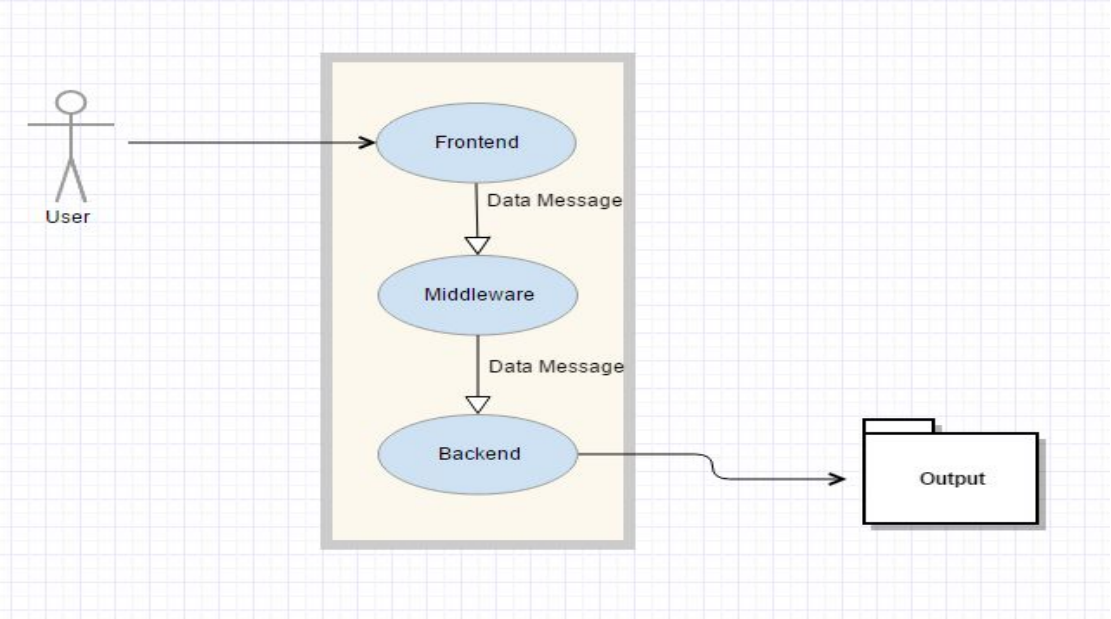

Figure 1 *- Workflow Architecture*

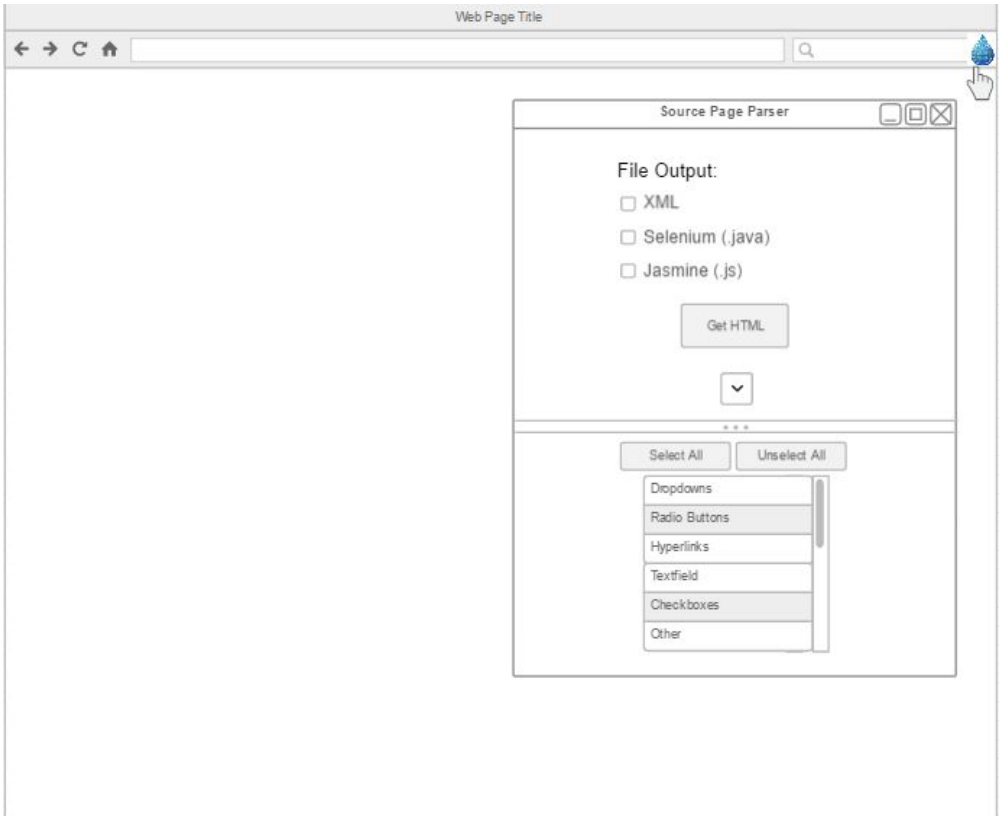

Figure 2 *- Wireframe*

The image in the previous page(page 5), figure 2: is a rough visual mockup of our product's user interface. Once the user has clicked on the plugin icon in their web browser, the small graphical user interface window will pop up. The user experience is meant to be simple and easy. The user will utilize checkboxes, buttons, a drop down, and a scroll bar.

The file output option will be checkboxes, for multiple possible selections. The lower section represents the "advanced options" where the user can configure exactly which UI elements to analyze (which are all selected by default).The 'Get HTML' button is the main one, actually calling the script and generating the automatically downloaded file(s). Note that colors are absent from this mock up, but will be used to direct the user in the actual application.

4. Module and Interface Descriptions

The following section lays out the comprehensive functionality of each component of the modules.

4.1. Frontend

As previously mentioned, the popup module is responsible for collecting user data from the popup box, checking the input, and sending it to the frontend module. The following functions will achieve this goal.

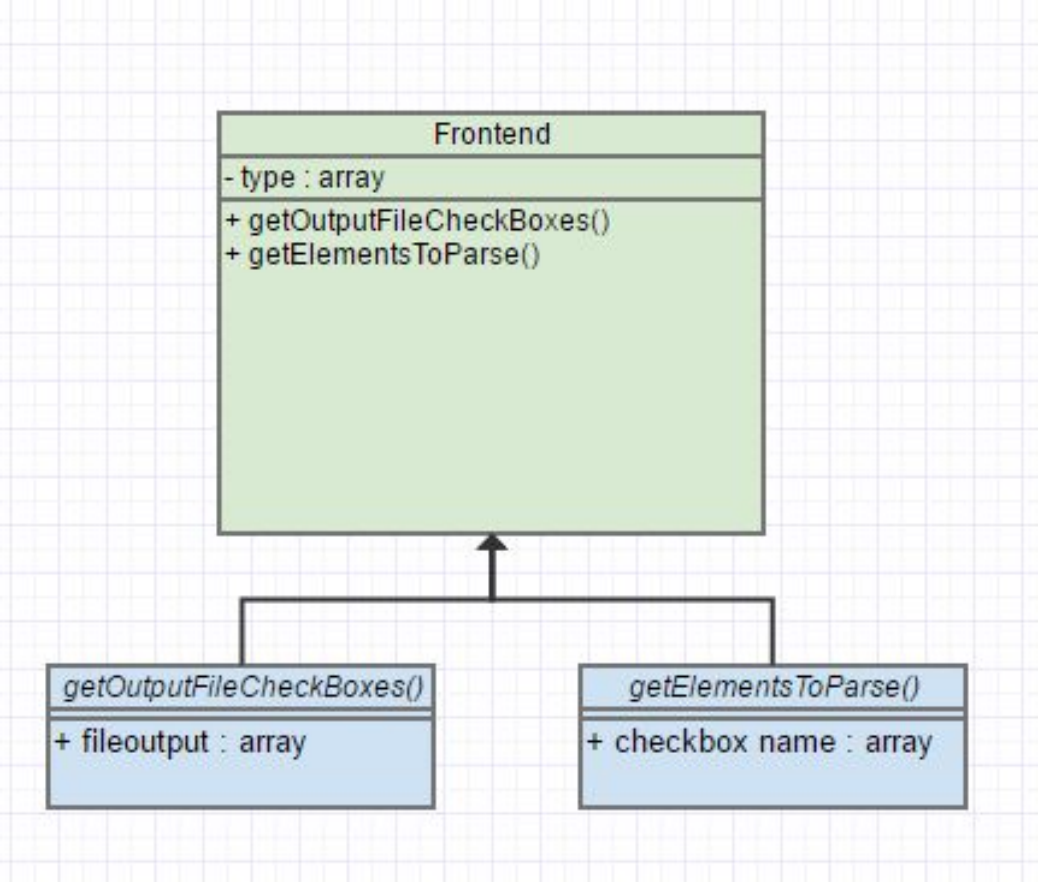

Figure 3 *- Frontend UML*

- Function: getOutputFileCheckBoxes():
    - Parameters: Nothing
    - Outputs: Array - File Outputs
    - Description: This function checks which checkboxes has been checked by the user for the output file. It returns an array with the names of the output files that the user checked for.

- Function: getElementsToParse():
    - Parameters: Nothing
    - Outputs: Array - Element Types
    - Description: This function does the same as the one above but for the element types the user checks to parse.

4.2. Middleware

Gets all elements from the page, instantiates custom 'Element' object, and passes that object array to the backend.

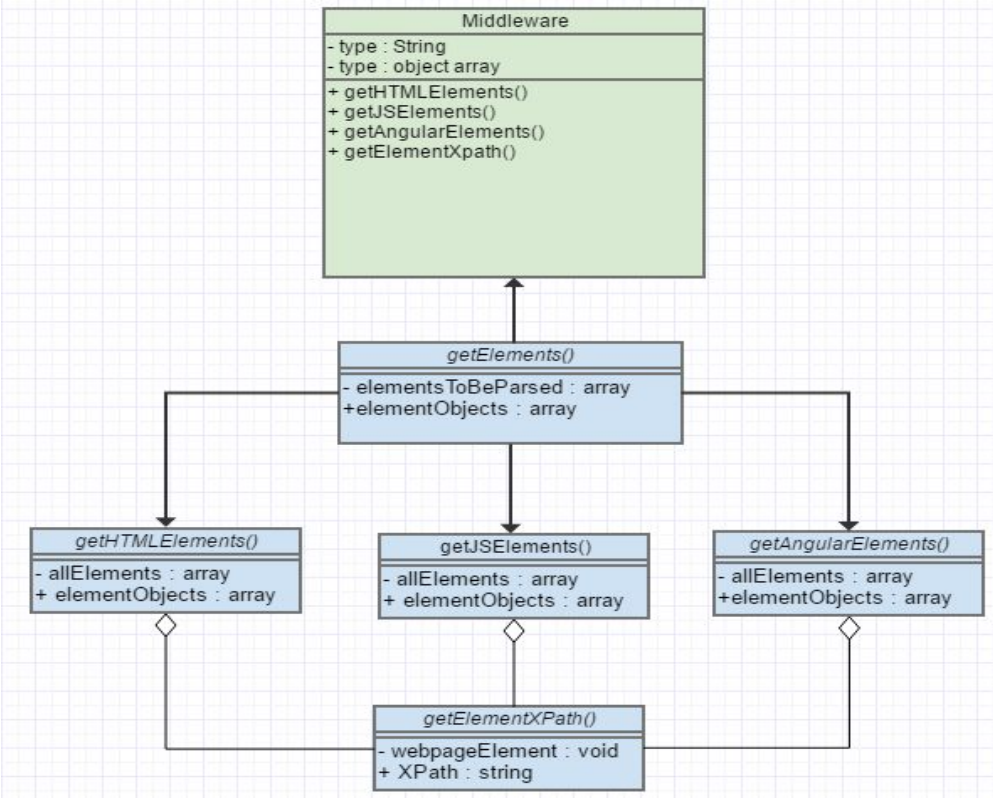

Figure 4 *- Middleware UML*

- Function: recieveFrontendData():
    - Parameters: Nothing
    - Outputs: Array - Output Files, Array - Elements to be Parsed
    - Description: Function that takes nothing as a parameter and listens for a message from the frontend containing file output checkbox and element checkbox data as arrays. Then closes the listener.

- Function: getElements():
    - Parameters: Array - Elements to be Parsed
    - Outputs: Array - Element Objects
    - Description: Gets all elements of the page into an array. The functions getHTMLElements, getJSElements, and getAngularElements are then called. An element objects array containing the parsed elements, is returned.

- Function: getHTMLElements():
    - Parameters: Array - All Elements
    - Outputs: Array - Element Objects
    - Description: Parses through the elements gotten from getElements function. Looks for the User specific elements that are generated through HTML (stores elements for elements that are generated through JS , Angular, etc.). The id, name, and XPath of the element is found.

- Function: getJSElements():
    - Parameters: Array - All Elements
    - Outputs: Array - Element Objects
    - Description: Parses through the elements gotten from the getElements function. While looking for elements generated through JS it keeps track of the elements generated by the HTML ensure duplicates of the same element is not recorded. The id, name, and XPath of the element is found.

- Function: getAngularElements():
    - Parameters: Array - All Elements
    - Outputs: Array - Element Objects
    - Description: This function is the same as the one above but instead it looks for elements that are generated through Angular.

- Function: getElementXPath():
    - Parameters: String - Webpage Element
    - Outputs: String - XPath
    - Description: Function that given an element , finds the XPath and returns it as a string.

- Object: Element
    - An object that stores the data of an element found from the functions above. These objects are stored as an array of element objects.

4.3. Backend

Takes the objects array and writes them as the output file types the user specified. Those files are then downloaded through the browser.

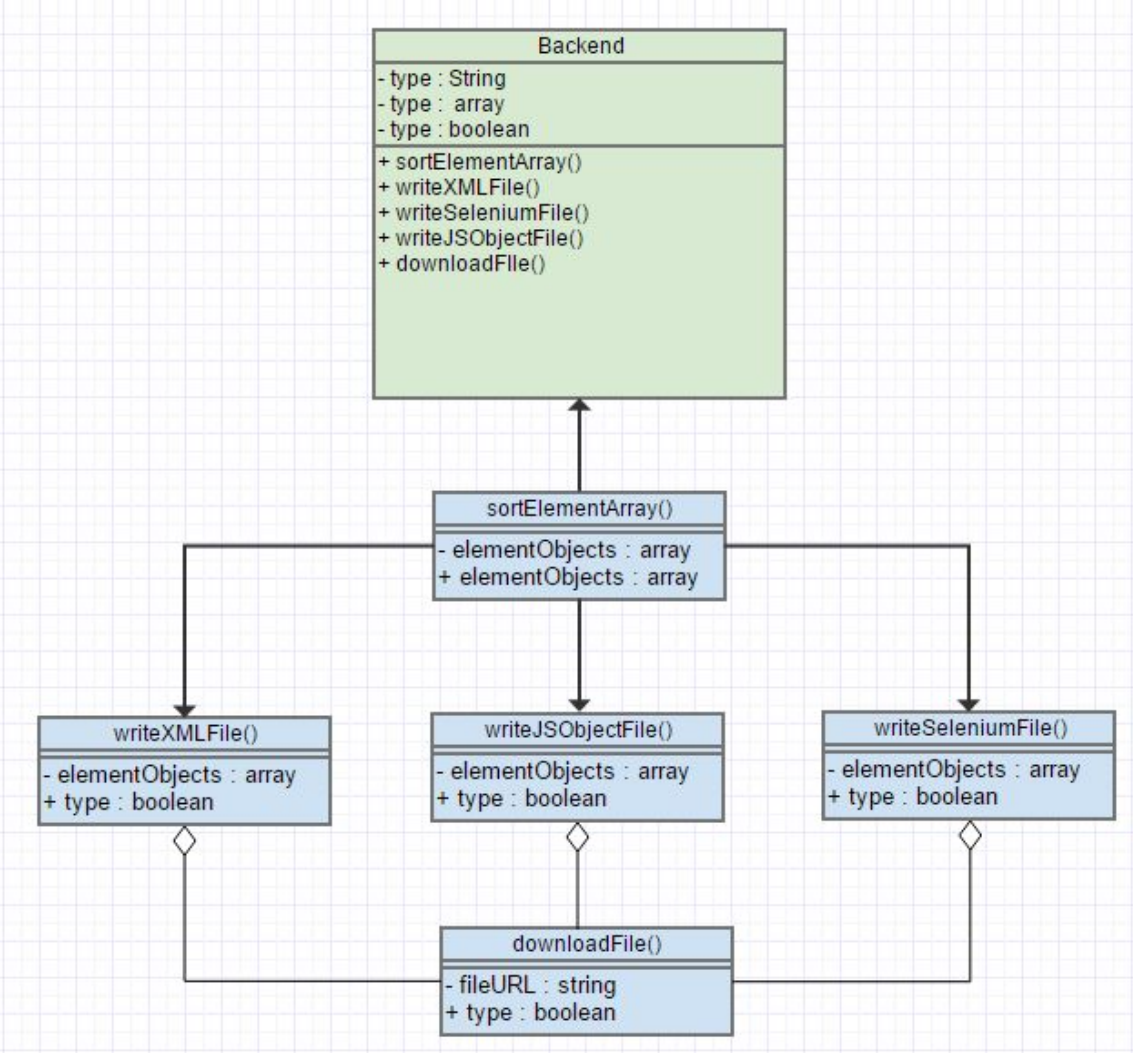

Figure 5 *- Backend UML*

- Function: recieveMiddleWareData():
    - Parameters: Nothing
    - Outputs: Array - Output File , Array - Element Objects
    - Description: Function that listens for a message sent from the middleware that contains two arrays one with the Output files, and another with the element objects.

- Function: sortElementArray():
    - Parameters: Array - Element Objects
    - Outputs: Array - Sorted Element Objects
    - Description: Takes the element objects array and sorts them by element type.

- Function: writeXMLFile():
    - Parameters: Array - Element Objects
    - Outputs: Boolean - Successful Writing
    - Description: Takes the sorted element objects array and writes them into a XML file.

- Function: writeSeleniumFile():
    - Parameters: Array - Element Objects
    - Outputs: Boolean - Successful Writing
    - Description: Takes the sorted element objects array and writes them as a file compatible with Selenium(.java).

- Function: writeJSObjectFile():
    - Parameters: Array - Element Objects
    - Outputs: Boolean - Successful Writing
    - Description: Takes sorted element objects array and writes them to a file compatible with .js Javascript file.

- Function: downloadFile():
    - Parameters: fileURL
    - Outputs: Boolean - Successful Download
    - Description: Takes a created file URL and downloads it using Chrome API download functionality.

The following figure shows the relationship of our system:

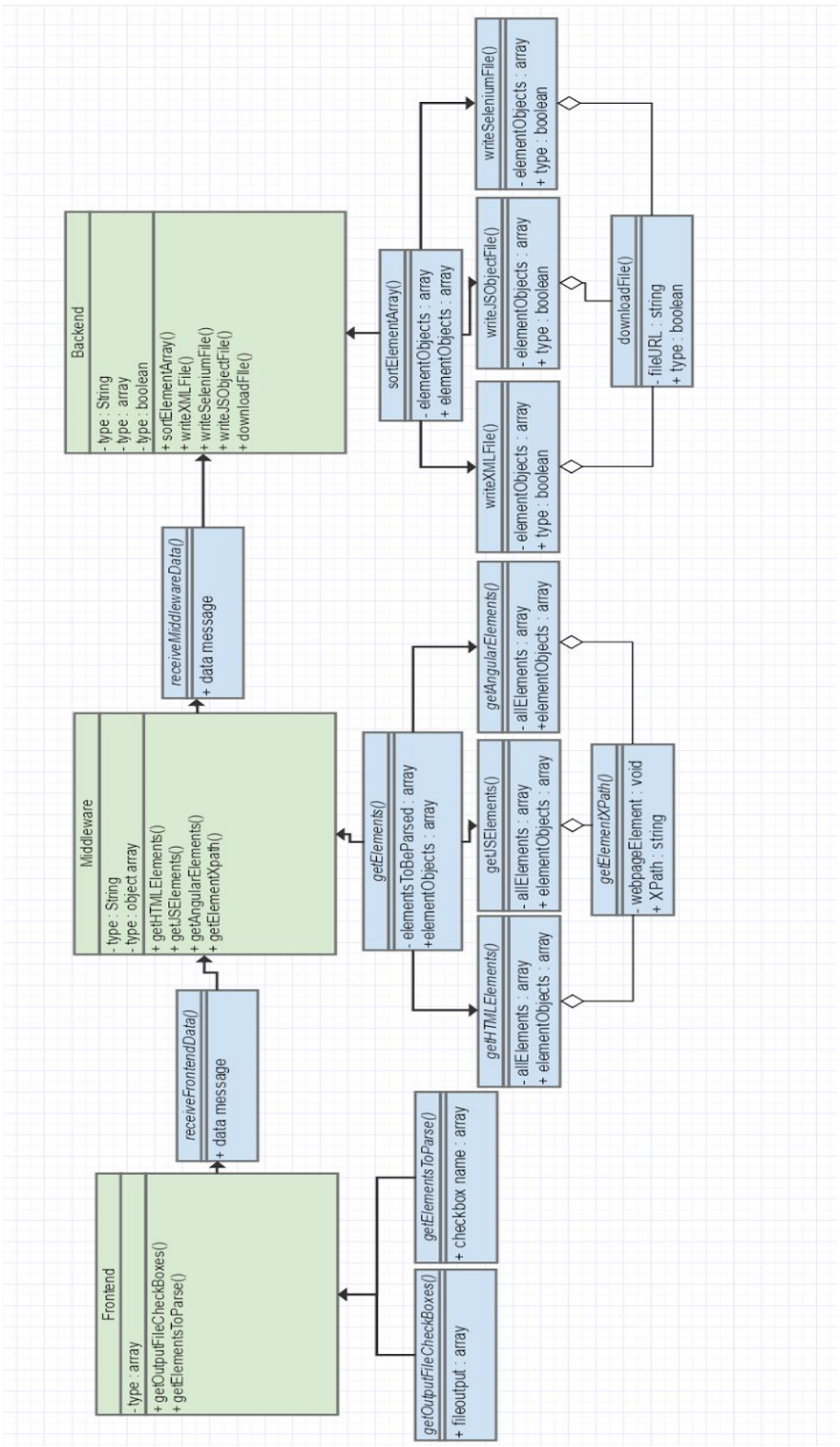

Figure 6 *- System Relationship*

5. Implementation Plan

Currently, we are working on furthering the design and implementation of our plugin. So far, we have been able to crawl through a web page and we've begun obtaining some interactive elements of that specific page's HTML. We have successfully embedded this functionality in a working Chrome plugin prototype and we simply need to keep moving forward. At this moment, it doesn't extract every single interactive UI element from a given source page, and the object oriented element structure it all rests on is not yet fully implemented. As you have read in the previous section, it is quite important that we obtain every interactive element, so addressing this (via a robust and flexible backend) is our current main concern.

Our work delegation thus far has been mostly seamless; we have met together, discussed, and then written down (usually on paper) a list of tasks to accomplish and then have broken those down to the individual level. While this has been effective so far, in order to further our goal of integrating with Choice Hotels' systems and workflow we plan on now using Rally, an online Agile development tool, to organize our workflow. We have already talked with Choice about how to properly leverage this system and will use it to map out iterations of our development using common Agile methodologies such as sprints, user stories, tasks, and story points. By centralizing the delegation of work into a unified area, with well-defined structures, we hope to break down the large task before us into manageable chunks that we can work efficiently on. The large scale process is shown below, in figure 5.1. The fine grained details will be present on Rally itself, it would be a waste of time to duplicate that information here, especially as everything isn't fully worked out yet. With that in mind, the following is a set of milestones that we will aim for in the upcoming months:

5.1. Project Workflow

Figure 7 *- Project Workflow*

6. Conclusion

Our project is an ambitious one. We are attempting to eliminate one of the largest remaining manual tasks in the web development testing toolchain for a multi-million dollar company. However, we are confident in our ability to do so. The document that you have in front of you lays out the detailed design that constitutes our vision of a successful project. We have done the necessary work to feel justified in our assessment; we truly believe that Team W.A.T.E.R. has the ability to create an elegant product that will not only solve this particular problem, but influence the very foundations of modern web development for the better.