Project: MathJax

Team Name: Team Two4Now

Team Members:
Charles Thiry
Hasam Solano-Morel
James Keels

Class: CSCI 362-02

Table of Contents

Introduction

MathJax is a cross-browser JavaScript library that displays mathematical notation in web browsers, using MathML, LaTeX and ASCIIMathML markup. MathJax is released as open-source software under the Apache license.

Chapter 1 – Initial Experiences

James Keels

Building the MathJax application on my local machine has been a learning experience to say the least. This is my first experience using Git and building an application. It amazes me how much setup is required to get code to work on a computer. I am still trying to fine tune some things; thankfully the MathJax site has a lot of documentation available. The biggest hurdle I ran into was with my Linux setup. I initially set it up on my Virtual Machine with default settings for memory. Soon I figured out that I would need much more and had to scrap it and start over…"Live and Learn" I guess.

Hassam Solano

My experience compiling/building MathJax on my machine began smoothly with a straightforward clone of the platforms GitHub repository. Once it was completed I proceeded to read through the documentation and quickly realized I would need to pick up a few new skills in Terminal. Fortunately MathJax is very well documented and included an installation guide that is descriptive and concise. The guide helped me understand what the script lines I was writing were doing which I think will be helpful should a problem arise at a later time. I did encounter a few set backs especially when configuring certain dependencies and when working with Apache (no prior experience). To help resolve these issues I referred back to the available documentation and to online sources on StackOverflow. Overall the compiling/building process was challenging but I was able to gain new knowledge about using the Terminal command line.

Charlie Thiry

After cloning the repository into a folder in my linux VM and installing the necessary dependencies (Apache, PHP, MySQL), I feel a little bit better about working in the famous LAMP stack that I am not used to. As Hassam pointed out, MathJax is a large project that is contributed to by many developers and therefore has excellent documentation for the noob like myself looking to jump right in. Though I will need to dedicate more time and effort to understand the code base itself, I have an overall optimistic viewpoint towards the project.

Chapter 2 – MathJax

Testing Process

We will use Python scripting with the Selenium testing framework to create automated test cases for MathJax. Our script will use Selenium to open a browser and navigate to a specific html file of our own creation. This HTML file will act as our testcase as it will display a specific feature of MathJax's text conversion process. Selenium will then take a screenshot of the page and contrast it pixel by pixel against a previously specified image of the correct conversion from MathJax.

Test Cases

ID:001
Title: One Variable and Positive Literal
Preconditions:
1. Import MathJax

Steps:
1. Create test HTML file with using MathJax markup.
2. Add the file path to the testing framwork.
3. Run test.
4. Verify that the test HTML matches oracle.

Expected Results:
1. Message prompt asserts equality of test and oracle pixel arrays

ID:002
Title: One Variable and Negative Literal
Preconditions:
1. Import MathJax

Steps:
1. Create test HTML file with using MathJax markup.
2. Add the file path to the testing framwork.
3. Run test.
4. Verify that the test HTML matches oracle.

Expected Results:
1. Message prompt asserts equality of test and oracle pixel arrays

ID:003
Title: Two Variables
Preconditions:
1. Import MathJax

Steps:
1. Create test HTML file with using MathJax markup.
2. Add the file path to the testing framwork.
3. Run test.
4. Verify that the test HTML matches oracle.

Expected Results:
1. Message prompt asserts equality of test and oracle pixel arrays

ID:004
Title: One Variable Squared
Preconditions:
1. Import MathJax

Steps:
1. Create test HTML file with using MathJax markup.
2. Add the file path to the testing framwork.
3. Run test.
4. Verify that the test HTML matches oracle.

Expected Results:
1. Message prompt asserts equality of test and oracle pixel arrays

ID:005
Title: Square root of one variable
Preconditions:
1. Import MathJax

Steps:
1. Create test HTML file with using MathJax markup.
2. Add the file path to the testing framwork.
3. Run test.
4. Verify that the test HTML matches oracle.

Expected Results:
1. Message prompt asserts equality of test and oracle pixel arrays

Chapter 3 – Architecture

File Structure

- •TestAutomation/
- •docs/
- •project/src/MathJax ** where MathJax source code is stored
- •reports/ ** where past test results are stored
- •scripts/runAllFiles.py **main script to run
- •testCases/ **Where test cases are stored

Flow of Execution

- •-test case file names are stored
- •-selenium browser pulls up skeleton file
- •-skeleton.html imports mathjax and sets up easy testing input/output
- •-selenium pulls up browser and navigates to skeleton.html
- •-test case information is pulled from txt files
- •-data is parsed as json strings
- •-test value is put into input box and output is observed
- •-output is compared to oracle
- •-test case result is appended to result html
- •-result html is displayed in a newly opened browser ***

<u>How-To</u>

How To Run

- •-Navigate to scripts folder
- •-Run "python runAllFiles.py"

How To Add Test Cases

- •-Create a new txt file in testCases folder
- •-Write a json string that follows the example:
- •{"id": "001", "title" : "Equation 001", "req": "Must display in line mathMode equation", "component":"tex2Jax", "method":"PreProcess", "testVal": "x + 2", "oracle":"x+2"}

<u>Experience</u>

The most difficult part of this deliverable was getting MathJax to correctly compile and display the MathML in the browser. This has not explicitly been done before so looking for code that did this for us on the web was impossible. We had to improvise from multiple examples and spend a lot of time on trial and error before we were able to get the correct JavaScript that pushed MathML into the browser like we wanted. Overall, this was an interesting and challenging task that was enjoyable to accomplish.

Chapter 4 – Experience

For this increment of our project we made some steps in the wrong direction. We completed our testing framework with 25 oracle images that were compared against screenshots taken by the selenium automated browser. As our browser itself took these oracle images, we truly weren't testing anything more than our framework itself, and this became apparent when it came time to present this to the class. In order to truly test MathJax, we decided to use MathJax's ability to convert LaTeX to MathML. We created an html file that imported MathJax's js and used some further scripting to force the MathML output to the screen. Our selenium browser then inputs the test case value and waits for the MathML to be drawn and extracts that from the page to compare with our textual MathML oracle. This method has proven to be far more effective than our previous method of testing MathJax.

Chapter 5 – Adding Faults

We inserted five faults that could easily be caused by developers making typos.
Source File Edited: project/src/MathJax/config/TeX-AMS-MML_HTMLorMML.js

Fault 1:

By replacing "mfrac" with "mfrac-fault", the mml that is generated is incorrect. This should make testCases/tc001.txt fail as it uses fractions.

```
a.mfrac = a.mbase.Subclass({
    //FAULTFAULT

    type: "mfrac-fault",

    //END FAULTFAULT

    //type: "mfrac",
```

Fault 2:

By replacing "mroot" with "mroot-fault", the mml that is generated is incorrect. This should make testCases/tc0023.txt, tc0024.txt, and tc0025.txt fail as it uses roots.

```
a.mroot = a.mbase.Subclass({
    //FAULTFAULT
    type: "mroot-fault",
    //END FAULTFAULT
    //type: "mroot",
```

Fault 3:

By replacing "mover" with "mover-fault", the mml that is generated is incorrect. This should make testCases/tc022.txt fail as it uses mover.

```
a.mover = a.munderover.Subclass({
    //FAULTFAULT
    type: "mover-fault",
    //END FAULTFAULT
    //type: "mover",
```

Fault 4:

By replacing "mtr" with "mtr-fault", the mml that is generated is incorrect. This should make testCases/tc021.txt, tc022.txt fail as it uses tables.

```
a.mtr = a.mbase.Subclass({
    //FAULTFAULT
    type: "mtr-fault",
    //END FAULTFAULT
    //type: "mtr",
```

Fault 5:

By replacing "mtd" with "mtd-fault", the mml that is generated is incorrect. This should make testCases/tc021.txt, tc022.txt fail as it uses tables.

```
a.mtd = a.mbase.Subclass({
    //FAULTFAULT
    type: "mtd-fault",
    //END FAULTFAULT
    //type: "mtd",
```

Experience

This was a difficult experience. Finding the right piece of code to inject faults into was the most difficult part. Much of the code base is contained in compressed JavaScript files that we had to "beautify" to be able to read. Once doing so, we realized that the code is not well organized or documented as they often use single letters for variables instead of self-documenting variables. This made things very difficult, so we began to comment out large portions of code to find parts that actually affected our process. We finally found a remote file in the config folder that broke the process when we commented it out. We then quickly found within that code the correct lines and then inserted 5 faults. Overall, it was a challenging task and it was rewarding to finally conquer it.

Wrap-Up and Final Thoughts

What one word or phrase best summarizes the experience of team Two-4-Now in regards to this particular project.... “Roller-coaster”, plain and simple. From early miss-starts to giant setbacks due to misunderstandings, this venture has tested our resolve. Collectively we feel that a lot of the difficulties could have been avoided by more guidance in the initial stages. Very few of us (if any) in the class have had any prior experience in automated testing. And even the few that have, may not have had experience using the technologies required for this class. While the “jump in the deep end” approach is understood, the real-word application of such a paradigm is murky at best. Most of the students taking this class will either find themselves starting a new venture or joining a business that already has a working development model. In the case of the new venture, the initial developers would probably build test frameworks using technologies with which they are already familiar, if the choose to build one at all. If its an already established business, they will (or should) already have a testing framwork implemented and if they don't, its hard seeing a newly minted Jr. Developer being elevated to an “Engineer – In Test” position. With these points in mind, the “jump in the deep end” style is counterproductive and inefficient.