Chad Maughan 04/27/2012
CS6890 - Research Project

Abstract

Web applications are increasingly moving from the server side to the client side (browser). Traditional, request/response applications are quickly being replaced by single (HTML) page Ajax (XHR) heavy applications. Frameworks such as Google's Java based GWT and Javascript based Backbone.js, Spine.js and Ember.js are facilitating this move. With this migration, testing frameworks need to move from the server to the client as well.

Problems often occur on the client-side in the form of Javascript exceptions and transport (network) related errors. As such, testing needs to be done the same way that the user consumes the web application.

Introduction

Like most projects of mine, this was overly ambitious for the allotted time. As such, I didn't complete as much as I had hoped in my initial project plan. A large amount of my time was invested in obtaining a working understanding of the underlying technologies used. Prior to this project, I had no experience with Selenium Webdriver, Neo4j, or BrowserMob. My hope is that these technologies prove useful for me for many years to come.

The project is broken up into two main sections. The first part is responsible for crawling a rich web application and modeling it in Neo4j, a graph database for later test suite creations and reductions.

In the context of this project, it is important to understand the meaning of a Uniform Resource Locator (URL) fragment. A URL fragment is a portion of a URL that points to a subordinate resource. A URL fragment identifier is introduced in a URL by a hash mark # or a "hash bang" #!. Prior to the existence of rich web applications, a fragment identifier was typically used to denote a position within the page. With the introduction of rich web applications, the fragment identifier use has increased. It is now used to denote a state in the application or possibly trigger an asynchronous request for data to the server without requiring a full page refresh.

The crawling looks at each URL or URL fragment (in the case of client side anchors) as a state of the application. Each HTML anchor element (an <a> tag) clicked represents a transition between these states. As the subject applications for this project are rich web applications, the HTML anchor links may not represent a tradition request/response resulting in the reloading of a resource. In some cases, a link, when clicked, may do nothing but display data that has already been preloaded. In other cases, it may start an asynchronous request to the server to retrieve information for immediate display.
In the Neo4j database, a URL (including any associated fragments) is recorded as a Node in the graph database as a source and or target and the relationship between Nodes is represented as an Edge. The crawling is done with a depth-first strategy.

The second portion runs those recorded tests via prioritization or a reduced test suite. Prioritization is determined by link pixel location. When each HTML anchor tag is crawled, a single number is calculated based on the number of pixels from the left and top most pixel. According to Jakob Nielson's usability study[1], links that are placed along the top and the left (usually as navigation features) are more important than those in the bottom right. During the testing, the project starts up an embedded, BrowserMob proxy server that allows for HTTP request and response interception. With response interception, a custom Javascript array is injected into each page that captures any client side exceptions that are thrown. Those are then extracted by Selenium for later investigation. BrowserMob also records latency and network transport problems the site may experience.a

Crawl - Site Analysis

The first step in test creation is to crawl the rich web application from the client side using Selenium Webdriver. The project contains a main `Crawl` class that requires implementations of two customized classes, a `NodeMapper`, and a `UrlCrawler`. The project provides default implementations of those classes, `DefaultNodeMapper` and `DefaultUrlCrawler`.

The `NodeMapper` is responsible for converting a URL state into a Node in the graph. It also creates a Neo4J index that allows for easy searching and querying by Node attributes such as link location.

The example implementation of `NodeMapper` in the project converts an application state to a Node as follows:

```
@Override
public Node mapNode(String url, WebElement anchor) {
        Node node = this.graphDatabaseService.createNode();

        // set the href as a property
        node.setProperty("href", url);

        // set the pixel location of a link as a property
        if(anchor != null) {
                int location = anchor.getLocation().getX() +
                        anchor.getLocation().getY();
                node.setProperty("location", location);
                // get the anchor text as a property
                String anchorText = anchor.getText();
                node.setProperty("text", anchorText);
         }
         // add the index
         if(this.index != null) {
                 index.add(node, "href", url);
```

[1]Jakob Nielson - Horizontal Attention http://www.useit.com/alertbox/horizontal-attention.html

```
            }
            return node;
        }
```

Additional customizations can be made to the UrlCrawler include the ability to 1) ignore specific URLs (via a blacklist), 2) define what is a valid URL, and 3) set a custom delay in milliseconds to pause between each page process.

One limitation of the crawling functionality is that each

The design goal of the application crawling functionality of the project is to be as generically applicable as possible ensuring compatibility across all rich web sites.

Test - Prioritization and Reduction

The second part of the project is the reduction of test cases. As rich web applications can have many states per logical view, reduction is an important part of making test suites manageable.

The process for reducing a set of URLs is to graph the hierarchy of the URL structure to identify excessive branching on particular levels. As an example, consider the following example URL structure:

> http://example.com/#/state/{state code}/county/{county name}/city/{city name}
> or an example with real data
> http://example.com/#/state/ut/county/salt-lake/city/murray

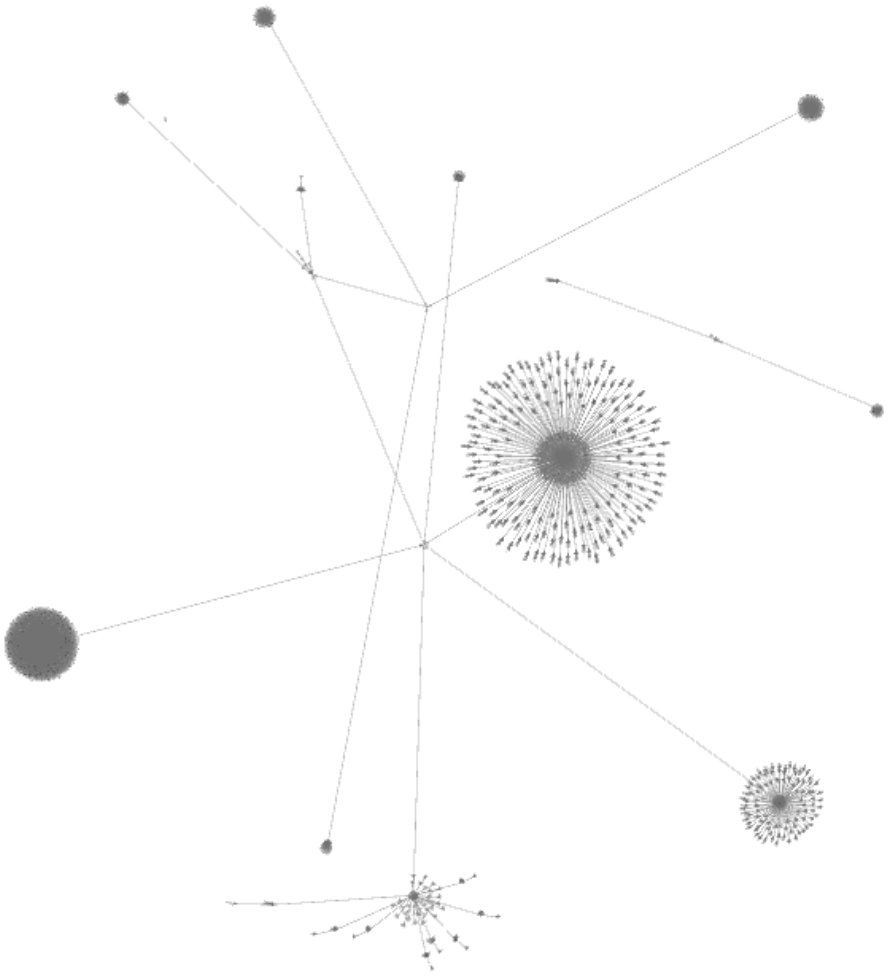
A graphing of this url structure would have a root node representing http://example.com/#/ with one child node connected to it, namely http://example.com/#/state/, then 50 nodes connected to that node for each state. From each of those 50 nodes, there would be a single county Node, and then branching of 3,141 nodes (for each county). This would be repeated for cities as well. A breadth-first search of the hierarchy with the calculated number of branches then makes it possible to identify what portions of a URL structure are variables.

An ideal reduction technique would recognize that the same templates are used with varying data. With the example URLs given above, the application would have a state template for the 50 states, a county template and a city template. Instead of exhaustively testing all states, counties, and cities, the reduction would randomly select a small sample size.

While not implemented yet, I would like to have a sample size calculator option added to this. One would then be able to sample size calculations. If you have a degree of branching that was 3,000+ and you wanted a confidence interval of 95% tested then the reduction would randomly select approximately 350 nodes to test.

Here is an example visualization of branching analysis. The “flowers” represent pruning

opportunities for reduction.



Summary

While the pieces of the project are functional, there is a lot of work that can be done on polishing and better integrating the code. Additionally, specific metrics on how many errors were discovered with a comprehensive crawl versus a reduced test suite needs to be explored. I enjoyed this project immensely and have a much deeper respect for testing. No longer will I just guess what can go wrong in an application. I'll apply the methods and combinatorial algorithms learned in the class to be much more comprehensive and methodical about creating test suites.

Conclusion

As the web development community continues to cohere around client-side MVC frameworks such as Backbone.js and Ember.js, and more functionality moves from the server to the client

side, testing rich web applications will become more important. While many of the same algorithms applied to other types of applications can be translated to rich web applications, there is much work that can be done.

Resources

The source code for this project is available at:
http://code.chadmaughan.com/cs6890-testing/

Source code for the project can be cloned with the following git command:
$git clone https://bitbucket.org/chadmaughan/cs6890-testing.git

The code is written in Java, is a Maven project and developed using the Eclipse IDE.