IMPLEMENTATION OF A VISUALIZER FOR WEB REQUEST DATA
GENERATED BY AN INTERNET INFORMATION SERVICES SERVER

by
Daniel James Garrett

A thesis submitted to the faculty of The University of Mississippi in partial fulfillment of
the requirements of the Sally McDonnell Barksdale Honors College.

Oxford
May 2014

Approved by

Advisor: Professor Tobin Maginnis

Reader: Professor Jianxia Xue

Reader: Professor Dawn Wilkins
I was tasked by Graham Green at FNC, Inc. to design a web tool that is capable of visualizing log data that is produced by FNC’s many servers. These logs contain data that will warn Graham about web pages that are failing, servers that are experiencing heavy loads, and sundry other pieces of information, but the data is pure text, and each server has thousands of logs. A visual tool is needed to display necessary information about the data. Rather than make a graph for each individual purpose that might be needed, I built visualization tools that are customizable for every conceivable use of the data. The log data itself is taken from the server that created it, using a tool called nxlog. That data is then sent to a proprietary API, which stores the data in a Redis database. My visualizer reads data from this database and presents it in a visual format.
### Table of Contents

1. Introduction  
   1.1. Problem  
   1.2. Solution  

2. Tools  
   2.1. Internet Information Services  
   2.2. Nxlog  
   2.3. Redis  
   2.4. C#  
   2.5. ASP.NET MVC Framework  
   2.6. JavaScript  
   2.7. JQuery  
   2.8. Google Charts API  

3. Design and Development  
   3.1. Log Storage  
   3.2. Server Side Code  
   3.3. Visualization Pages  
   3.4. Filter Partial View  
   3.5. Live Updates  

4. Use  
   4.1. Front Page  
   4.2. Custom Pie Chart  
   4.3. Custom Histogram  
   4.4. Custom Raw Data
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Log Filter</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Front Page</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Custom Pie Chart</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Custom Histogram</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Custom Raw Logs</td>
</tr>
</tbody>
</table>
List of Terms

Application Programming Interface (API): A set of rules that defines contracts for how foreign code should interact with a specific code piece.

Backend: Code for a web application that runs on a server prior to giving a web page to the user. Backend code is responsible for permanent data storage, and web request handling. Backend code may be written in a variety of languages.

Document Object Model (DOM): The DOM is a hierarchical graph of all the various elements in a web page, in the format of JavaScript objects.

Frontend: Code for a web page that is run by the browser on the user’s computer. Frontend code is written in the JavaScript programming language.

JavaScript Object Notation (JSON): A human and machine readable format for associating key-value pairs in the form of a string.

Outward-facing server: A server that is exposed to the world to accept requests from any computer connected to the internet, as opposed to a server that is for internal use only.

Response code 200: A response indicating that a request was processed successfully

Response code 404: A response indicating that the requested page was not found on the server, usually because of a misspelled address on the user’s part or because of a bad link.

Response code 500: A response indicating that the server encountered an error in the midst of processing a request.
1 Introduction

1.1 Problem

A company like FNC has a fair number of servers that all are generating data about the pages that they serve: a start-time, the duration, a specific page, a client, and a response code. The data that is generated can be used for a large number of purposes, from reporting to troubleshooting, but only if that data can be filtered to bring out the relevant information. Currently, the logs must manually be investigated to solve a problem when it is discovered, and there is no system in place that can warn anybody when a problem exists. Without a system to check for problems, such as webpages that sporadically generate response codes of 500, there is no early warning until the problem is discovered by developers, testers, or, in the worst case, end users. It becomes very expensive to operate without early warning and to have to dig through logs to get to the root of a problem when it occurs.

1.2 Solution

To fix these issues, I designed and implemented a visualization tool to display the log data in a format that can be interpreted in only a few seconds, rather than requiring a considerably longer time to dig through the logs individually. The tool displays graphs of the log data, based on a series of filters. The user is able to filter out logs from the mass of logs based on specific criteria: the time the log was created, the server that created it,
the response code generated, the client that made the request, and how long it took the server to process the request.

In addition to the different filters, there are different types of graphs. There is a pie chart to display numbers of logs that meet certain criteria. For example, a pie chart can be made that displays all logs in the last 12 hours that have taken longer than half a second to process, divided by the specific server that processed the request. The result would be a chart with a legend containing each server and the count of logs for that server.

The second graph is a histogram that shows results over time, based on a given time frame and a given set of constraints. An example of this graph is getting the number of requests for a certain page, on a certain server, from a certain client over the past week. In this graph, there are vertical bars for each couple of hours, showing the number of requests meeting the criteria in that time frame.

The final display does not focus on graphing, but rather on filtration. There are the same filters that are present in the other graphs, but for this visual, the result is simply all of the data that meet the requirements, presented in a textual format. This representation is not as sophisticated as the other visuals, but it has some advantages. The first is that it shows all aspects of the log instead of only a single aspect. The second is that it shows the data aligned by column, whereas the original log only added a single space between columns in the logs. The third reason that a text filter is helpful is that the filtration makes this tool better than looking at the original data because it cuts out all of the logs that the user does not care about, leaving only the logs that are useful at the moment.

Each of these displays updates regularly to ensure that the latest data is being displayed. Having the latest data makes the tool useful as a long-running monitor,
notifying the user of any new results as they happen, rather than relying on the user to refresh the page.
2 Tools

2.1 Internet Information Services

Internet Information Services (IIS) is a Windows Web Server that is used on almost all of FNC’s servers and all of the servers that are outward-facing. IIS accepts web requests and uses a set of rules to pass the request to where they should be processed and turned into the appropriate responses, then returns those responses the sender. For each of these requests, IIS makes an entry in a log file. Each entry in the log gets its own line in the file, and once the file reaches a certain length, the file is closed and a new log file begins.

These log files contain the raw data that we are interested in, so it is important to have a basic understanding of how these files are generated, but since the IIS servers are already set up and creating logs, there is no further interest in IIS within the scope of this project.

2.2 Nxlog

Nxlog is a tool to take the log files that are generated by IIS and send them to a central location. The specifics of how nxlog should interpret the log files and send them away are specified in a configuration file. This configuration file is written in XML and defines specifically how the data within the logs should be represented, including which columns to include, where to look for the logs, what format to represent the logs in, and
where to send the logs. Nxlog allows us to bring the logs to a central location, and to allow the data to be actively sent from each server, rather than having to query each server individually to retrieve the log data.

2.3 Redis

Redis is a database that holds data in memory, rather than storing the data in a file. The advantage of holding the data in memory is that it can be accessed much more quickly than data in a file, and that the data is more likely to be up to date. It uses key-value pairs to store data, rather than the tables and queries that SQL uses. These key-values pairs can be arranged to create complex data types that SQL is incapable of sustaining without substantial amounts of effort. Within the scope of this project, Redis is used to store the log data after it has been sent to FNC’s ticker API. The backend of the visualizer contacts the Redis database directly to get its information. In the future this Redis database can be used by other applications that need to use the log data. These potential other applications aren’t a part of the project but may use this part of the project as a base, so it is worth noting.

2.4 C#

C# is a programming language designed by Microsoft as part of their .NET framework. The .NET framework has the advantage of deep integration with most Windows services, so it is a good choice at a software company that uses mostly Windows software, servers, and products. The .NET framework also comes with a great host of libraries and plugins to make development more manageable and boasts a
powerful Integrated Development Environment (IDE). C# is object oriented and strongly typed; it has many similar features to C, Java, and others, but still has quirks of its own. This project uses C# for the backend of the visualizer. Since almost all development at FNC is done in C#, it will be easy for developers in the future to update or extend this project, since even the most well-designed program needs maintenance to handle new changes in environment or need to handle new use cases. Extensibility is an important part of programming, and using the common programming language of the client is just one part of that.

2.5 ASP.NET MVC Framework

ASP.NET MVC is a web framework with a goal to make web apps easier to create, maintain, and extend. MVC follows the development principle of separation of concerns, which says that every piece of code should have one goal and only one goal, and should not have to worry about what other pieces of code are doing. MVC stands for Models, Views, and Controllers. The models are representations of data. This data can either be coming up to the client, going down from the server, or interacting with some database. The Views represent each page that is sent to the client. In ASP.NET, Views are made of standard HTML, but C# code can be imbedded into the HTML and is evaluated right before the page is sent off, meaning that the page can hold different data each time the data is loaded. This allows for a single view with a single format to be used, but for the data displayed to vary every time the page is loaded, usually accepting data in the form of models, and putting the model data in the right parts of the web page. The final part of MVC is controllers. Controllers are the brains of MVC, when a request is
made, one of the controllers takes in the request, along with any uploaded data (in the form of the model), makes any necessary changes to permanent storage (again using models), then calls the correct view (probably sending the view a model). These three pieces work together to form the backend of a web application.

2.6 JavaScript

JavaScript is a programming language that is used primarily for frontend code on individual web pages. This frontend code serves to make a single web page more interactive by adding and removing, showing and hiding, and calculating and resetting fields as necessary. In addition to basic interactive functions, it is possible to make a request to a server and receive a response without leaving the webpage and then reincorporate the new data in that page. JavaScript is objected-oriented but loosely typed. In the scope of this project, JavaScript is used to update the graphs on each webpage, without needing to refresh the page.

2.7 JQuery

JQuery is a library for JavaScript that has a variety of useful features. One of the most noted features is the way that JQuery can be used to find and adjust Document Object Model (DOM) elements in a much more concise and programmer-friendly way than comes with basic JavaScript. Another useful feature in JQuery is Ajax, which is a unified technique for making requests to a server without leaving the page. Ajax calls are asynchronous, which means that the code that comes after a call to Ajax will be run before the Ajax call runs to completion. This means that the code that runs right after an
Ajax call needs to not expect the call to be complete. To allow for code to run directly after the Ajax call completes, functions called callbacks are passed as parameters to the Ajax call. When Ajax finishes, it will call one of the functions that were passed in, based on the response code that the server provides, and the code in that function will act on the response from the server. Asynchronous functions can make code complicated, but the ability to make calls to the server without reloading the page is worth the complications.

In this project, Ajax will be responsible for the actual updates from the server. JQuery will be used to find elements in the DOM as well and to make such changes as disabling/enabling controls as necessary.

2.8 Google Charts API

Google Charts is a JavaScript library designed to create visual graphs based on data that is passed in. Google Charts is responsible for all of the actual visualization in the project. When an API from a major corporation is used there is always a question of privacy, but the parts of Charts that are used run entirely on the browser, with no need for additional support once initial calls are made. This means that the FNC’s data stays on the client browser where it belongs and is never sent to Google’s servers, which is something FNC would like to avoid. The charts are generated quickly, even on the browser, so there is no worry that the charts will make the web pages unresponsive.
3 Design and Development

3.1 Log Storage

The first step of the log visualization process is performed by nxlog. Nxlog is a program that runs on a server and reads raw logs, sending the data to a desired location. In this case, nxlog must be configured to create JSON and send that JSON to the FNC ticker API. The ticker API will automatically place these logs in a Redis database, meaning that the logs are now available for visualization. There is not any development required for this section of the project, as applications already exist with the needed functionality. However, there are important design elements in this phase that affect development down the line, since that development will depend on the current format of the logs.

3.2 Server Side Code

The backend code is the central pillar of the project that supports all of the visualization portions of the project. The backend code obtains a filter from the client, in terms of a model. Then it uses that model to select data from Redis, based upon the information in that filter model. Once the appropriate logs have been obtained, the server digests the logs into a two dimensional array, that the client will be able to feed directly into Google Charts. The backend only uses a single controller, but uses four views within that model. In addition to the four main views, there are update actions in the main
controller that return the correct data for each kind of graph. As for models, there is a
model that represents filter settings, a model for each kind of chart return value, and a
model that represents a log from Redis.

3.3 Visualization Pages

There are four visualization pages that display log data in different formats. The
main page is an overview, and sports 4 different non-custom graphs. These graphs are as
follows: a pie chart of response codes 200, 404, and 500; a pie chart of total requests by
each kind of server; a pie chart of total requests by each client; and lastly, a histogram of
all requests for all servers in the last 24 hours. The second page allows the user to build a
custom pie chart based on any or all of the dimensions in the logs. The pie is split into
slices based on one of the log dimensions, which is also customizable. The third page has
the same filtering that the second page has, but the chart is a histogram over time. The
graph displays the count of all matching logs over a customizable time period that can be
anywhere from the last hour to the last week. The final display is not a graph but is
nonetheless useful. It displays the raw log data for all logs that meet criteria from the
same filter that was used in the previous two pages. These four pages together allow for
total transparency of the logs that FNC’s system produces. The pages are simple enough
to quickly get an idea of the health of the system, but in depth enough to allow a thorough
investigation in the case that a problem is found.
3.4 Filter Partial View

The filter is the pivotal piece in the project. It is what allows the same visual to produce a high-level overview and an in-depth investigation on the same web page, with the same tool. The filter becomes populated by the server with possible values for each column. From these possible values, the user is able to select one, several, or all of the values to be included in the search. For values that do not have a limited number of discrete values, but rather are placed along a continuum, such as response duration and time of request, it will be possible to select a maximum and minimum value for that dimension. When the user presses the filter button, all of the filter data is sent to the server and the server returns new data to populate the particular display for that page. This is the most complicated part of the project, as there are many parameters involved, but the same filter can be applied to all filtered visuals, which prevents code reuse.

The lifecycle of a filter is as follows. First, the backend of the site generates a list of all the unique values for each filterable column. For values that lie upon a continuum, only the maximum and minimum values are needed. All of these values are stored in a model that is passed into the view for the filterable page. Inside the view, each column is set up as an element or a pair of elements. The view is populated dynamically with the possible values and is sent to the client.

On the client, the filter is represented only as web elements. This web representation is depicted in figure 1. For the columns made up of discrete values, the filtering element is a modified version of a multiple select element, where each option represents a possible value that is currently represented in the available logs, with all values selected at the beginning. However, unlike a typical multiple select element,
**Figure 1: Log Filter**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min Date:</td>
<td></td>
</tr>
<tr>
<td>Max Date:</td>
<td></td>
</tr>
<tr>
<td>Min Response Time:</td>
<td></td>
</tr>
<tr>
<td>Max Response Time:</td>
<td></td>
</tr>
<tr>
<td>Min Response Bytes:</td>
<td></td>
</tr>
<tr>
<td>Max Response Bytes:</td>
<td></td>
</tr>
<tr>
<td>Min Request Bytes:</td>
<td></td>
</tr>
<tr>
<td>Max Request Bytes:</td>
<td></td>
</tr>
<tr>
<td>Server IP</td>
<td>10.15.3.151</td>
</tr>
<tr>
<td>Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request Page</td>
<td>/cms/bundles/globalize</td>
</tr>
<tr>
<td></td>
<td>/cms/bundles/jquery</td>
</tr>
<tr>
<td></td>
<td>/cms/bundles/jqueryui</td>
</tr>
<tr>
<td></td>
<td>/cms/Content/css/login.less</td>
</tr>
<tr>
<td></td>
<td>/cms/Content/fonts/meta_pro/metapro-normal-webfont.eot</td>
</tr>
<tr>
<td>Client IP</td>
<td>10.15.6.61</td>
</tr>
<tr>
<td>Response Code</td>
<td>200 302</td>
</tr>
<tr>
<td>Client Name</td>
<td>Itau+QATest</td>
</tr>
</tbody>
</table>
clicking on an option acts as a toggle, meaning that multiple items can be selected using a standard click. Usually, a standard click will unselect any selected elements and then select the clicked option, and a click while holding the control/command key will toggle the selection of that element. This change will make the filter simpler and more user-friendly. This change will be implemented via JavaScript, capturing the click and preventing it from performing its normal action. As for the columns whose values fall along a continuum, there will be a text field element for the maximum and minimum values. There will be client-side validation via JavaScript for these minimum and maximum fields such that whenever one of the values is changed there is a validation that the new value is within the given maximum and minimum values, and that after the change, the minimum is lower than the maximum. If one or both of these conditions fails, the value is changed to the most extreme value that passes both validations.

The user may interact with all of these elements at will, to achieve the desired filter. When the user is finished setting the filter to fit their needs, they press a button that serializes the filtering data into a single JSON object, and sends the JSON to the server. The root JSON object has for each column a key that holds a filter object for that column. The columns with discrete values hold a JSON array with a whitelist of values to allow through the filter. To save calculating time later, any column that has all options selected contains a JSON array that only contains the single value “#ALL#”, which should never be a valid value for any of the columns. The continuum columns have an object that contains a min, with the minimum value for that column, and a max, with the maximum value. The JSON calls an ApiController, which is a special kind Controller in MVC,
designed to allow a client to have interaction with the server and receive raw data, rather than load an entire new webpage.

On the server there is an MVC model that has the same construction as the JSON, with the discrete columns having arrays and strings, and the continuum columns having a special object that contains a max value and a min value. The final model is called a LogFilter object. The ApiController that the webpage called accepts one of these objects, and MVC automatically populates the data from the JSON into the LogFilter object. The server gets a List object from the Redis server of all the logs that are within the time frame that is specified by the LogFilter and then begins going through each column, trimming the list to only the logs that meet the requirements.

There is an issue with time complexity when it comes to the whitelisted values, as there can be a fair number of them. If there are \( n \) logs and there are \( m \) whitelisted values for this particular column, then the time complexity of filtering the column is \( O(nm) \), which is fast enough when \( m \) is small but can be incredibly slow if \( m \) gets to be large. There are two tricks that can be used to speed up the process. One is to avoid filtering altogether whenever possible. The “#ALL#” value that can be sent in the JSON means that this column should not be filtered upon, changing an \( O(nm) \) situation into an \( O(0) \) situation. The other trick that can be used to speed up the process is to filter the continuum columns first. These columns are naturally faster at \( O(n) \) complexity, and they have the added effect of making \( n \) tend to be smaller for the other filters that have yet to be run, making them run faster.

Once the logs have been filtered, the data must be condensed into a format that Google Charts will be able to directly interpret. This process varies based on the graph
being generated. Once the data is prepared, it is returned as JSON to the client. The client then uses the JSON to create a JavaScript object and passes that object to Google Charts, which generates a new Chart and replaces the old chart.

3.5 Live Updates

One of the requirements of the project is that each web page be able to perform live updates without requiring the page to be refreshed. As such, each page of the project is set up so that it refreshes its data every 15 seconds. These refreshes will allow the site to be used as a constant monitoring service. Should something happen to one of the servers, the monitor will display the new development just seconds after it begins, giving anyone who is watching immediate visibility on the issue.

Live updates are accomplished via the sameApiController that handles filter changes, except that the filter uses a stored JSON object instead of generating one from the current layout of the filter control elements. This way, if the user is in the middle of changing the settings when it is time to get a data update, the half-finished filter will not cause the results to be halfway between the old and new filter. Once the ‘submit’ button has been pressed on the filter, the new filter will be used.
4 Use

4.1 Front Page

The front page is designed with the big picture in mind. It has no settings, yet gives a multi-faceted view of everything going on in each of the servers. If one server is experiencing an extra-large load, the second chart will show this. If there is suddenly a mass of 404 response codes, the first chart will show this. If there is a time when the request load is the worst, the fourth chart will show this too. This general purpose front page is the early warning that further investigation is needed, as well as a nice way to see that everything on the servers is still running smoothly. Figure 2 depicts the front page with its four graphs.

4.2 Custom Pie Chart

The custom pie chart is designed to help discover the source of a problem once it is noticed, to determine how often a certain error code occurs for a given web page, and for general examination of the servers. On this page, a single pie chart takes up the main portion of the screen, with a filter set above the chart. Total customization through the filter means that the chart can be used in cases that were not thought of during the project, but can be adapted to use later. Figure 3 depicts the pie chart showing data by response code.
Figure 2: Front Page
Figure 3: Custom Pie Chart
4.3 Custom Histogram

The custom histogram is similar in use to the custom pie chart, except that instead of choosing a variable over which to split the pie, the user chooses a time frame over which to view results. The timeframe over which the data is displayed can be one of three values: the last hour, the last day, or the last week. It is impossible to look further back than that due to a limitation built in to the API that is used to store the log data on Redis. Specifically, all logs are removed from Redis when they reach a week old. When set to the last hour, there are thirty bars split into two minute windows. When set to a day, there are twenty-four histogram bars, each representing an hour. Finally, when set to a week, there are twenty-eight bars, each covering a six-hour interval. The histogram takes the same part of the screen as the pie chart, and uses the same filter to select results. This graph, like all the others, updates in real time.

4.4 Custom Raw Data

Graphs are great for getting an overview of a system, but to get to the heart of an issue, original logs have to be found. Rather than send the user on a hunt for the logs, this page allows users to filter logs by the appropriate variables, and then see the actual data that they care about. This page shows the same filter as the other custom pages at the top, and then shows log files, stopping at 50 if there are more than 50 logs that meet the given criteria. At the bottom of this page there is a link to a page that shows all of the results for this search, but this new page is static and does not update. Instead, the user must navigate back to the original page for automatic updates to begin again.
Figure 4: Custom Histogram

Filtered logs, over Last Day

Logs
Figure 5: Custom Raw Logs

<table>
<thead>
<tr>
<th>Date</th>
<th>Server IP</th>
<th>Method</th>
<th>Page Requested</th>
<th>Client Name</th>
<th>Client IP</th>
<th>Response Code</th>
<th>Response Size</th>
<th>Request Size</th>
<th>Time Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/bundles/globalize</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>16089</td>
<td>431</td>
<td>5678</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/bundles/query</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>93675</td>
<td>428</td>
<td>5709</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/bundles/jqueryui</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>226410</td>
<td>430</td>
<td>5959</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/Content/css/login.less</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>6248</td>
<td>394</td>
<td>15</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/Content/fonts/meta-pro/meta-pro-normal-webfont.eot</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>48963</td>
<td>416</td>
<td>15</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/images/login_bg-grid.png</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>1181</td>
<td>397</td>
<td>78</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/middlewareock/</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>2474</td>
<td>489</td>
<td>5444</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/middlewareock/Scripts/jquery-1.8.2.js</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>268468</td>
<td>414</td>
<td>15</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/Images/favicon.ico</td>
<td>-</td>
<td>10.15.6.61</td>
<td>200</td>
<td>11956</td>
<td>326</td>
<td>15</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/middlewareock/home/GetJsonFromLogin/QA_Itau_Master</td>
<td>-</td>
<td>10.15.3.150</td>
<td>200</td>
<td>1242</td>
<td>120</td>
<td>218</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/Account/HandleCadiLogin</td>
<td>-</td>
<td>10.15.6.61</td>
<td>302</td>
<td>826</td>
<td>481</td>
<td>2308</td>
</tr>
<tr>
<td>11/9/2013 10:46</td>
<td>10.15.3.151</td>
<td>GET</td>
<td>/cms/Home/Dashboard</td>
<td>Itau+QATest</td>
<td>10.15.6.61</td>
<td>200</td>
<td>40886</td>
<td>800</td>
<td>2761</td>
</tr>
</tbody>
</table>


