**Browser: Scaling X axis**

**Borja Velez**

**Ruben Martin**

**Manuel Rico**

PBA Software Development

Development of Large Systems

*13/04/2018*

Contenido

[1. Introduction 3](#_Toc511023232)

[2. Considerations and links 3](#_Toc511023233)

[3. System architecture 4](#_Toc511023234)

[4. Load Balancers 5](#_Toc511023235)

[5. Monitoring 5](#_Toc511023236)

[6. Conclusion 8](#_Toc511023237)

# Introduction

We have decided to create a search engine that indexes words from folders, upload them in a Database, and make queries to that database. The system is going to equal the words with a Thesaurus (a file with all the existing words in English) before indexing them.

If the words can’t be found in the thesaurus, it won’t send a query to the DB because it means that the word is not properly written or it does not exist in English, that way we can limit the number of queries to the DB, and we can avoid inserting wrong words into our database.

In the X-axis scaling we have decided to clone and have several identical instances of a component. In order to balance the load on the instances, a load balancer was included in the system.

The load balancer will split the load between the clone instances of the component we want to scale.

# Considerations and links

The code showed in this document and the full projects that have been developed can be downloaded from the following repositories:



<https://github.com/borjavelez/Browser> (Desktop Application)

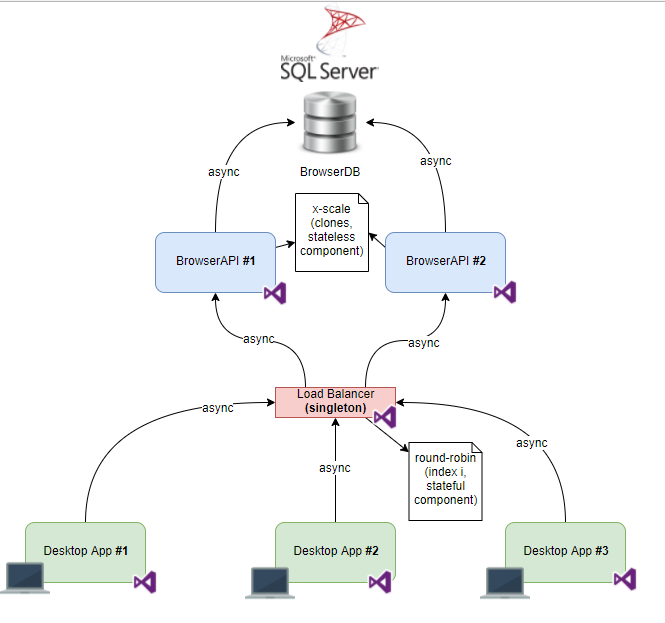
<https://github.com/Reubems/LoadBalancerSingleton> (Load balancer)

<https://github.com/Reubems/VS-API> (Monitor and Browser APIs to access DDBB)

However, for the sake of simplicity, on this documentation we will refer to the deployed resources with variables that might be changed with the paths of those components, since the current paths might not be final.

# System architecture

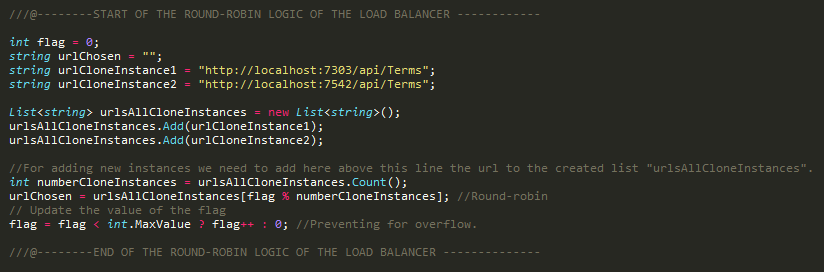
In the following diagrams we will describe how the data flow is in the app and the interaction between the components.



All the calls between the components are asynchronous. We have implemented that by using the Task Parallel Library (TPL) offered by Microsoft for C#.

There are multiple desktop applications that can send asynchronous HTTP POST requests to the load balancer.

# Load Balancer

This code will be called for every word indexed in the document.

CURRENT\_NUM\_ CLONE\_INSTANCES = 2

CLONE\_INSTANCES\_LIST = [URL\_CLONE\_INSTANCE\_#1, URL\_CLONE\_INSTANCE\_#2]

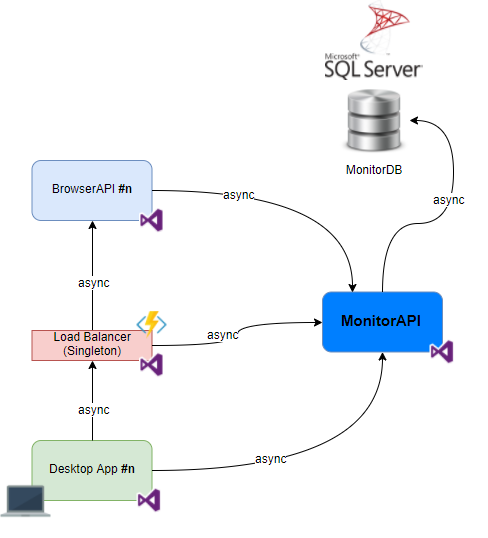
WORDS = [This, is, a, test]

|  |  |  |  |
| --- | --- | --- | --- |
| i | word | i % 2 | URL chosen |
| 0 | This | 0 | URL\_CLONE\_INSTANCE\_#1 |
| 1 | is | 1 | URL\_CLONE\_INSTANCE\_#2 |
| 2 | a | 0 | URL\_CLONE\_INSTANCE\_#1 |
| 3 | test | 1 | URL\_CLONE\_INSTANCE\_#2 |

# Monitoring

Each instance of the Browser API, DesktopApp and Load Balancer is connected to the Monitor API with an asynchronous POST request, which will stock the different messages depending on where the application has failed and at what time.

As we saw in class, we need to use a KPI to evaluate the performance, in our case the number of errors registered in the Monitor DB, it´s an indicator to see if the system is working fine or if its failing too much in relation to the amount of words inserted to the other DB in the Browser API.

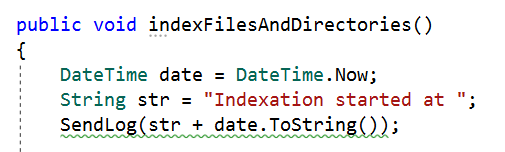
Our KPI would be the quotient between numTotalInsertedWords/numTotalRegisteredErrors. 

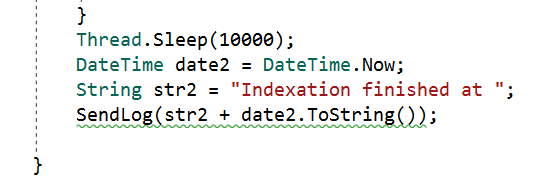
The next method is very similar in all the components: the origin property must indicate the component type and the instance number of that component (as an identifier), so that we can properly search for it in the logs of the MonitorAPI.



In order to prove that there can be several instances running simultaneously, we have launched 2 Desktop applications and indexed documents at the same period of time.

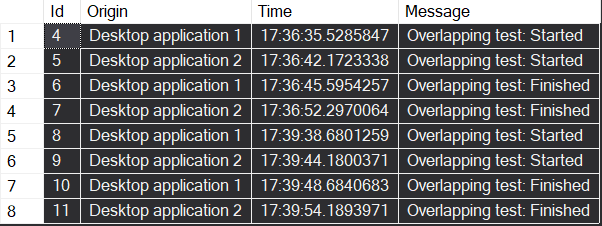
We are going to send a log when the indexation starts, and also a log when it finishes.



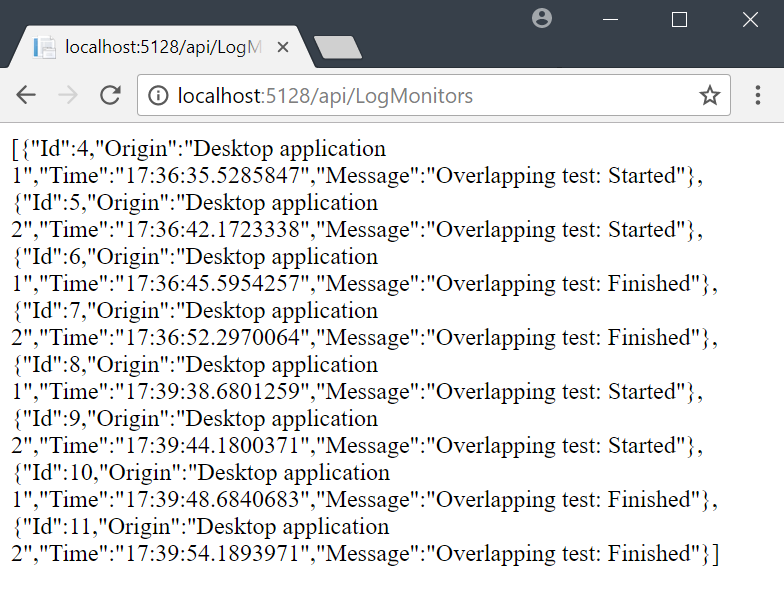


In the following screenshots we can appreciate that these time intervals are overlapped, proving that our application can manage several requests at the same time thanks to the asynchronous POST functions we have defined for communication between the components.





We can also see our MonitorAPI deployed with the results in JSON.



# Conclusion

In the development of this project we have focused mainly in how to design correctly the structure and connection between components. We have tried to follow when possible the architectural principles of large systems we saw in class.

One of them is an *asynchronous design*. In order to guarantee a high scalability in the development of large systems it is very important a right definition of how the distinct components are connected between them. For that we have implemented asynchronous functions with the TPL Library offered by Microsoft, that are called by the components and is an effective way to allow multiple instances running at the same time.

Other principle we have followed is the *design to be monitored* of the components. If something goes wrong when trying to communicate with other components or to persist in the database, we will know it thanks to the monitor component that we have designed and the function to call it, giving information about the moment, the component where it happened and a description of the error.

When we define a default number of load balancers and update the value *CURRENT\_NUM\_LOAD\_BALANCERS* either to the result of *GetNumLoadBalancersAsync()* or *DEFAULT\_NUM\_LOAD\_BALANCERS* if that method fails or does not give a result, we are following also a principle called *design to be disabled*. We do not need that component for the correct working of the application.

To prevent that when a component is down the whole application is down as well, it is good to allow different instances of the same component working at the same time. This is the purpose of the *N+1 design* principle, and we had it in mind when designing the client desktop app for the case we wanted to add more load balancers, then it would not be necessary to update that code thanks to the pattern we defined for the URLs and the component for getting the *CURRENT\_NUM\_LOAD\_BALANCERS.*

The way we have designed the monitoring logic allow us to meet also the *fault isolation* principle. For each level and component instance we can have records in the monitor database if something went wrong. These records can specify in which level of the application and when that error happened, and a brief description of what happened with the error message when for instance there is a bad request, an object is not found, etc.

Furthermore, moreover, we have made easy to have a horizontal scalability in our application with minor changes. In our design it would not be complicated at all to add new instances of load balancers or API components, we would just change a few lines of code and reuse most of the code already implemented in the same kind of component. This is related with the *scale-out* principle.

Finally, regarding the *stateless design* principle, we have some components like the APIs or the load balancers that do not have state, they just redirect HTTP requests or in the case of the APIs make calls to the DBSM. However, other components in our design do not follow that principle because of the logic of the application and to guarantee other principles we defined above. It is the case of the *GetNumLoadBalancersAsync()* component and the desktop app components. In the first one we keep the state of how many load balancers are available in the system, and in the second ones we keep the state of some variables that might change during the execution of the program, like the list of load balancers, for example.

All these principles are just a guideline about how to design large systems that we have try to follow when possible, however they are not all mandatory, and sometimes only some of them can be applied to the structure because of certain business rules in the application.

But to be aware of them has helped us a lot in order to design this project more maintainable and scalable.