ERHVERVS AKADEMI SYDVEST

Browser: Scaling X axis

Borja Velez

Ruben Martin

Manuel Rico

PBA Software Development

Development of Large Systems

13/04/2018

1. Introduction

We have decided to create a search engine that indexes words from folders, upload them into 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 cannot be found in the thesaurus, it will not send a query to the DB because it means that the word is not properly written or it does not exist in English, in 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.

2. Considerations and links

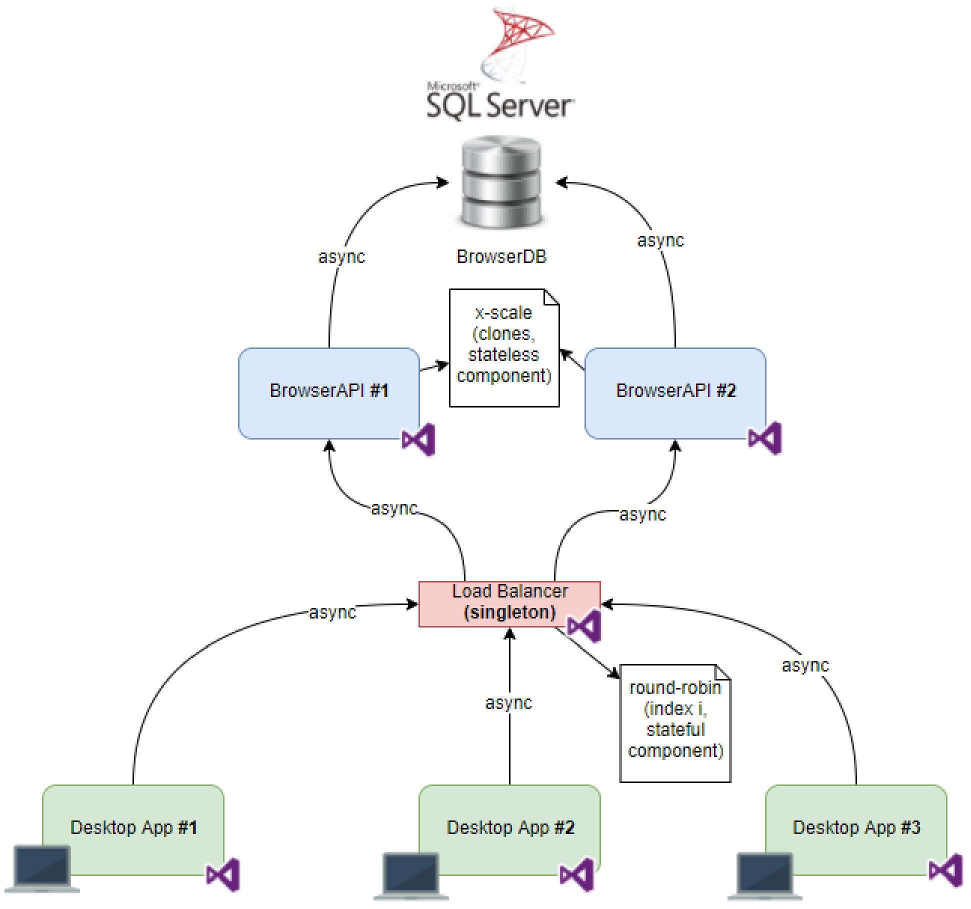
The code showed in this document and the full projects that have been developed can be downloaded from the following repository:

GitHub

https://github.com/borjavelez/BrowserDLS

3. System architecture

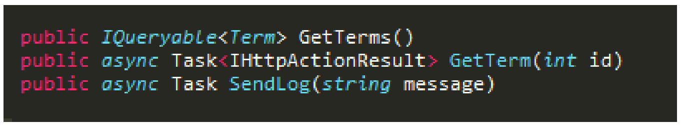
In the following diagrams we will describe how the data flow is in the app and the interaction between the components.



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 singleton instance of the load balancer.

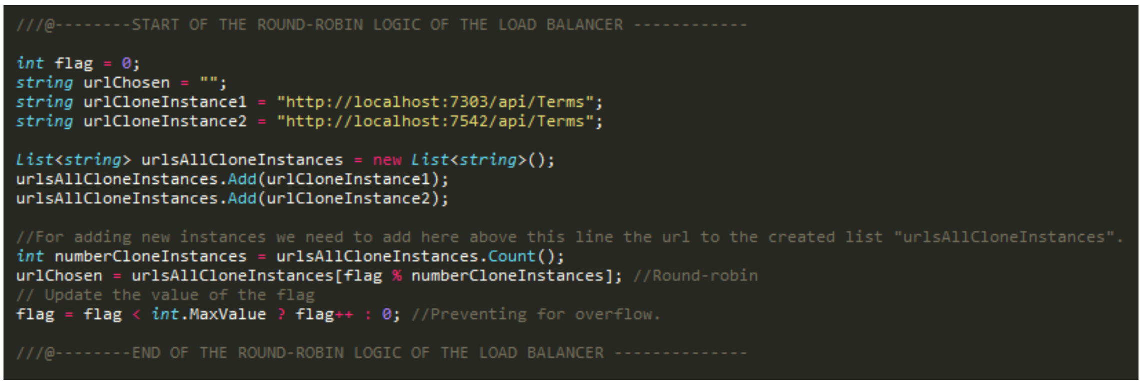
The interface of the component we want to scale and create clone instances is the following:



```
public IQueryable<Term> GetTerms()
public async Task<IHttpActionResult> GetTerm(int id)
public async Task SendLog(string message)
```

4. Load Balancer

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



```
///@--------START OF THE ROUND-ROBIN LOGIC OF THE LOAD BALANCER ------------

int flag = 0;
string urlChosen = "";
string urlCloneInstance1 = "http://localhost:7303/api/Terms";
string urlCloneInstance2 = "http://localhost:7542/api/Terms";

List<string> urlsAllCloneInstances = new List<string>();
urlsAllCloneInstances.Add(urlCloneInstance1);
urlsAllCloneInstances.Add(urlCloneInstance2);

//For adding new instances we need to add here above this line the url to the created list "urlsAllCloneInstances".
int numberCloneInstances = urlsAllCloneInstances.Count();
urlChosen = urlsAllCloneInstances[flag % numberCloneInstances]; //Round-robin
// Update the value of the flag
flag = flag < int.MaxValue ? flag++ : 0; //Preventing for overflow.

///@--------END OF THE ROUND-ROBIN LOGIC OF THE LOAD BALANCER -------------
```

Next, we have an example about how the round-robin works in our case to choose which clone instance will receive the request:

- CURRENT_NUM_ CLONE_INSTANCES = 2
- CLONE_INSTANCES_LIST = [URL_CLONE_INSTANCE_#1, URL_CLONE_INSTANCE_#2]
- WORDS = [This, is, a, test]

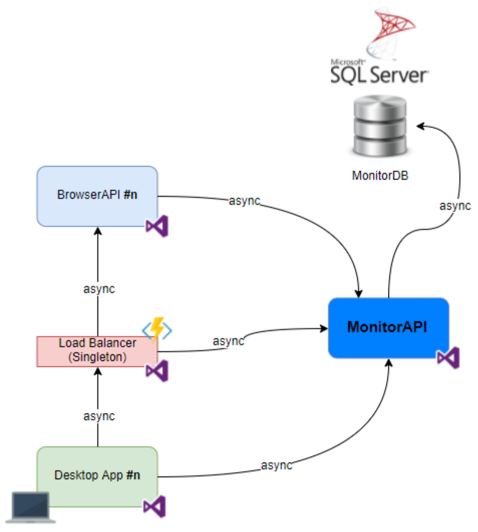
index	word	index % 2	URL chosen
0	This	0	URL_CLONE_INSTANCE_#1
1	is	1	URL_CLONE_INSTANCE_#2
2	a	0	URL_CLONE_INSTANCE_#1
3	test	1	URL_CLONE_INSTANCE_#2

5. Monitoring

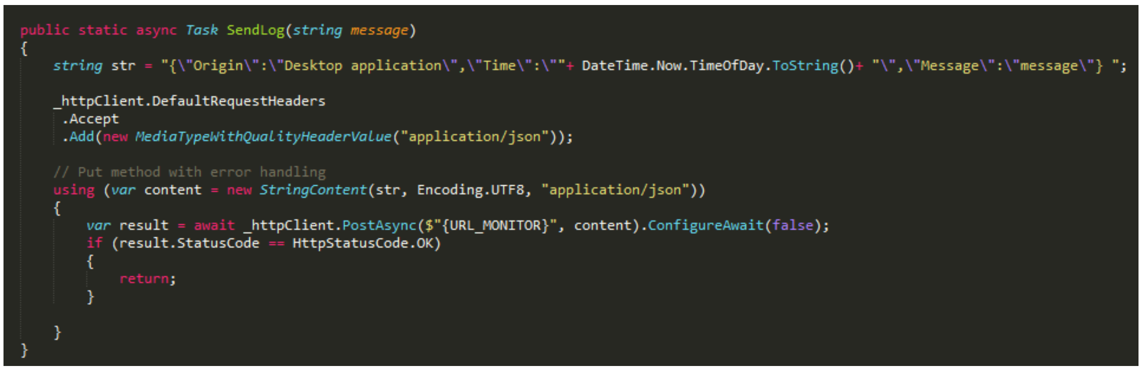
Each instance of the Browser API, DesktopApp and Load Balancer is connected to the Monitor API with an asynchronous POST request, which will stock the different messages depending on where the application has failed and at what time.

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, this is 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 DB in the Browser API.

$$\mathrm{KPI} = \frac{\mathrm{numTotalRegisteredErrors}}{\mathrm{numTotalInsertedWords}}$$



The next method is very similar in all the components and it is responsible of sending the logs to the Monitor API in case something went wrong:

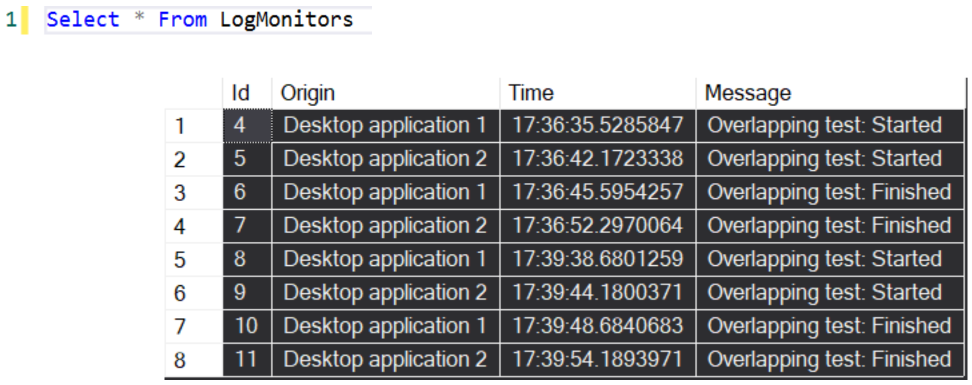


```
public static async Task SendLog(string message)
{
    string str = "{\"Origin\":\"Desktop application\",\"Time\":\""+ DateTime.Now.TimeOfDay.ToString()+ "\",\"Message\":\"message\"} ";

    _httpClient.DefaultRequestHeaders
     .Accept
     .Add(new MediaTypeWithQualityHeaderValue("application/json"));

    // Put method with error handling
    using (var content = new StringContent(str, Encoding.UTF8, "application/json"))
    {
        var result = await _httpClient.PostAsync($"{URL_MONITOR}", content).ConfigureAwait(false);
        if (result.StatusCode == HttpStatusCode.OK)
        {
            return;
        }

    }
}
```

In order to prove that there can be several instances running simultaneously, we are going to call the previous method *SendLog()* from 2 different Desktop Applications to test it.

```
public void sendLogTest()
{
    SendLog("Overlapping test: Started");
    Thread.Sleep(10000);
    SendLog("Overlapping test: Finished");
}
```

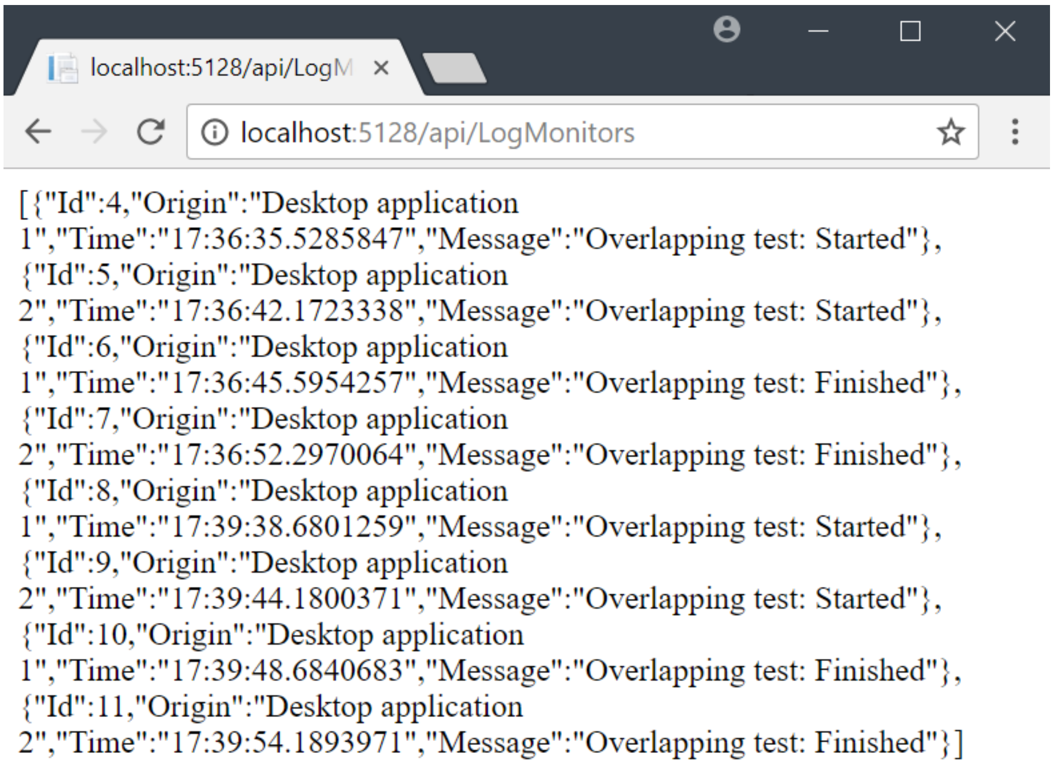
In the following screenshots we can appreciate that these time intervals are overlapped, proving that our application can manage several requests at the same time thanks to the asynchronous POST functions we have defined for communication between the components.



```sql
Select * From LogMonitors
```

	Id	Origin	Time	Message
1	4	Desktop application 1	17:36:35.5285847	Overlapping test: Started
2	5	Desktop application 2	17:36:42.1723338	Overlapping test: Started
3	6	Desktop application 1	17:36:45.5954257	Overlapping test: Finished
4	7	Desktop application 2	17:36:52.2970064	Overlapping test: Finished
5	8	Desktop application 1	17:39:38.6801259	Overlapping test: Started
6	9	Desktop application 2	17:39:44.1800371	Overlapping test: Started
7	10	Desktop application 1	17:39:48.6840683	Overlapping test: Finished
8	11	Desktop application 2	17:39:54.1893971	Overlapping test: Finished

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



[{"Id":4,"Origin":"Desktop application 1","Time":"17:36:35.5285847","Message":"Overlapping test: Started"},{"Id":5,"Origin":"Desktop application 2","Time":"17:36:42.1723338","Message":"Overlapping test: Started"},{"Id":6,"Origin":"Desktop application 1","Time":"17:36:45.5954257","Message":"Overlapping test: Finished"},{"Id":7,"Origin":"Desktop application 2","Time":"17:36:52.2970064","Message":"Overlapping test: Finished"},{"Id":8,"Origin":"Desktop application 1","Time":"17:39:38.6801259","Message":"Overlapping test: Started"},{"Id":9,"Origin":"Desktop application 2","Time":"17:39:44.1800371","Message":"Overlapping test: Started"},{"Id":10,"Origin":"Desktop application 1","Time":"17:39:48.6840683","Message":"Overlapping test: Finished"},{"Id":11,"Origin":"Desktop application 2","Time":"17:39:54.1893971","Message":"Overlapping test: Finished"}]

6. 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.

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 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 clone instances of APIs that they just redirect HTTP requests to the same database. However, other components in our design do not follow that principle because of the logic of the application (like the singleton instance of the load balancer that needs to preserve the state of the index).

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.