SYSTEM DESIGN DOCUMENT

1. Introduction

We are developing a simple HTTP-based web site with REST API that maintains an integer array, initially empty, and features the following functionality:

- Displays the array content (HTTP) or returns the current array (API)
- Provides functionality to add an integer to the array
- Provides functionality to check whether the array can be split in two, without reordering the elements, so that the sum of the two resulting arrays is equal. For example, array [4, 1, 1, 1, 1] can be split this way (in [4] and [1, 1, 1, 1]), while [1, 1, 5, 3] cannot be split in two arrays with equal sums.

2. Requirement analysis

a. Functional requirements

- The end user should be able to view the current contents of the array.
- User should be able to add a new element to the existing array.
- User should be able to check whether the current array is possible to split into two based on the sum and keeping the ordering of the array.
- Optional API to delete all the data in the existing array(an add-hoc requirement on the go.)

b. Non-functional requirements

- The system should be highly available. Every time a user tries to view/add/or split the array, they should get a valid response.
- The system should work with minimum latency.
- The system should be scalable and handle growing amount of requests from end users.
- Should have some metrics on the API consumption and trend by the users.

3. API design

We will have a dedicated Open API specification document for the exposed API's, below listed is the overview of the various API's that will be exposed in our system.

- List getArrayContent()
- void addArrayElement(number)
- List getDividedArrays()
- void deleteArray()

4. Defining data model

- List <Integer>

Since we are designing a highly available and scalable application, its convenient to use a NoSQL database for storing the data. For our purpose we will be using MongoDB.

5. High level design

The main requirement of our system is for delivering a highly available and scalable system, which consists of mainly various array operations.

Considering the requirements, we have decided to go with NoSQL(MongoDB) for our implementation. Along with that we can think of adding a cache layer to reduce the latency, which may occur in case of increased traffic.

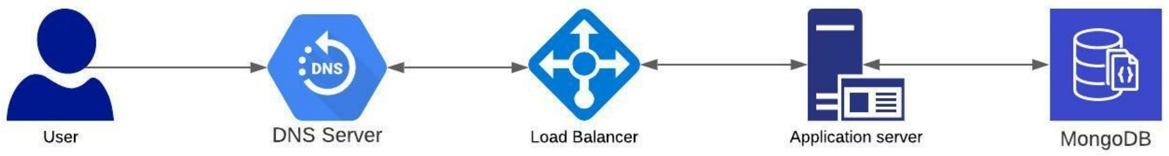

Fig-1: Simple application design

With the above solution we will be able to serve the functional requirements, in order to meet the non-functional requirements, we need to ensure high availability, scalability and lower latency. In order to achieve the same, we need to think of scaling up the application servers, location wise server hosting for lower latency, and redundant database for higher availability and an optional cache layer to further decrease the latency.

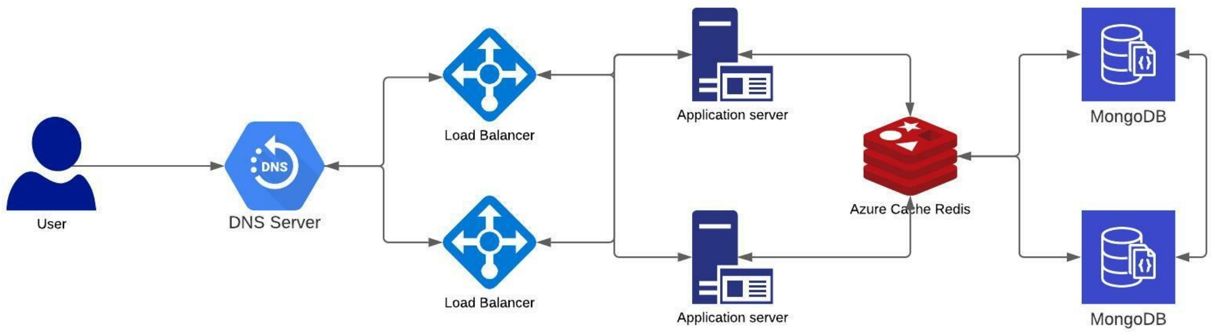

Fig-2: Scalable and Redundant application design

With this approach we can have multiple load balancer's to share the traffic so that, we will not have the bottleneck issue. Similarly we can do horizontal and vertical scaling for our application servers so that it can handle the huge traffic and it will be highly available. Similarly with the Active and passive database approach we can ensure there will be no loss of data in case of any unforeseen issues. The cache layer in between application server and database layer will increase overall latency of our application by reducing the direct hit towards the database. (Note: haven't mention about the audit logging, and API metrics handling in the diagrams, it can be handled by forwarding our application logs to some elastic search platforms like Splunk, Grafana etc or explicitly tracking via saving in database.)

6. Algorithm and implementation approach

Algorithm 1

1. Check if the array is empty or not, if empty provide respective response message.
2. Iterate over the complete array and find out the PIVOT point so that the sum of the sub arrays are equal.
   a) Make the 1st index as the PIVOT and find sum of the sub arrays and compare.
   b) Increase the PIVOT value by 1 and retry the step 2.a.
   c) Repeat steps 2.a – 2.b until we get an equality on the sum of sub arrays.
3. If we got a valid PIVOT index, split the array based on it and return the response, else error message like “Not possible to split”

With this algorithm we were able to solve the task, but the time complexity is little bit higher which made me come up with another approach which drastically reduced the time and space complexity. With the new approach we would be able to achieve the task in just O(n) time complexity.

Algorithm 2

1. Check if the array is empty or not, if empty provide respective response message.
2. Find sum of the complete array. And also find the half of the sum as the threshold value.
3. Find the MOD of the sum and see if its equally dividable.
4. Iterate over the complete array or until the calculatedSum is less than the threshold value.
   a) Calculate the calculatedSum = calculatedSum + arrayElement(i)
   b) check if calculatedSum == threshold, if its equal get the splitting point and return the sub lists.
   c) Repeat the steps 4.a – 4.b until the condition is met.