# Word counter application

Implementation:

* Created an interface exposing two methods enterWords(words) and getNumberOfOccurrencesForWord(word)
* Created a implementation class that implements the before mentioned interface and its methods
* The enterWords method takes care of any whitespaces on the ends of the word entered, converts multiple spaces between words into one space and splits the string into words in case the entered string is a multi-word string (eg. “like this”) will be sent to validation as two different terms, “like” and “this”.
* If it passes validation it gets sent to be injected into the map in an uppercase format so we can ensure case sensitivity of the entries in the map.
* Created a Translator Singleton class since I’m assuming I’ll be using its translate method to translate the word from a certain foreign word to english. (Mocked the functionality of it to return the actual word provided and to “translate” one word from Serbian to English in the implementation itself)
* Each instance of counter has its own ConcurrentHashMap<String, Integer> which is mainly used because it’s thread safe and has locking operations when updating the keys and their values.
* Upon analyzing, destructuring the words into terms and translating the words I inject the translated words into the before mentioned map as a key using parallelStream() which creates a stream of words which will run the injection logic on multiple cores (threads) which makes the operation much faster for a larger entry of words.
* If the key isn’t present in the map I default the number of its occurrences to 1 since it’s the first showing of that word in the map, otherwise I retrieve the number of occurrences and add 1 to it and put it back in the map under the same key value so the value gets overridden.
* When invoking the getNumberOfOccurrences method the user provides a word which he wants to get the number of occurrences for, and the method returns a 0 if the word is not found and the actual number of occurrences if the word is found

When it comes to memory management a ConcurrentMap is an in-memory solution which will be a good alternative to a datastore and will be almost as fast for a small and a large number of words since its access time has a time complexity of ~O(1). It ensures locking when updating entry values and has a small value access time like I mentioned before. Periodic resets or persistence of the data in the map would be needed after some time, but it depends on the use case and the number of users that will have access to the application.

Architecture solution:

If the microservice is in a multiple microservice environment, there will probably be a need to have an API Gateway that is used as a sort of middleware (single access point of the application). The logic in the API Gateway is for the developer's discretion and usually mechanisms for authentication and authorization are implemented here. Here we can implement the authentication to see if the user is trying to use the services of our library and if he/she is authorized to do so (whether it being through a JWT or an API key + secret combination). In a happy flow scenario, the API Gateway would then proxy on the request to the adequate microservice (probably based on the request URL).

The microservice itself would expose two REST API endpoints which would enter the words in the map and read the number of occurrences for each word, respectively.

As far as design patterns go, I didn’t see much of a need for many of them for such a small library. The only one that was used was the singleton pattern which I used in the library to inject the Translator that we assumed was going to be an externally provided service to translate the words for us. I’ve also used the singleton pattern in the microservice by using the @Autowired annotation since it then lets the framework inject the singleton object for me.

As far as resiliency of the microservice goes it depends on the number of users and the time the data will be stored in the map in memory. A solution to this is as I mentioned, persisting the data to a database or any other form of persistence to then flush the contents of the map in memory and re-use it again.

\*\*There will possibly be a problem if you’re trying to horizontally scale the microservice. With the current implementation, because it wasn’t stated otherwise, every microservice has its own central concurrent map. If we instantiate the microservice multiple times to assure availability, then each microservice will have its own map which will then provide non-consistent results to the user since the user could be entering words for a map in one microservice instance through the first request, and reading the map from another microservice instance he didn’t enter words for.