The Problem:

The problem we would like you to solve involves writing a rate-limited API.

1. Given the attached csv file, provide an HTTP service with 2 endpoints:
2. `/city`, that returns all the hotels belonging to a specific city
3. `/room`, that returns all the hotels that have the requested room
4. Both the endpoints can have an optional request to sort the hotels by price (ASC or DESC)
5. We need to limit the usage of the API, by limiting the rate at which the endpoints are called
6. The first endpoint can receive a maximum of 100 requests every 10 seconds
7. The second endpoint can receive maximum 1000 requests every 10 seconds
8. Both these values have to be easily configurable per endpoint, and should fallback to a default 50 requests every 10 seconds if no configuration is provided.
9. If the rate gets higher than the threshold on an endpoint, the API should stop responding for 5 seconds on that endpoint ONLY, before allowing other requests.
10. Please do not use any external library or key-value store to implement this rate limit functionality
11. Provide at least one integration test that calls your API using HTTP.

Here some extra guidelines:

1. Remember to write well styled code (show us what you can do) and have a decent test coverage
2. Provide instructions on how to compile/run your tests and API(s), and provide basic instructions on how to use the API's
3. With the submission, please add a small readme file explaining your choices and the decisions you made
4. If there are any requirements that are unclear, use your best judgement and provide documentation for your choice
5. I would prefer the solution written in either Scala, Java or Python, but also solutions in C#, C++ or F# are accepted.
6. Don't hesitate to send me any questions you might have, and please confirm that you have received this email.

Solution Approach (Algo):

Assuming the requirement is to allow x number of api calls in y seconds and to block all api calls for z seconds once the threshold is reached.

There are two ways to start thinking to rate-limit solution, the moment an API call is received:

1. Starting now, it'll be allowed to execute x number of calls in next y seconds.
2. At this point of time, does it already received x number of calls in last y seconds.

The first thought leads to a fix-span solution. By fix-span I mean that the time slots of y seconds are fixed.

The second thought leads to a flexible-span solution, where I'm not concerned about the start/finish time of a slot of y seconds.

To decide what is the correct thought to start with, I needed to understand the requirement better. I could think of 2 reasons to implement rate-limiting:

1. Resource Limitations: The api is heavy in terms of resources (may be some buffer/queue limitations or processing limitations), that ultimately restricts the number of API calls in certain time duration. So that we give it enough time to digest.
2. 3rd Party Limitations: The api might be using some 3rd party API calls internally e.g. some bank transactions and the contract has certain QPS/TPS limitations.

In both of the above cases, I could reason that the second way of thinking was needed i.e. the flexible-span, because the fix-span was allowing an API to be called 2x number of times in y seconds i.e. y seconds that include the last part of previous slot and the starting part of current slot.

To implement the solution for flexible-span, following points would guide:

1. API could be called at any point of time
2. It is to be tracked that how many api calls had been received in last y seconds.
3. All future y api calls might need the present count

Here's the brief idea about all solutions that I could think of, every solution helped as the base to reach to next better solutions:

1. A key value store where key:timestamp, and value:total number of api calls since beginning, of size y. Key-value store would have each second entry, might be done via some demon or can be done when requests are received to maintain the store. This will put extra burden on each call. Api calls can be received at random point of times, so this seems a bad choice.

2. Maintain a more complex data-structure might be involving a queue and key-value store. Key-value store for O(1) access to values, queue to easily access the latest and oldest timestamps. This would also need balancing of queue and dictionary on each call as we always want to keep the useful information in the collections. This was better than previous one as performance-wise but worse in space. Also additional complexity of a TimeLine data structure.
3. **Final:** There had to be a more elegant way to solve it, following are the ideas that pushed towards this solution: (these are some abstract thoughts, it'll get more clear in the Implementation section of this document):
    a. All timestamps could be accessed in a circular way i.e. by mod y, can be maintained in an array, where indexes are the seconds and value is the number of requests received in that particular second.
    b. There are two points in time, the time when api call received last time and the present point (when api is received). Past will always follow the Present. We'll know when the cycle is being repeated and what past information is not needed any more. The total requests count can be used to know the request received in a slot, which will be keep updating as we're sliding through the array (can be thought of like a circular queue).

A consistent concern for concurrency was always there e.g. the collection read and writes have to be thread-safe. As, while reading there shouldn't be any thread coming and incrementing the counter in the background and while writing as there shouldn't be any counter increment being lost.

Problem Part 2:

The problem has one more part, which says that all API calls should be blocked for z seconds, once the threshold of x number of calls in y seconds is crossed.

The reason of the feature I could think of was:

1. The system might need some buffer time before starting to process next API call, once the x number of requests already received in y seconds, again which might be because of resource limitations or 3rd party contract.

Solution:

This can be solved by simply keeping track of the time stamp when threshold was crossed last time.

Solution Approach (Design):

The Rate Limiting needs to be checked in each API endpoint like a decorator, which the rate limit service could be implemented as a library or completely separate service

Following qualities are needed in Rate Limiting Service:

1. Extremely Low Latency : For minimal impact on API endpoint performance, can be achieved by:
    a. Via an efficient algorithm implementation.
    b. Using in-memory cache to faster access.
2. Scalable: Consistency is a big challenge involved in a distributed environment.
    a. The service would need access to consistent, fast, distributed cache.

The first point has been taken care of in the solution, the second point would need more active discussions and drill downs to understand the requirements better i.e. the overall size of the system and the consumers of the service, which is not implemented in the solution.

Assumptions:

1. Granularity of units for time for rate-limiting is “seconds”
2. Configuration numbers are always +ve integers and time unit is seconds.
3. 429 response when API is blocked.

Implementation:

Language: C#
Tools: Visual Studio 2007 Professional Version 15.7.4
Framework: .Net core 2.1

Here's the code snippets of code implementation:

```
private bool _IsLimitedApiGetHotelsByCity()
{
    // get now without milliseconds
    var now = UnixEpoch.Now;

    lock (thisLock)
    {
        // If API has been blocked in last 5 seconds: return True
        if (ShouldBeKeptBlockedFor(now)) return true;

        // If API has been already called, 10 times in last 10 seconds: return True
        if (IsThreshholdCrossed(now)) return true;
    }

    // Don't limit the API call
    return false;
}
```

```
internal bool ShouldBeKeptBlockedFor(uint now)
{
    var timSpanSinceThreshHoldCrossedLastTime = now - _lastTimeWhenThresholdCrossed;

    return timSpanSinceThreshHoldCrossedLastTime <= _timeSpanToKeepBlockedInSeconds;
}
```

```
internal bool IsThreshholdCrossed(uint now)
{
    ushort presentIndex = Convert.ToUInt16(now % _slotSpanInSeconds);

    var secondsSinceLastRequest = now - _timeOfLastApiCall;

    if (secondsSinceLastRequest > _slotSpanInSeconds)
    {
        _totalRequestsInSlotSpan = 0;
        _requestCounts.Initialize();
    }

    if(presentIndex > _pastIndex)
    {
        _totalRequestsInSlotSpan -= _requestCounts[presentIndex];
        _requestCounts[presentIndex] = 1;
        ++_totalRequestsInSlotSpan;
    }
    else
    {
        ++_totalRequestsInSlotSpan;
    }

    _pastIndex = presentIndex;
    _timeOfLastApiCall = now;

    if(_totalRequestsInSlotSpan > _maxRequestsInSlotSpan)
    {
        --_totalRequestsInSlotSpan;
        _lastTimeWhenThresholdCrossed = now;
        return false;
    }

    ++_requestCounts[presentIndex];

    return true;
}
```

Installations:

1. Clone the repo from https://github.com/Vikas-Kaushik/AgodaAssignment
2. Configure values in startup.cs
3. File: AgodaAssignment\RateLimitService\RateLimitService.cs has the core logic implemented and there are some assertions in \AgodaAssignment\RateLimitServiceTest\RateLimitServiceShould.cs to validate its working.
4. Install visual studio code
5. Install dotNet sdk 2.1
6. Open the solution folcer in Visual Studio Code
7. Press 5
8. Hit: api endpoint: /api/hotels/city/bangkok