**Promises Exercise:**

The following github repository is implementing a server which tries to send 3 requests to a different server, receives a number and aggregates the results:

<https://github.com/tamarstern/requestAggregatorNoPromises>

The aggregation is done with async parallel.

Pay attention that 2 tests fail –

* Both of the tests that fail mock an error received from the url we are sending requests to. Async parallel, while receiving an error, stops the execution of the tasks immediately. In that case, the server will stop to aggregate the results and will return the aggregated number he had collected so far.

Your Task is:

In your task you are going to write the request aggregator in several different ways in order to practice the different methods that we had studied in the previous class. In each step you will need to write a different implementation of the request aggregator.

1. Re-write the async parallel to use promise chaining, make sure all tests pass as your success criteria. The server should behave as follows :
   1. send the requests one after another
   2. aggregate the results
   3. Handle errors – if one of the requests fail – continue aggregate the results from the other requests.
2. Re-write the promise chaining to use promise.all(). Make sure all the tests pass.
   1. Pay attention – if there is an error in resolving one of the promises, promise.all execution will stop and the promise that it returns will be rejected. Handle this issue – meaning that
      1. Use promise.all for parallel execution
      2. Make sure exception is handled, and if one or more requests fail, you continue to execute the other requests and collect the results.
3. Re-write the implementation of promise.all to get read of another thing that needs to be fixed in the implementation :
   1. The variable ‘aggregateRes’ is located in the function of the requests aggregator, and our inner functions are using it.
      1. Re-write the code to use more elegant solution. Consider the reduce method of array for example and see how it can be integrated with promises.
4. Use promisifyAll :
   1. Create a new module in a separate file.
      1. To this new module add the request aggregator.
      2. Make sure that all the API exposed by the new module that you created is written with callbacks.
   2. Implement the same functionality using promisifyAll to the new module and promise.all:
      1. Import your new file to the controller of the request aggregator.
      2. Use promisifyAll to wrap the API with promises.
      3. Implement the requests aggregator using this functionality.
      4. Make sure that you still catch the errors and continue to collect the results from the requests which succeeded.
5. Let’s generalize!
   1. In the assignment you had to implement a module that sends 3 requests. Now let’s be more generic and create the possibility to execute N requests.
      1. Change the API to receive a parameter n – which will be the number of requests. If the parameter is not transferred – return http code 500 with an appropriate message.
      2. Write tests that verify that the parameter is transferred – and that the server returns 500 if it is not transferred.
      3. Generalize the implementations of section 1, 2, 3 and 4 to N requests – write 2 different implementations.
6. After the generalization – let’s implement concurrency.

What is concurrency here?

So in general – that means that no more than x promises will be able to resolve in parallel, when x represent the concurrency level.

A concrete example:

Say that I have 10 promises and my concurrency level is 3.

Only 3 promises can be resolved simultaneously.

Since I have 10 promises, 3 will start to run, and more will become active only when one of the running 3 will finish (or – resolved).

You can read more about this feature here, in bluebird map – and look at the concurrency level explanation.

<http://bluebirdjs.com/docs/api/promise.map.html>

Your last mission is to implement concurrency on the request aggregator.

Transfer a concurrency level M which will be smaller than N (the number of requests) and execute the requests using the concurrency limit.