**Generators Exercise:**

**Request Aggregator:**

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:

1. Re-write the async parallel to use generators – 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.
   4. Put the actual send of the request as a different module , in a different script, and use it with ‘require’ in your controller.
   5. Use promisifyAll and coroutine functions of bluebird.
2. Add the number of requests send as a parameter for the request and re-write the generator to receive a number N and send N requests.
   1. Use
   2. Make sure that if number of requests is not transferred :
      1. The server returns 500 internal error with appropriate message.
      2. Add test that verify that.
3. Add a feature to the code :
   1. Add the possibility to send every request to a different url.
   2. Use generators messages to achieve this functionality.
   3. Send a parameter to the request – and array of urls.
      1. If this array is not transferred – send 500 internal error + message.
      2. If the size of the array is not equal to the number of requests N – return 500 internal error + appropriate message.
      3. Add tests to cover those cases.

Attention – Your code should look as much as possible like synchronous code. The goal of this exercise is to generate a code that looks as synchronous as possible – in the aspects of syntax and error handling – but will be asynchronous behind the scenes.

**Background Request Processing**

Now, after we had done request sending synchronously, let’s move them to a background job which will execute them.

For that task, you can start a new Node.js project.

**Your Task Is:**

Step One: Every request is sent once

* Create a model in MongoDB for the request execution. The model will have the following fields :
  + The url to send the request – note, for simplicity we will use only the http verb get.
  + Execution status – by default will be pending.
  + Request result – by default will be null. The result will always be an integer.
* The controller will expose the following http verbs :
  + Post – post new request execution
  + Get by Id – will get a record by Id
* Implement a background job using setTimeout :
  + The background job will call a generator.
  + The generator will read from database records with status which is pending
  + The request will be executed using promises (choose which method you like from the previous lesson)
  + Once the request will return, the appropriate record will be updated to status “Done” and the result will be updated in the result field.
* Note :
  + Implement the generator in the background job so that it will always return results – meaning, that every call to next will work on more pending records from the database if exists. (Hint – you can use while true loop)
  + Implement as many tests as you can.

Step 2: Every request will be executed several times:

* Add a new field to the schema which will represent the number of times to execute each request.
* Change the request sending code in the background job to work with generators too - executing several requests mechanism will be implemented using generators.
* At the end you will have two generators calling one another :
  + First for reading from DB
  + Second for executing the requests
* Cover your code with as many tests as possible.