Building a NodeJS Library

Below is a code challenge where you are going to build a library. No, not a code library. A fully fledged, real library with books. Well, a *virtual* fully fledged real library.

The library is going to be accessed by a set of RESTful services. You are free to choose your backend NPM packages of choice but using NodeJS is a requirement. We strongly suggest using Express as well, and keeping things as simple and well organized as possible. The main goal of the challenge is to assess your skills at the following:

- JavaScript and asynchronous programming
- Web services
- Unit Testing

The basic requirements are as follows:

1. The server should operate locally and be accessible at the http://localhost URL.
2. No data persistence is required as part of this challenge. All data can be kept in memory.
3. All the services to be written as part of the challenge should include test cases. The test cases should account for success and failure scenarios and cover all methods.
4. As mentioned above, no external libraries are required.
5. Any enhancements beyond the requirements included here should be marked as such, and reasoning should be provided for why the enhancements were made. However, the API should remain strictly adherent to the requirements present here.
6. Appropriate errors should be thrown whenever an exception is encountered.

The following are the expected API methods:

POST

This method is used to add a book to the library. Books should be kept in memory as an array of strings containing the book titles. No duplicate titles are allowed. New books are added to the end of the array. The title of the book to be added is passed to this method in the body of the POST request with a parameter called "`book`".

DELETE

This method is used to remove a book from the library. Errors should be thrown for attempted removal of non-existent books. If a book is removed, all subsequent books are shifted up by 1 index. The body of this DELETE request should contain a "`book`" parameter with the name of the book to be removed from existence.

PATCH

This method updates the name of an existing book. Errors should be thrown for attempts to update non-existent books, or if the updated name would match the name of another book already in the library, to avoid confusion (or an existential book crisis.) The index of the book should stay the same after its name has been updated. The request body should contain two parameters, "`original_book`", the initial name of the book to be updated, and "`new_book`", the new name of said book.

GET

This method returns the full contents of the library. No parameters are needed. The following requirements should be met by the code however:

- There should be an asynchronous function called "`getBookList`"

- The function should have the following prototype:
  `function getBookList (list, index, callback, [, . . .])`
- `list` : the first parameter is the book array
- `index` : the second parameter is an integer index
- `callback` : the third parameter is a callback function
- `[, . . .]` : other parameters are permitted at your discretion

This function should **recursively** read the books in the library, providing a string with all the books delimited by a separator character of your choice as the parameter to the callback function.

The service should return in the response body the string evaluated by the "`getBookList`" function.

PUT

This method should simulate asynchronous persistence of the current book list to a database (no actual saving of the book list to a database is required.) A function called "`saveItemOnDatabase (name, callback)`" should be defined such that the first parameter is the name of the book to be saved and the second is a callback function. This function should be called for every book in the list separately.

To simulate a database delay for every write operation, you can use the native JavaScript "`setInterval`" function in combination with "`Math.random()`" and the book name string's "`length`" property. If you like, you can write individual files for each book using the NodeJS File System module, but only asynchronous methods are accepted as part of the solution.

The request main thread should wait for the callback responses of every call to "`saveItemOnDatabase`", and it should be responsible for the synchronization of these asynchronous events. After all the books are "saved to the database", a response should be sent to the PUT request. The response should contain a JSON object where each key is a book name, and each value is an associated integer value that is the number of milliseconds elapsed from the beginning of the main request until the moment the callback related to that particular key/book was called. As an example, please see the following sample response:

```
{
      "1984": 233,
      "A Christmas Carol": 506,
      "Moby Dick": 708,
      "The Hitchhiker's Guide to the Galaxy": 1476,
      "The Lord of the Rings": 1091
}
```

Test Cases

The final solution should include a set of test cases for each method of the API, and each success or failure scenario. The test cases should be organized as to be easily verifiable during a code walkthrough.

Final Consideration

There are no restrictions on the use of libraries or frameworks as mentioned above, but as it is a simple challenge, we recommend avoiding external libraries as much as possible, as it can increase the time needed for analysis and development, and the complexity of the solution. Questions while trying to solve this challenge are allowed.