Server (shell) Exercise

General

Submission date: 3.6.24 | Submission mode: Single | Exercise weight: 35%

In this exercise, you will implement an HTTP server that can respond to incoming (HTTP) requests.
As part of the exercise, you will be given a description of how the server operates, what the endpoints are and their structure (both request and response), and how they should behave.

You can write your code in every tech spec you wish. You will need to supply an executable file through which your application (i.e. the server) will be invoked.

As part of this exercise, you will need to work with a server shell framework. You need to investigate this topic independently, and you can select any framework that will suit your needs and exercise requirements.

Some of the data you will need to send/receive to/from the server will be in JSON format. We recommend seeking and using 3rd party libraries that help consume and produce JSON data format.

There is no need to consider, or support, multiple threading behaviors in this exercise.
You can assume requests will be invoked serially, one after another.

Your code will be tested by an automation tool that invokes HTTP requests and will expect proper responses from your server.
As such, it is crucial to stick to the instructions and follow them precisely.

You are encouraged to test your implementation using Postman or any other HTTP Client you like… (e.g., ex2 ?)

The Exercise

You will need to write a server that supports a books store and helps managing its inventory

Each Book has the below properties

- **Id:** a unique id which is a simple integer counter, starting from 1,2,3.. (that is, the id of the first book is 1)
- **Title**: The book's title. String. Can contain spaces
- **Author**: The book's author. String. Can contain spaces
- **Print year**: a 4-digits of the year the book was printed. Integer
- **Price**: Book's price. Integer (Positive)
- **Genre :** list of Book's genres. The genre can be any of the below values (strings; capital case)
  ["SCI_FI", "NOVEL", "HISTORY", "MANGA", "ROMANCE","PROFESSIONAL"]

The server will enable basic CRUD operations: Create, Read, Update and Delete Books from the inventory.

All endpoints (except the health, see below) will return the same result in a common json structure:

```
{
        result: <result of operation> : depends on the context
        errorMessage: <message in case of error> : string
}
```

For any response, only one of these fields is filled with data; the second one is ignored and can be omitted.

In the rest of this document, the result will mean the result attribute on the response object; the errorMessage means the errorMessage on the response object.

The signs <...> represent placeholders for data that you should put. They are NOT part of the expected result.
All names of endpoint, query parameters, and json attributes are case sensitive.

The server will listen on port **8574 (i.e., localhost:8574/…).**
The automation expects your server to come up in a reasonable time: MAX 8 seconds.
You should test and verify the uptime of your server process.

The endpoints:

1. **Health**
   This is a sanity endpoint used to check that the server is up and running.
   Endpoint: **/books/health**
   Method: GET
   The response will be 200, and the result is the string **OK** (not a json, simply the string itself, case sensitive)

2. **Create new Book:**
   Creates a new Book in the system.

   Endpoint: **/book**
   Method: **POST**
   Body: json object:
   {
   title: <Book title>, //string
   author: <Book author>, //string
   year: <print year>, // integer
   price: <Book price>, // integer
   genres: <Book genres> // json array of strings
   }

   When a new Book is created, it is assigned by the server the next id in turn.

   Upon processing, you need to verify if:
   1. Is there already a Book with this **title** (Books are distinguished by their title). Comparison is case **insensitive** (that is: 'The Partner' is the same as 'tHE pARTNER')
   2. Year limits: 1940 <= year <= 2100

   If the operation can be invoked (all verification went OK): the response code will be 200;
   The result will hold the (newly) assigned Book number

   If there is an error, the response will end with 409 (conflict); the errorMessage will be set according to the error:
   - Book already exists:
     "Error: Book with the title [<book title>] already exists in the system"
   - Year is not in range:
     *"**Error: Can't create new Book that its year [<book's year>] is not in the accepted range [1940 -> 2100]**"*
   - Price must be positive:
     *"**Error: Can't create new Book with negative price**"*

3. **Get Books count**
   Returns the total number of Books in the system, according to optional filters.

   Endpoint: **/books/total**
   Method: **GET**
   Query Parameter:
   - **author**: all books written by that author. Matching is full and case insensitive
   - **price-bigger-than:** number. All books that their price is bigger or equals (>=) to the given price
   - **price-less-than:** number. All books that their price is less or equals (<=) to the given price
   - **year-bigger-than:** number. All books that their year is bigger or equals (>=) to the given price
   - **year-less-than:** number. All books that their year is less or equals (<=) to the given price
   - **genres:** All books that have any of the given genres. Genres will be given in capital case only, in CSV format i.e. ROMANCE,PROFESSIONAL

   All query parameters are optional.
   In case none of them exist, returns the total of all books in the system.
   In case more than one of them exists, perform AND operator between them
   If the **genres** values are not precisely the above options, case sensitive, the response will end with 400 (bad request)

   The response will end with 200; The result will hold the actual number of Books according to the filter

4. **Get Books data**
   Returns the content of the books according to the given filters as described by the total endpoint

   Endpoint: **/books**
   Method: **GET**
   Query Parameter: as described in the **total** endpoint above.

   The response will be a json array. The array will hold json objects, each describing a single book.
   Each Book object holds:

```
{
    id: integer,
    title: string,
    author: string,
    price: number,
    year: number,
    genres: json array of strings,
}
```

   All query parameters are optional.
   In case none of them exist, it returns a list containing all books in the system.
   In case more than one of them exists, perform AND operator between them
   If the **genres** values are not precisely the above options, case sensitive, the response will end with 400 (bad request)

   The array will be sorted according to the book's title, case **insensitive**.
   You can assume stable sorting.
   The sorting will always be ascending (that is, from a to z).
   If no Books stand in the filter requests, the result is an empty json array.

This response code is 200;
The result will hold the json array as described above.

5. **Get single Book data**
   Gets a single book's data according to its id
   Endpoint: **/book**
   Method: **GET**
   Query Parameter: **id**. Number. The Book id

   The response will be a json array. The array will hold json objects, each describing a single book.
   Each Book object holds:
   {
   id: integer,
   title: string,
   author: string,
   price: number,
   year: number,
   genres: json array of strings,
   }

   This response code is 200;
   The result will hold the json array as described above.

   If no such Book with that id can be found, the response will end with 404 (not found); the errorMessage will be set according to the error: ***"Error: no such Book with id <book number>"***

6. **Update Book's price**
   Updates given book's price

   Endpoint: **/book**
   Method: **PUT**
   Query Parameter: **id**. Number. The Book id
   Query Parameter: **price**. The new price to update. A number.

   If no such Book with that id can be found, the response will end with 404 (not found); the errorMessage will be set according to the error: ***"Error: no such Book with id <book number>"***

   If the price is less than 0 (<=) the response will end with 429 (conflict). the errorMessage will be set according to the error: ***"Error: price update for book [<book number>] must be positive integer"***

   If the Book exists (according to the id), and the price is valid, the price gets updated.
   The response will end with 200. The result is the old price of the Book.

7. **Delete Book**
   Deletes a Book object.

   Endpoint: **/book**
   Method: **DELETE**
   Query Parameter: **id**. Number. The Book id

   Once deleted, its (deleted) id remains empty (that is, the next Book that will be created DOES NOT take this id)

   If the operation can be invoked (the Book exists): the response will end with 200; The result will hold the actual number of Books left in the system, after the deletion.

   If the operation cannot be invoked (Book does not exist): the response will end with 404 (not found); the errorMessage will be: ***"Error: no such Book with id <book number>"***

You can test yourself using the supplied Postman collection that runs through some (not all !) of the endpoints and includes tests that expect certain results. Verify you pass all tests !
This collection requires setting a postman environment holding a ***port*** variable. You can find it here.

What to Submit

You should submit a zipped file that holds your implementation along with a file called **run.bat**.
The file should hold all the commands to run and execute your code.
Make sure you follow these guidelines when writing your **run.bat** file.
Your submission should also include the source files of your project.

The automation runs in an isolated environment, so ANY 3rd party dependencies can NOT be downloaded as part of your run.bat (e.g., pip install, npm install, etc.).
As your code relies on other 3rd parties to run, you should prepare them beforehand and place them on the zip file itself (e.g., node_modules in Node JS or create a self-contained executable in python..).

You can zip your files using zip format ONLY. (**DO NOT** use rar, 7z, arj, gzip, cpbackup, or any other means)
Please note that your zip is expected to be 'flat'; all files are located directly inside it, not within an inner nested folder or something. (that is, after unzipping your submission, the folder will contain the **run.bat**).
You can view this short movie demonstrating how the zip file should look alike.