Samurai e-Book Store

Problem Statement

The country is currently hosting a group of exceptionally well-known investors, and they are currently having discussions with the country's top IT policymakers about the possibility of establishing an investment framework. After seven days, they will participate in a program allowing them to examine several projects. Over the next two days, those responsible for formulating national IT policy are soliciting suggestions from individuals who are interested in receiving them. You are a young entrepreneur who is enthusiastic about starting a business, and have an idea for an e-book sales platform. However, at this point, you need to develop the minimum viable product possible and demonstrate progress. The only thing that investor groups are interested in seeing is the backend platform. For the purpose of this project showcase selection, you have one day to create a high-quality backend application for your e-book sales platform concept.

This project will involve the designing and implementing a backend system for a new e-book platform that intends to transform how users discover digital content, access it, and interact with it. For the purpose of providing consumers with an immersive and individualized reading experience, the platform should be scalable, secure, and loaded with extensive features. However, it is well addressed that this notice was given one day before, and you will have to design a backend API prototype very quickly. For the next few hours, you will be developing the following APIs:

- Add books,
- Update books,
- Fetch books,
- Fetch a particular book,
- Search books by their title, author name, and genre.

The following section will describe details of the required API endpoints to be developed:

API Endpoint Specifications

It is assumed that your API will be hosted in your local machine at 5000 port. So, all your APIs will have a host prefix of http://localhost:5000.

Also note that all field values will be compared in a case-sensitive manner, as shown in this problem specification. Do not change formatting; otherwise you will fail the judge tests. The judge will not care about space, line breaks, or the order of fields within the JSON output as long as the JSON output is consistent.

The judge will ignore any additional fields in your JSON response. The judge will only consider the fields described in this problem statement.

Add Books

User can add a book to the bookstore. This will be a POST request that will send details about the book. The API is responsible for storing the book in the database and successfully responds to the user.

Please see the following specification for the request-response details.

Request Specifications

URL: /api/books

Method: **POST**

Request model:

```
{
  "id": integer,         # A numeric ID
  "title": "string",     # A book title string
  "author": "string",    # A book author string
  "genre": "string",     # A genre string
  "price": float         # A real number price
}
```

Success Response:

Upon successful operation, your API must return a 201 status code with the saved book object.

Response status: 201 - Created

Response model:

```
{
  "id": integer,         # A numeric ID
  "title": "string",     # A book title string
  "author": "string",    # A book author string
  "genre": "string",     # A genre string
  "price": float         # A real number price
}
```

Failure Response:

For this API, you do not need to produce any failure response. You can asssume we will not provide an ID that is already in the database.

Examples

Let's look at some example requests and response.

Example request:

- Request URL: [POST] http://localhost:5000/api/books
- Content Type: application/json
- Request body:

```
{
  "id": 1,
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "genre": "Fiction",
  "price": 19.99
}
```

Example success response:

- Content Type: application/json
- Response HTTP Status Code: 201
- Response body:

```
{
  "id": 1,
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "genre": "Fiction",
  "price": 19.99
}
```

Update Books

Users can update an existing book on the e-book store. This will be a PUT request that will send details about the book. The API is responsible for updating the book in the database and responds to the user with success. The API must return an error response if the book does not exist. Note that the stored book object must be replaced by the new book details sent by the request.

Please see the following specification for the request-response details.

Request Specifications

URL: /api/books/{id} - Here, `id` is the book's ID we want to update.

Method: **PUT**

Request model:

```
{
  "title": "string",    # A book title string
  "author": "string",   # A book author string
  "genre": "string",    # A genre string
  "price": float        # A real number price
}
```

Success Response:

Upon successful operation, your API must return a 200 status code with the updated book object.

Response status: 200 - Ok

Response model:

```
{
  "id": integer,         # A numeric ID
  "title": "string",     # A book title string
  "author": "string",    # A book author string
  "genre": "string",     # A genre string
  "price": float         # A real number price
}
```

Failure Response:

If the requested book does not exist, respond with a 404 status code and an error message.

Response status: 404 - Not found

Response model:

```
{
    "message": "book with id: {id} was not found"   # Replace {id} with
the reuqested book id
}
```

Examples

Let's look at some example requests and response.

Example request with an id that exists:

- Request URL: [PUT] http://localhost:5000/api/books/1
- Content Type: application/json
- Request body:

```
{
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "genre": "Fiction",
  "price": 24.99
}
```

Example success response:

- Content Type: application/json
- Response HTTP Status Code: 200
- Response body:

```
{
  "id": 1,
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "genre": "Fiction",
  "price": 24.99
}
```

Example request with an id that does not exist:

- Request URL: [PUT] http://localhost:5000/api/books/25
- Content Type: application/json
- Request body:

```
{
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "genre": "Fiction",
  "price": 24.99
}
```

Example error response:

- Content Type: application/json
- Response HTTP Status Code: 404
- Response body:

```
{
  "message": "book with id: 25 was not found"
}
```

Fetch Book by ID

Users can fetch an existing book by its ID from the e-book store. This will be a GET request. The API is responsible for fetching the book from the database and successfully responding to the user. The API will respond with an error message if no book is found with the specified ID.

Please see the following specification for the request-response details.

Request Specifications

URL: /api/books/{id} - Here, `id` is the book's ID we want to fetch.

Method: **GET**

Request body: This request will not have any request body.

Success Response:

Upon successful operation, your API must return a 200 status code with the saved book object.

Response status: 200 - Ok

Response model:

```
{
  "id": integer,         # A numeric ID
  "title": "string",     # A book title string
  "author": "string",    # A book author string
  "genre": "string",     # A genre string
  "price": float         # A real number price
}
```

Failure Response:

If the requested book does not exist, respond with a 404 status code and an error message.

Response status: 404 - Not found

Response model:

```
{
    "message": "book with id: {id} was not found"   # Replace {id} with
the requested book id
}
```

Examples

Let's look at some example requests and response.

Example request with an ID that exists:

- Request URL: [GET] http://localhost:5000/api/books/1
- Request body: None

Example success response:

- Content Type: application/json
- Response HTTP Status Code: 200
- Response body:

```
{
  "id": 1,
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "genre": "Fiction",
  "price": 24.99
}
```

Example request with an id that does not exist:

- Request URL: [GET] http://localhost:5000/api/books/25
- Request body: None

Example error response:

- Content Type: application/json
- Response HTTP Status Code: 404
- Response body:

```
{
  "message": "book with id: 25 was not found"
}
```

Fetch all books

Users can fetch all existing books from the e-book store as a list. This will be a GET request. The API is responsible for fetching all books from the database and successfully responding to users. The books should be sorted in their IDs' ascending (increasing) order.

Please see the following specification for the request-response details.

Request Specifications

URL: /api/books

Method: **GET**

Request body: This request will not have any request body.

Success Response:

Upon successful operation, your API must return a 200 status code with the saved book objects, sorted in the ascending order of their IDs. The list must be wrapped with a `books` object in the response json, even if

only one book exists. If no book is found, the `books` field will return an empty list.

Response status: 200 - Ok

Response model:

```
{
  "books": [                # A list object with 0 or more books
    {
      "id": integer,        # A numeric ID
      "title": "string",    # A book title string
      "author": "string",   # A book author string
      "genre": "string",    # A genre string
      "price": float        # A real number price
    },
    {
      "id": integer,
      "title": "string",
      "author": "string",
      "genre": "string",
      "price": float
    },
    ...
  ]
}
```

When no books are found, the response body will look like the following:

```
{
  "books": []
}
```

Note: This is still considered a successful response.

Failure Response:

For this API, you do not need to produce any failure response.

Examples

Let's look at some example requests and response.

Example request:

- Request URL: [GET] http://localhost:5000/api/books
- Request body: None

Example success response:

- Content Type: application/json
- Response HTTP Status Code: 200
- Response body:


```
{
  "books": [
    {
      "id": 1,
      "title": "To Kill a Mockingbird",
      "author": "Harper Lee",
      "genre": "Fiction",
      "price": 19.99
    },
    {
      "id": 2,
      "title": "Go Set a Watchman",
      "author": "Harper Lee",
      "genre": "Drama",
      "price": 24.99
    },
    {
      "id": 3,
      "title": "To Kill a Mockingbird: A Graphic Novel",
      "author": "Harper Lee",
      "genre": "Graphic Novel",
      "price": 14.99
    }
  ]
}
```


Search books

Users can search and filter using various criteria. Users can also provide a desired sorting order. This API is responsible for searching the books based on provided search filters and sorting them using the provided sorting order. Your API should be able to receive query parameters as listed below.

Search fields:

1. title
2. author
3. genre

At most, one search field will be present in the query. When no search field is provided, the API should respond with all books found in the database. Searching should only fetch book objects with the fields that are matched as is. Do not perform fuzzy or partial or case-insensitive searches.

Sorting fields:

1. title

2. author
3. genre
4. price

At most, one sorting field will be present in the query. If present, this will be the primary sorting criterion. When no sorting field is provided, assume to sort based on book ID. In tie-breaker situations, you should also use ascending order on ID field as a secondary sort criterion. For example, when you are sorting based on `genre`, but there are 3 entries with the same genre, sort them in the ascending order of their IDs.

Sorting orders:

1. ASC - ascending
2. DESC - descending

If no sorting order is provided, assume to sort in ascending order.

Request Specifications

URL: /api/books?{search_field}={value}&sort={sorting_field}&order={sorting_order} - Query parameters and their values are described above.

Please note that all query parameters are optional. All can be present, one or more can be present, or none can be present. Your API should be able to handle all possible forms of search requests. Also note that when no query parameters are provided, this API is essentially the same as the "Fetch All Books" API.

Method: **GET**

Request body: This request will not have any request body.

Success Response:

Upon successful operation, your API must return a 200 status code with the searched book objects, sorted according to the described sorting mechanism. The list must be wrapped with a `books` object in the response json, even if only one book exists. The `books` field will return an empty list if no book is found.

Response status: 200 - Ok

Response model:

```
{
  "books": [               # A list object with 0 or more books
    {
      "id": integer,       # A numeric ID
      "title": "string",   # A book title string
      "author": "string",  # A book author string
      "genre": "string",   # A genre string
      "price": float       # A real number price
    },
    {
      "id": integer,
```

```
      "title": "string",
      "author": "string",
      "genre": "string",
      "price": float
    },
    ...
  ]
}
```

When no books are found, the response body will look like the following:

```
{
  "books": []
}
```

Note: This is still considered a successful response.

Failure Response:

For this API, you do not need to produce any failure response.

Examples

Let's look at some example requests and response.

Example request: search by title

- Request URL: [GET] http://localhost:5000/api/books?title=Island
- Request body: None

Example success response:

- Content Type: application/json
- Response HTTP Status Code: 200
- Response body:

```
{
  "books": [
    {
      "id": 11,
      "title": "Island",
      "author": "Aldous Huxley",
      "genre": "Science Fiction",
      "price": 16.99
    }
  ]
}
```

Example request: search by author and sort by descending price

- Request URL: [GET] http://localhost:5000/api/books?author=Jane%20Austen&sort=price
- Request body: None

Please note that some characters are URL encoded in the query param. For example, `Jane Austen` will be sent as `Jane%20Austen`: Your application should handle them automatically. If not, you must use URL decoder on the fields.

Example success response:

- Content Type: application/json
- Response HTTP Status Code: 200
- Response body:


```
{
  "books": [
    {
      "id": 5,
      "title": "Sense and Sensibility",
      "author": "Jane Austen",
      "genre": "Romance",
      "price": 17.99
    },
    {
      "id": 6,
      "title": "Emma",
      "author": "Jane Austen",
      "genre": "Romance",
      "price": 22.99
    },
    {
      "id": 4,
      "title": "Pride and Prejudice",
      "author": "Jane Austen",
      "genre": "Romance",
      "price": 29.99
    }
  ]
}
```


Example request: search by title, but no books found

- Request URL: [GET] http://localhost:5000/api/books?title=Tarjan&sort=title
- Request body: None

Example success response:

- Content Type: application/json

- Response HTTP Status Code: 200
- Response body:

```
{
  "books": []
}
```

Example request: search by genre and sort by descending without specifying a sorting field

- Request URL: [GET] http://localhost:5000/api/books?author=Jane%20Austen&order=DESC
- Request body: None

Example success response:

- Content Type: application/json
- Response HTTP Status Code: 200
- Response body:

```
{
  "books": [
    {
      "id": 6,
      "title": "Emma",
      "author": "Jane Austen",
      "genre": "Romance",
      "price": 22.99
    },
    {
      "id": 5,
      "title": "Sense and Sensibility",
      "author": "Jane Austen",
      "genre": "Romance",
      "price": 17.99
    },
    {
      "id": 4,
      "title": "Pride and Prejudice",
      "author": "Jane Austen",
      "genre": "Romance",
      "price": 29.99
    }
  ]
}
```

Verdict and Scoring

Unlike a traditional programming contest, you will not get an immediate verdict.

For each judge test on your API, you have a total score of 100. The score distribution for each test is as follows:

- 10% score for producing the correct HTTP status code.
- 10% score for producing a valid JSON object.
- 80% score for producing a correct response.

Scores will be published on the contest website or via email after the judging process is finished.

Submission Rules

Please read the submission rules in your Google classroom thoroughly. Ask questions if there is any confusion.