**UNIVERSITY OF TECHNOLOGY, MAURITIUS**

**SCHOOL OF INNOVATIVE TECHNOLOGIES AND ENGINEERING**

**DEPARTMENT OF INDUSTRIAL SYSTEM ENGINEERING**

**Module Name: Web Services**

**Module Code: WAT2124C**

**Cohort: BCNS/ 19A/ FT**

**Student Name: Arouven POOLIAN**

**Student ID: 1903\_16737**

Table of Contents

[Chapter 1: Introduction 1](#_Toc78373508)

[Chapter 2: Design and implementation 2](#_Toc78373509)

[2.1: Core functionalities 2](#_Toc78373510)

[2.2: Additional functionalities 3](#_Toc78373511)

[2.2.1: Headers 3](#_Toc78373512)

[2.2.2: How to use 4](#_Toc78373513)

[2.2.3: Scalable API 5](#_Toc78373514)

[2.2.4: XML and JSON format 7](#_Toc78373515)

[2.3: Client website 9](#_Toc78373516)

[2.3.1: JSON format 10](#_Toc78373517)

[2.3.2: XML format 12](#_Toc78373518)

[2.3.3: JavaScript 13](#_Toc78373519)

[Chapter 3: Testing 15](#_Toc78373520)

[3.1: API Testing 15](#_Toc78373521)

[3.1.1: How to use page 15](#_Toc78373522)

[3.1.2: Book not found 15](#_Toc78373523)

[3.1.3: Bad request to API page 16](#_Toc78373524)

[3.1.4: Request XML format 16](#_Toc78373525)

[3.1.5: Request JSON format 17](#_Toc78373526)

[3.1.6: Request for part of word and several queries at once 17](#_Toc78373527)

[3.2: Website Testing 18](#_Toc78373528)

[Chapter 4: Conclusion 19](#_Toc78373529)

[Annex A: myjs.js 20](#_Toc78373530)

[Annex B: display.php 21](#_Toc78373531)

[Annex C: database.php 22](#_Toc78373532)

[Annex D: credentials.php 23](#_Toc78373533)

[Annex E: myapi.php 24](#_Toc78373534)

[Annex F: index.php 25](#_Toc78373535)

[Annex G: detail.php 26](#_Toc78373536)

[Annex H: bookstore.sql 27](#_Toc78373537)

**List of Figures**

[Figure 2.1: function to add headers. 3](#_Toc78373538)

[Figure 2.2: code for how to use. 4](#_Toc78373539)

[Figure 2.3: push to array. 5](#_Toc78373540)

[Figure 2.4: WHERE and AND added to subquery string. 6](#_Toc78373541)

[Figure 2.5: full substring output. 6](#_Toc78373542)

[Figure 2.6: code to output in JSON format. 7](#_Toc78373543)

[Figure 2.7: code to output in XML format. 8](#_Toc78373544)

[Figure 2.8: user-defined function xml\_encode 8](#_Toc78373545)

[Figure 2.9: Add php file to index page. 9](#_Toc78373546)

[Figure 2.10: default constructor of display class. 9](#_Toc78373547)

[Figure 2.11: JSON display function. 10](#_Toc78373548)

[Figure 2.12: filler function. 11](#_Toc78373549)

[Figure 2.13: XML Display function. 12](#_Toc78373550)

[Figure 2.14: Verify format. 13](#_Toc78373551)

[Figure 2.15: build URL method. 13](#_Toc78373552)

[Figure 2.16: fill elements. 14](#_Toc78373553)

[Figure 3.1: How to use page. 15](#_Toc78373554)

[Figure 3.2: No book found. 15](#_Toc78373555)

[Figure 3.3: Bad request. 16](#_Toc78373556)

[Figure 3.4: Request for XML format. 16](#_Toc78373557)

[Figure 3.5: Request for JSON format 17](#_Toc78373558)

[Figure 3.6: Request for part of word and several queries at once. 17](#_Toc78373559)

List of Acronyms

API - Application Program Interface 1

HTML - HyperText Markup Language 11

ISBN - International Standard Book Number 7

JSON - JavaScript Object Notation 1

OOP - Object-Oriented Programming 7

php - Hypertext Preprocessor 1

SQL - Structured Query Language 6

URL - Uniform Resource Locator 5

XML - eXtensible Markup Language 1

# Introduction

In order to have some functionalities on the same page, a website is built to query a server and receive the JavaScript Object Notation (JSON) or extensible markup language (XML) file and display the required data. There will be a rest web service application program interface (API) using Hypertext Preprocessor (php).

# Design and implementation

## Core functionalities

The core functionalities developed on the API are also implemented in the client’s web site. They are as follows:

* Get books information by title
* Get books information by author
* Get books information by publication year
* Get books information by author and category
* Get books information by author and publication year

Rather than the traditional approach (SELECT \* FROM bookstore.books WHERE title=’harry potter’ ;), another type of approach has been used which will make the API more scalable and to work better with multiple query string. This will be explained further section 2.2.3: Scalable API.

## Additional functionalities

### Headers

In the case of the bookstore, either the books are found or not. But there is another one that can arise; bad request. To help programmability, status code, message, and total book in the database are prepended to the response. A function addStatusToData ($status, $statusMessage, $count, $data) was created (as show in Figure 2.1) to add the above before the data/response.

The status code and messages are as follow:

* 200 – Successful (represent book found in bookstore)
* 400 – Bad Request (represent wrong URL query)
* 404 – Not found (represent book not found in bookstore)

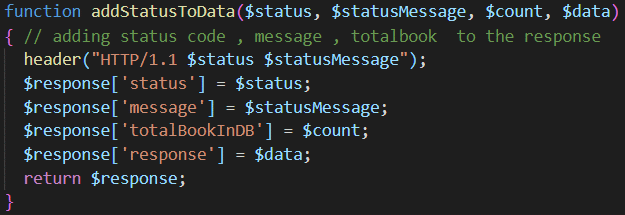


Figure .: function to add headers.

### How to use

If there is no query in the URL, it is assumed that the client don’t know how to use the API. Figure 2.2 shows part of the code to display the how to use page to the user. This will be further explained in Chapter 3: Testing.

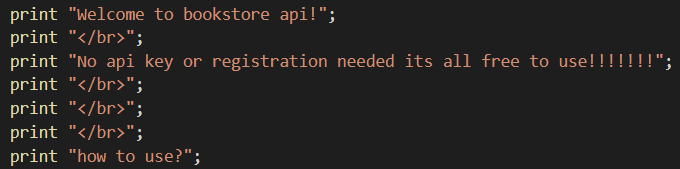


Figure .: code for how to use.

### Scalable API

As mentioned before, this will make the API more dynamic. The API will be programmed to allow any of its users to use it without any restriction in the number of query passed as request. Annex E: myapi.php has the full code for any reference.

The approach is as follows:

1. Get the main query – “SELECT \* FROM books”
2. Combine/concatenate it with the subquery (if any). A function was built to perform this task. It requires all the parameters of the Uniform Resource Locator (URL) to perform it task and return the subquery.

 $this->query = "SELECT \* FROM books" . $this->subquery($params);

**Exploring subquery ($params) method/function.**

The principle is that every query parameters should be prepend with the word “AND”. To achieve this, an array is created. All the query that required the keyword will be push to that array. Figure 2.3 shows two examples of parameters that should be prepended.

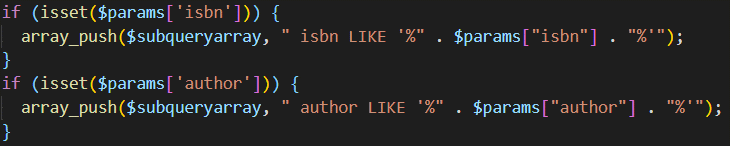


Figure .: push to array.

Then the next step is to add the “WHERE” keyword. This will be done only if there is something in the array. After that, a Foreach loop will do the job of adding the “AND” in front of each string in the array. The result of $subquery in Figure 2.4 with input author=harry&isbn=001 will be “ WHERE AND author LIKE '%harry%' AND isbn LIKE %001%”. The keyword LIKE will allow the user to insert part of the word to query.

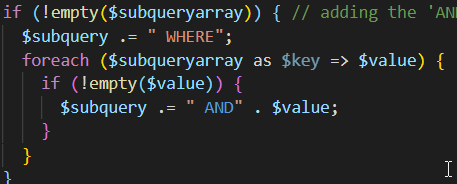


Figure .: WHERE and AND added to subquery string.

As the MYSQL syntax does not require any “AND” in front of “ORDER BY” and “LIMIT”, they were not push to array. For example, author=harry&isbn=001&orderby=rating&limit=5 will output structured query language (SQL) $subquery = “ WHERE AND author LIKE ‘%harry%’ AND isbn LIKE %001% ORDER BY rating LIMIT 5”. The last line of that function will solve the problem of the “WHERE AND” in front of the subquery by replacing it with “WHERE” and add the ending of SQL query as show in Figure 2.5. This will lead to output “ WHERE author LIKE ‘%harry%’ AND isbn LIKE %001% ORDER BY rating LIMIT 5;”

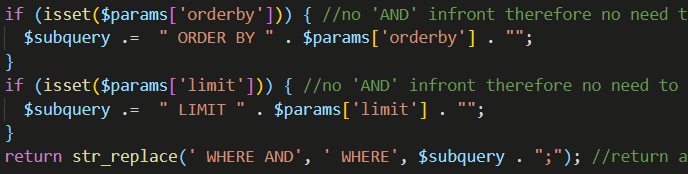


Figure .: full substring output.

Therefore,

 $this->query = "SELECT \* FROM books" . $this->subquery($params);

Will contain “SELECT \* FROM books WHERE author LIKE ‘%harry%’ AND isbn LIKE %001% ORDER BY rating LIMIT 5;”

Using this type of approach, the following can be accomplished:

* Search for word that contain some characters in title, author, International Standard Book Number (ISBN), category, language, and year published.
* Order the result by author, category, etc.
* Limit the result of the search to x number. Where x is a number define by the user.
* Above all, the user is now unlimited with the column to query. He/she can query more than one for example author and category and much more.

The query is then executed and return in a form of array. This array will be converted to either XML or JSON and outputted to the user.

### XML and JSON format

By using Object-oriented programming (OOP) and some functions/methods, both XML and JSON can be outputted from the API.

#### JSON format

Figure 2.6 show the code that will output the data in JSON format. Header is added and php already has its own function to output in JSON (json\_encode). This will take the array outputted from the SQL query and convert it to JSON format.

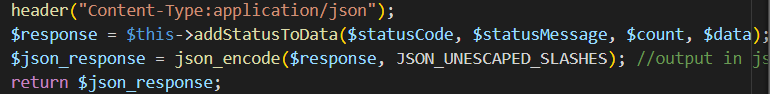


Figure .: code to output in JSON format.

#### XML format

XML on the other hand is more complicated as there is no inbuilt method to encode in XML format. But the principle stays the same. A user-defined function should be created to perform this task. Figure 2.7 shows how the user-defined will look like.

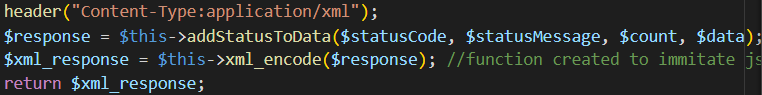


Figure .: code to output in XML format.

In Figure 2.8, the XML header is added together with the first element. Then the array will be retrieved from the SQL query will be converted to XML with bookX tag(s) (X start from 0 to total record in array) and inserted between the bookstore tags.

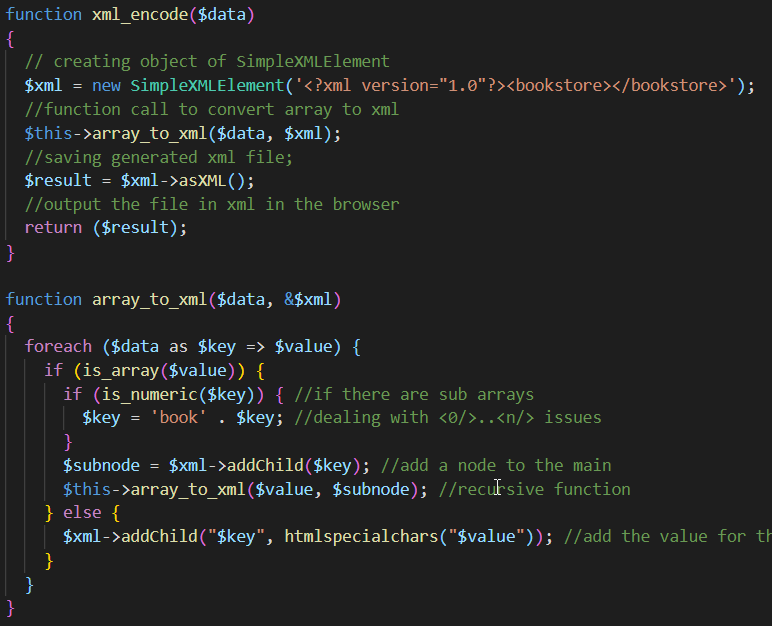


Figure .: user-defined function xml\_encode

## Client website

The client website support both XML and JSON responses. In order to use the template efficiently, all codes were put on a separate file and then called in the pages as shown in Figure 2.9. It also indicates that the same query from the client URL will be pass through the API.



Figure .: Add php file to index page.

Figure 2.10 shows the default constructor of the display class. The constructor will talk the API and detailed page (will be used in books detailed page) as parameters. It will then gather the client’s query in the client’s URL and execute the appropriate function if format detected is JSON or XML.



Figure .: default constructor of display class.

### JSON format

This function shown in Figure 2.11 will get the contents from the API and decode the retrieved JSON formatted data. It will then verify if the response has a status code of 200 and then start to display if it is not 200 an error message will be displayed. Then it will call the filler function for each records from the $data (JSON decode function).

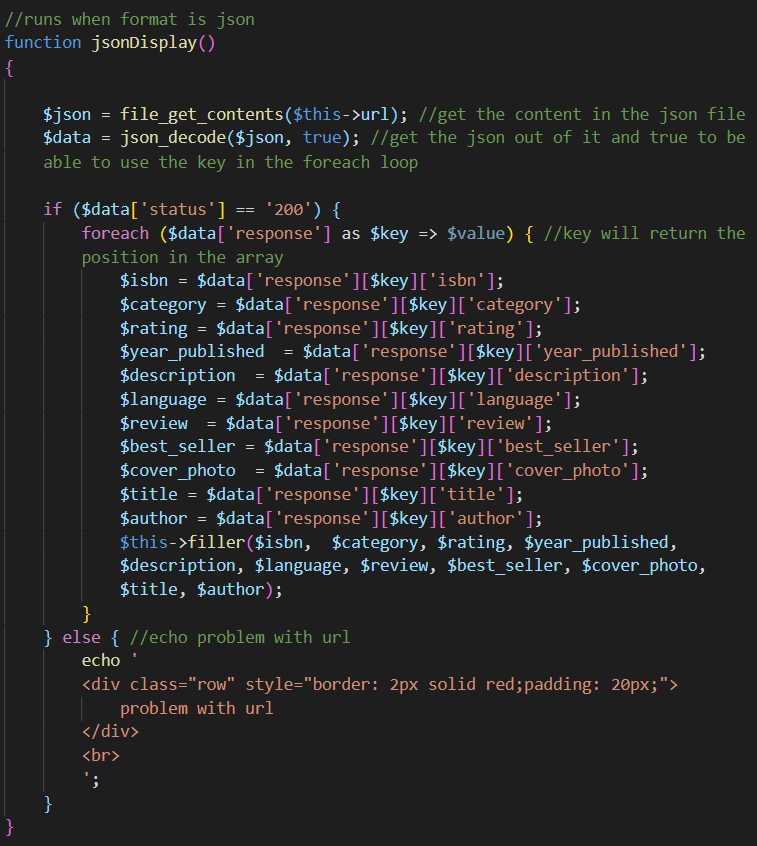


Figure .: JSON display function.

Figure 2.12 shows the filler function. This function will take all previously obtained values and put them in hypertext markup language (HTML) format. Here the detailed page came in play. There is only two pages on the client website, one to display the list of books and the other to display all the details of the book.

|  |
| --- |
|  |
|  |

Figure .: filler function.

### XML format

Figure 2.13 shows the XML Display function. It is the same principle as in section 2.3.1: JSON format. It will load the API, verify if its code is 200, and call the filler function. If it is not 200, error will be displayed.

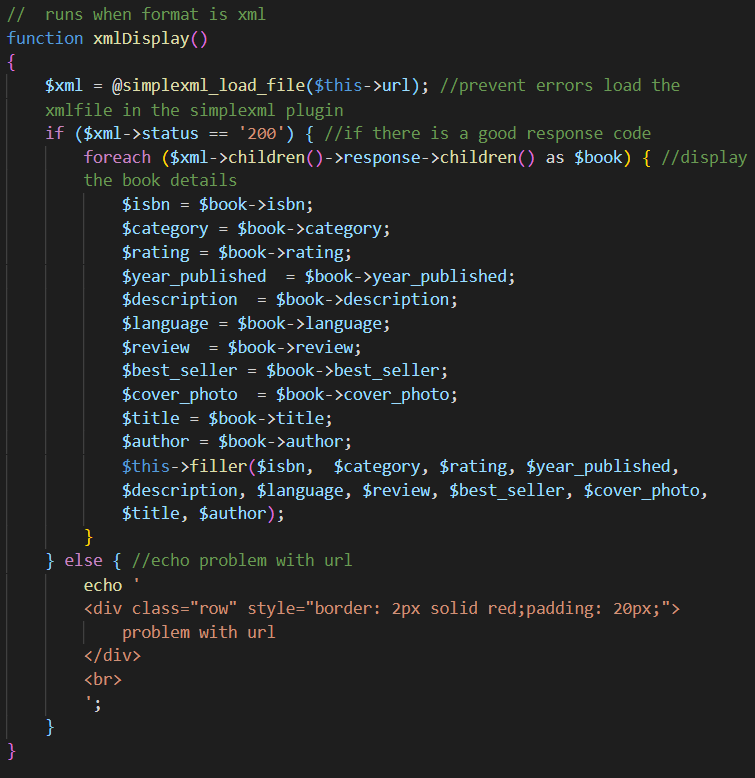


Figure .: XML Display function.

### JavaScript

The piece of codes shown in Figure 2.14 is responsible to verify if the client website has format in its query. This is vital as without this no book will be displayed. If there is no format in the query, java script will add a default format to it (the chosen default format is JSON). Then the client site will be loaded with the new URL.

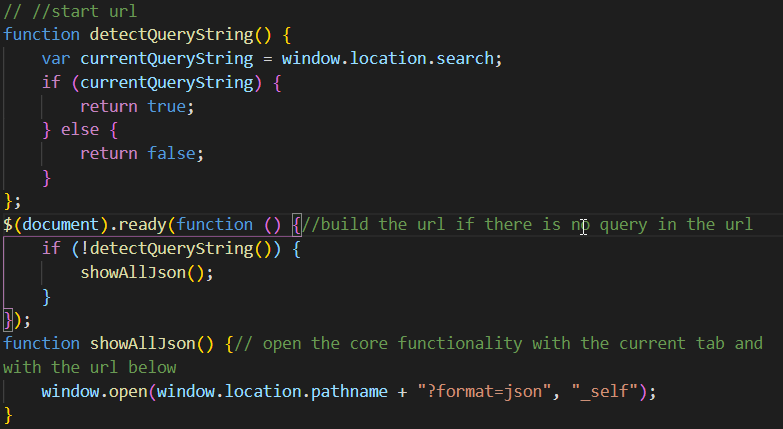


Figure .: Verify format.

Figure 2.15 shows the search system of the client website. On clicking on the search button, the query will be build and then opens. While opening, it will be the new link that will be in the URL (built URL). This will facilitates the php API query and also reduce the server side processing time.



Figure .: build URL method.

Other function of java script is the fill elements (see Figure 2.16). This snippet will take the parameters in the website URL and retrieve the queries. Each query will be associated with their respective textboxes or dropdown menu.



Figure .: fill elements.

# Testing

## API Testing

### How to use page

The how to use page appears when the link does not have any query as shown in Figure 3.1.

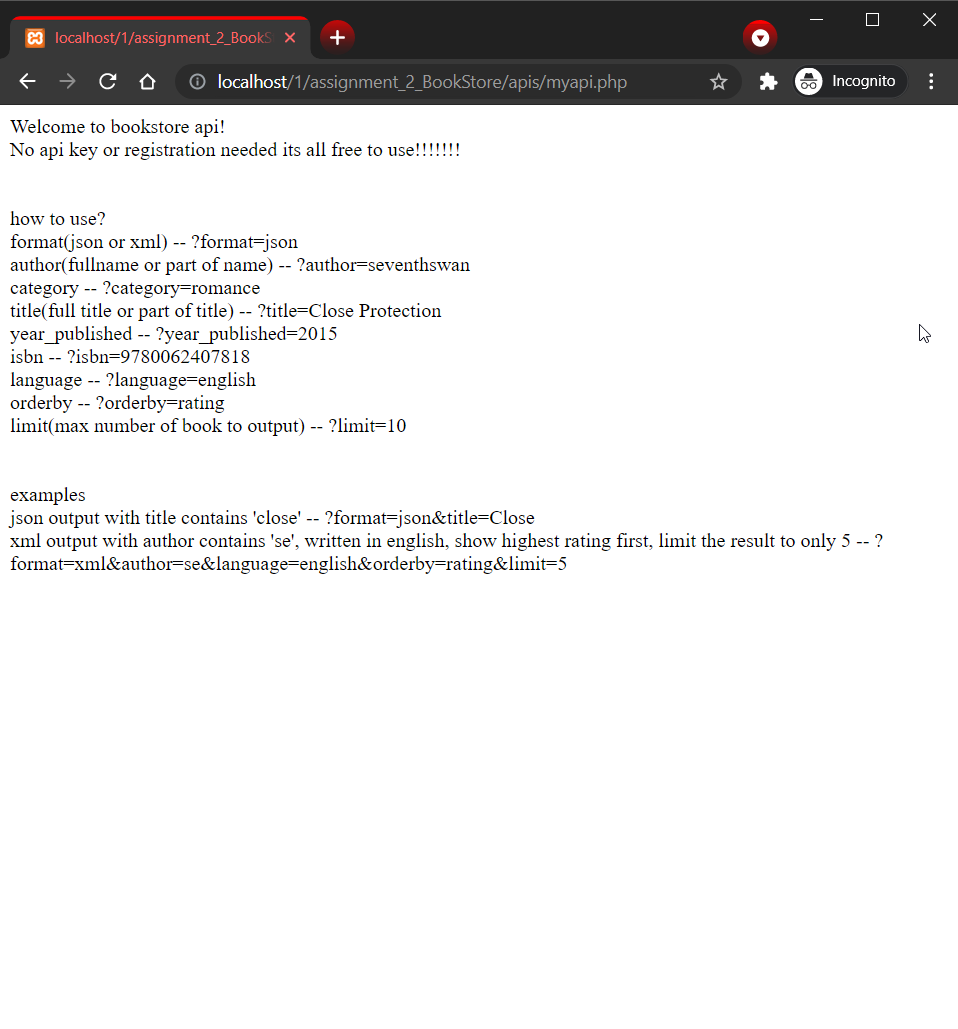


Figure .: How to use page.

### Book not found

Figure 3.2 shows the error code 404 for no book found. This is because there is no author that has “thorn” in their name in the database.

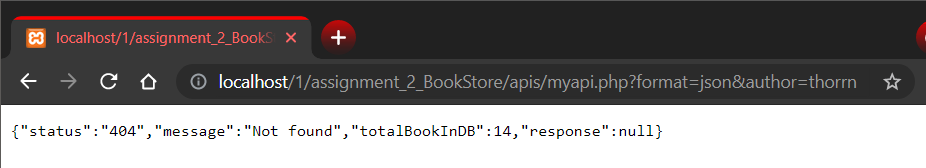


Figure .: No book found.

### Bad request to API page

Figure 3.3 shows what happened if there is a bad request. In this case a wrong format as query.

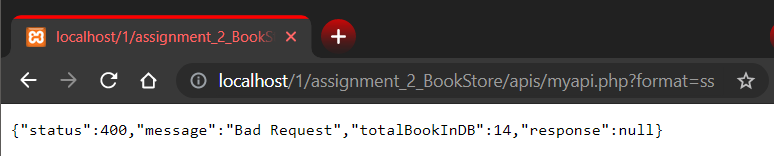


Figure .: Bad request.

### Request XML format

Figure 3.4 shows the output when requesting for the XML format.

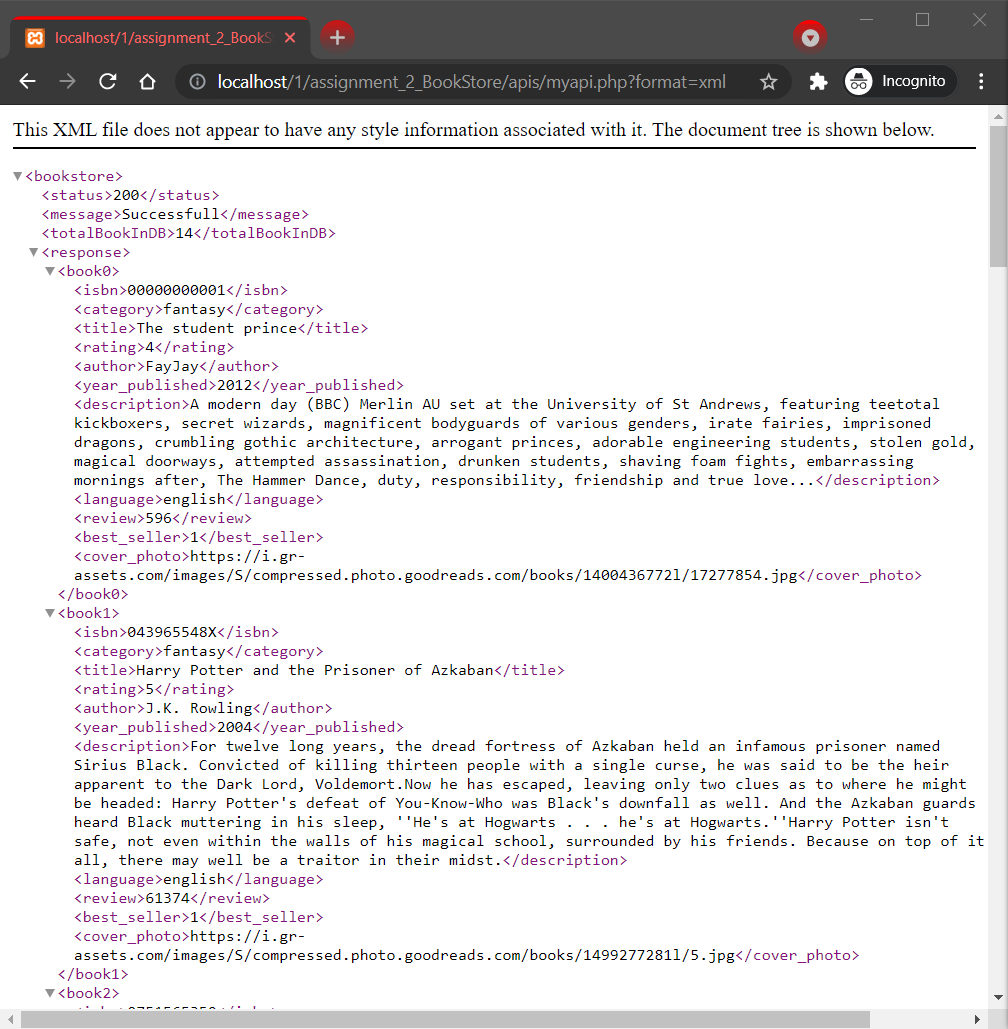


Figure .: Request for XML format.

### Request JSON format

Figure 3.5 shows the output when requesting for the JSON format.

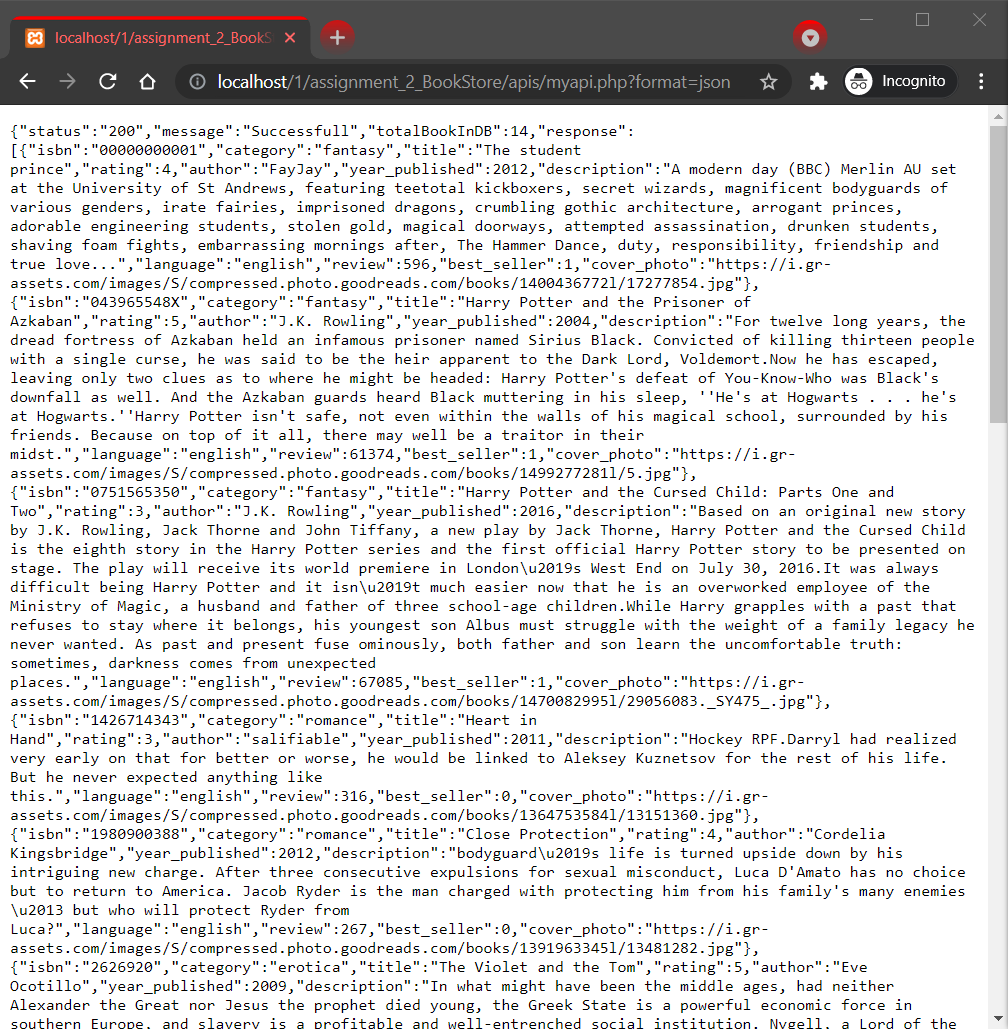


Figure .: Request for JSON format

### Request for part of word and several queries at once

The status code 200 with message Successful and the total book in the database are displayed in every request. Figure 3.6 illustrates the query for part of the title “harry” and multiple queries such as language, limit, and order by. In this database, the only books title that has “harry” in them are the “Harry Potter” s books.

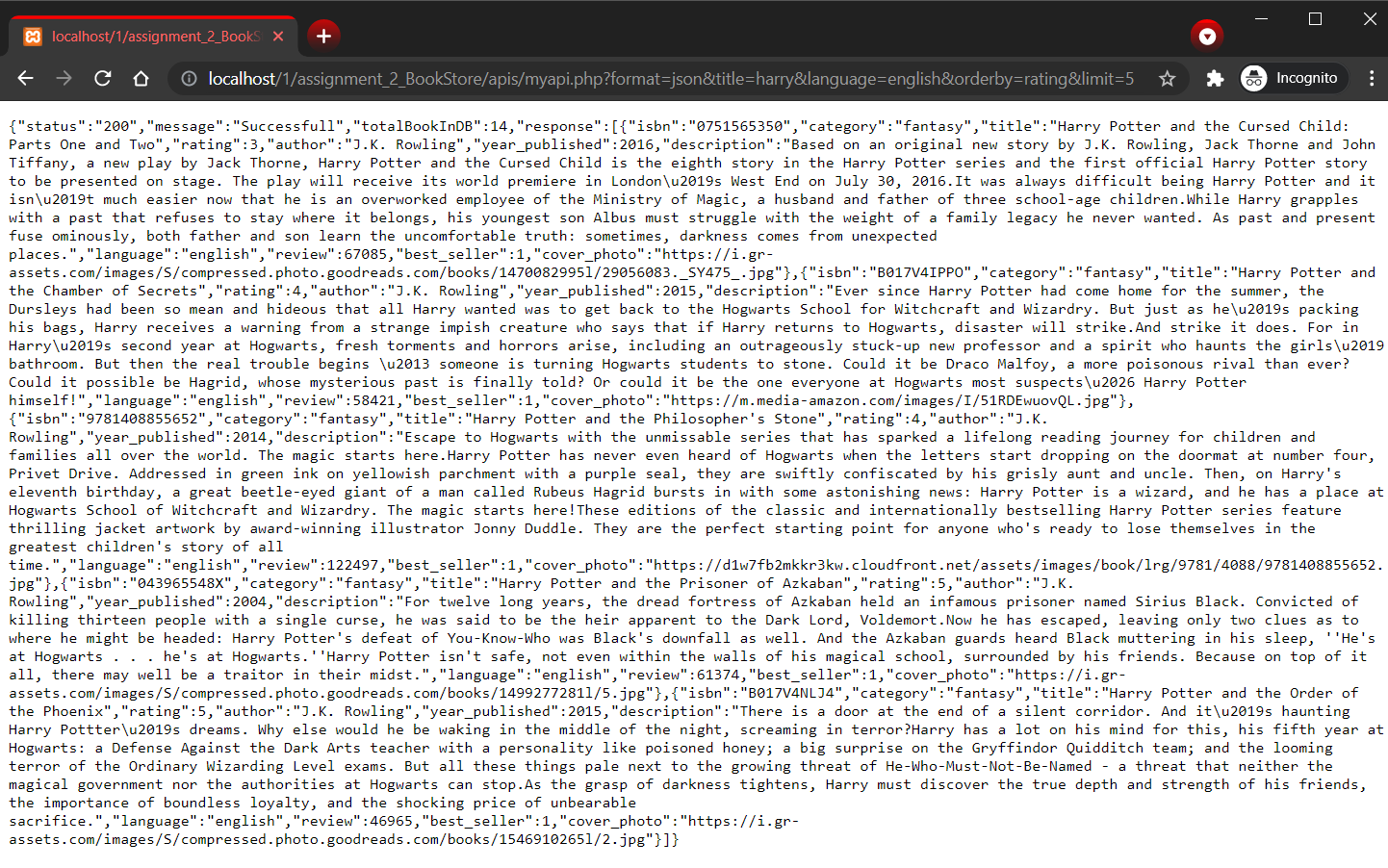


Figure .: Request for part of word and several queries at once.

## Website Testing

# Conclusion

As seen in the Chapter 3: Testing, the whole project works well. As an improvement, a user basket could be added. This will help the book buyers to know how much they will buy or improve checkout. Another improvement will be a notification that will tell the buyer when the stock is going down or in case of promotion.

The link to GitHub file:

<https://github.com/Arouven/webservices>

# Annex A: myjs.js

# Annex B: display.php

# Annex C: database.php

# Annex D: credentials.php

# Annex E: myapi.php

# Annex F: index.php

# Annex G: detail.php

# Annex H: bookstore.sql