**Assignment 2**

**Total:** 25 Marks

**Due:** 19th Oct 2022

**Problem:**

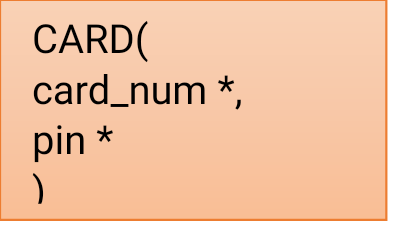
Note: If you have the expertise, you can build a proper HTML/CSS/JS interface, but it is not required.

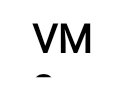
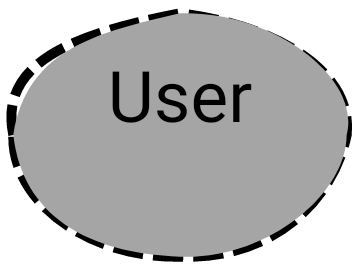
***This is an open-ended assignment, as long as the following functionalities are implemented, it is your choice how you implement them.***

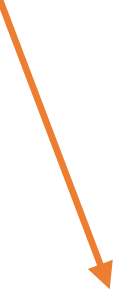
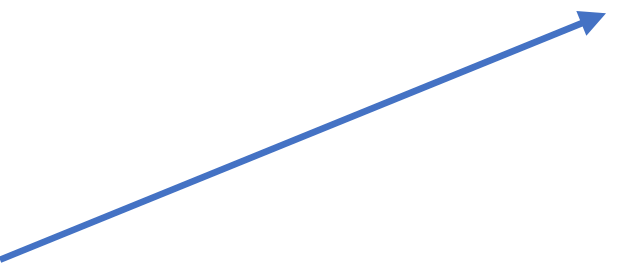
You have to establish a set of web services for an online shopping website.

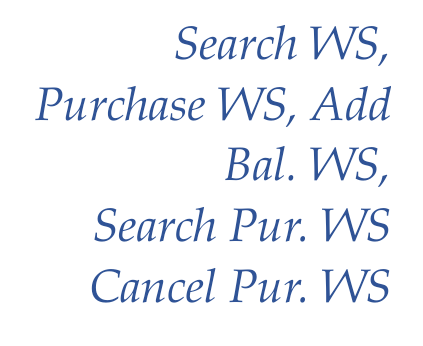
You do not have to create a proper HTML/CSS/JS-based website for this. But the web pages you will develop can look like the test and demonstration web pages, where we can test individual web services by passing the required parameters and seeing the outcomes.

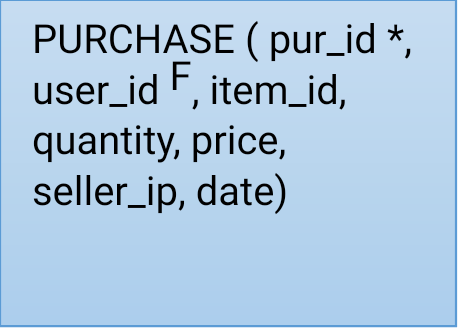
Consider the WS Architecture with FOUR sections (Bank, Market, Seller 1 and Seller 2):



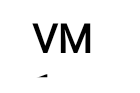
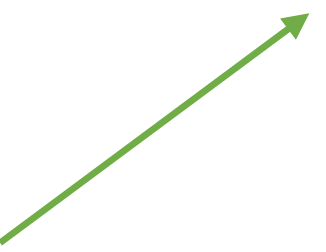


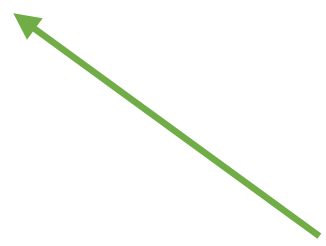


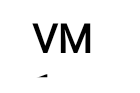


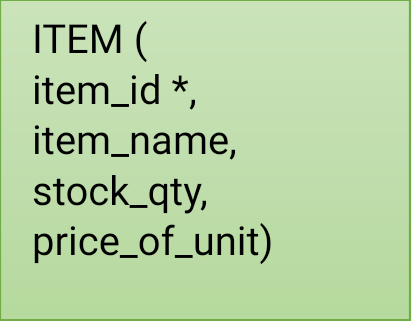
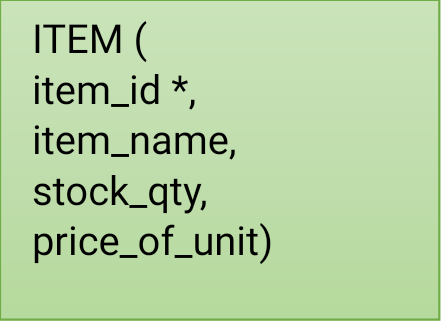


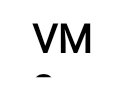












You have created the following web services:

1. ***Item list web service***: This web service will return an XML response of what items are available from a seller. Basically, each **IP address** you have will represent a **seller.** So, the seller’s item list can be determined by:

https://<ipaddress>/seller1/itemlist.php

You decide what you want to sell and prepare the list (at least 3 items). The XML file should look like this: (you can use JSON equivalent as well)

<root>

<item>

<itemid>100</item>

<itemname>Very legal item</itemname>

<price>5</price>

</item>

<item>

<itemid>120</item>

<itemname>Something</itemname>

<price>50</price>

</item>

...

</root>

Create appropriate tables in your MySQL database and fill in your data. Then create the XML/JSON dynamically. Use appropriate *content type*.

Given an IP address, the item list is shown in the XML (or JSON) output in the web browser.

1. ***Search for online items***: In this web service, you can pass a search parameter with any *name*, and if an item is found in any of the IP addresses, the details (*item\_id*, *name*, *price, seller\_ip*) are returned as an XML or JSON output in your web service. It should be sorted in *ascending* order with regard to *price*. A seller is identified by the *seller\_ip*.

Fix a set of IP addresses to search as you wish. You should not access the database directly with this WS; you must consume the book list WS. You can use your own VMs to set up at least 2 item lists.

*You have to merge the lists if the item is found in two separate seller lists*. *But the purchase in the next step happens in only one VM. You may manually copy-paste details from the search into the purchase service.*

If you feel more adventurous, ask other students for their IP addresses with the XML item list (you will also need to see the format they are using), but this is not required.

Hint: You have to check more than one Fixed IP address within the same web service.

1. ***Purchase an online item***: This web service will take the parameters *user\_id,* *item\_id, seller\_ip, and quantity* to the web service, reducing the *balance* the *user* has based on the *price* x *quantity*. The *price* and available *quantity* must be retrieved from the *seller\_ip’s* book list from (*i*). The transaction will fail if there is an inadequate balance or available quantity. The output (purchase fail or successful purchase details) is returned in XML or JSON format including the *pur\_id*, if transaction is successful. This adds a record to the purchase table, if transaction is successful.

You must create appropriate USER and PURCHASE tables for this.

1. ***Search Purchase***: This web service allows you to search for a previous purchase with either a *pur\_id* or *user\_id*. This service should return the result of the search format and should be in XML or JSON.
2. ***Cancel Purchase*:** Once a purchase is found, it can be cancelled as well using the *pur\_id*. This service should return the result of the search format and should be in XML or JSON. This deletes the record in the purchase table.
3. ***Add Balance***: This web service adds funds to the users’ *balance*. It takes an *amount,* and *user\_id* adds the amount to the balance. If zero is added, the current value is returned. Otherwise, the new value is returned. The return format should be XML or JSON.
4. ***CardCheck*:** This web service returns a *true* or *false* (or *0*/*1*), based on the correct card and pin combination. It will return an XML or JSON response as well. Only if this returns *true*, the purchase can go ahead, or a balance can be added. So, this WS impact *iii* and *vi*.

You can choose your style of XML or JSON response. But it should have the minimum information according to what is mentioned above. You will have to choose appropriate response codes 403, 404, 400, etc. Explain your decision in the comments and in the 2-page report.

**Database:**

Create your own database(s). Ideally, you should have different databases for each section, i.e., 4 in total. But if you cannot do this, you can put all tables in the same database. BUT … a Web Service should ONLY access data from its corresponding tables (and/or database) in its section.

**Typical outcomes:**

You can use the dynamic UI generation or design each form individually on your own separately. For each service, you will have a GUI like this (put all in one viewing a web page or different pages)

Graphical user interface, application

Description automatically generated 

**Submission:**

All PHP codes, XML/JSON, text files, and SQL files as a zip and a 2-page report on MyLO.

Submit a video of the operation, demonstrating search with an item in two lists, a purchase, search and cancel the purchase, balance adding and everything in the marking scheme below. Also, explain your code. The video length should be around 15 mins.

**Marking scheme:**

| **Item** | **Marks** | |
| --- | --- | --- |
| **Fully Completed** | **Partially Completed** |
| A 2-page report on what you have chosen as the XML or JSON and response code structures for each section (7 x 0.5) | **3.5** | 0 for each incomplete part |
| Video demonstration | **2** | 1 |
| The entire architecture is implemented as depicted | **2** | 0 |
| Everything is running on HTTPS | **0.5** | 0 |
| The UI is generated/created correctly with all required parameters and output sections. (7 x 0.5) | **3.5** | 0 for each incomplete part |
| ***Item List (i)*** | | |
| Item list properly set up in at least 2 VMs (2 x 1) | **2** | 0 |
| ***Balance (vi)*** | | |
| Balance is added and shown properly | **1** | 0.5 |
| ***Search for online items (ii)*** | | |
| Merging the lists works correctly if items are found in multiple lists. | **2** | 0.5 |
| Search features display correctly with at least 3 cases: (3 x 1.5)   * multiple items found in a separate list * 1 item found in 1 VM XML list * No item is found | **4.5** | 0.5 for each incomplete case |
| ***Purchase (iii-v)*** | | |
| Balance is subtracted properly | **1** | 0.25 |
| Purchase can be searched | **1** | 0.25 |
| Purchase can be cancelled | **1** | 0.25 |
| Transaction failure is working properly when there is not enough balance | **1** | 0.25 |
| **Total** | **25** |  |