**Table of contents**

[Learning Objectives 1](#_heading=h.gjdgxs)

[Introduction 1](#_heading=h.30j0zll)

[Composite Microservice “Place Order” 2](#_heading=h.1fob9te)

[Part 1: Preparation 2](#_heading=h.3znysh7)

[Part 2: HTTP GET/POST in Python - invokes.py 2](#_heading=h.2et92p0)

[Part 3: What is CORS? 3](#_heading=h.kt1v2qf4in3l)

[Part 4: Preparing the atomic microservices 4](#_heading=h.tyjcwt)

[4.1: Create MySQL tables for the Order microservice 5](#_heading=h.1t3h5sf)

[4.2: Run and try the microservices 6](#_heading=h.4d34og8)

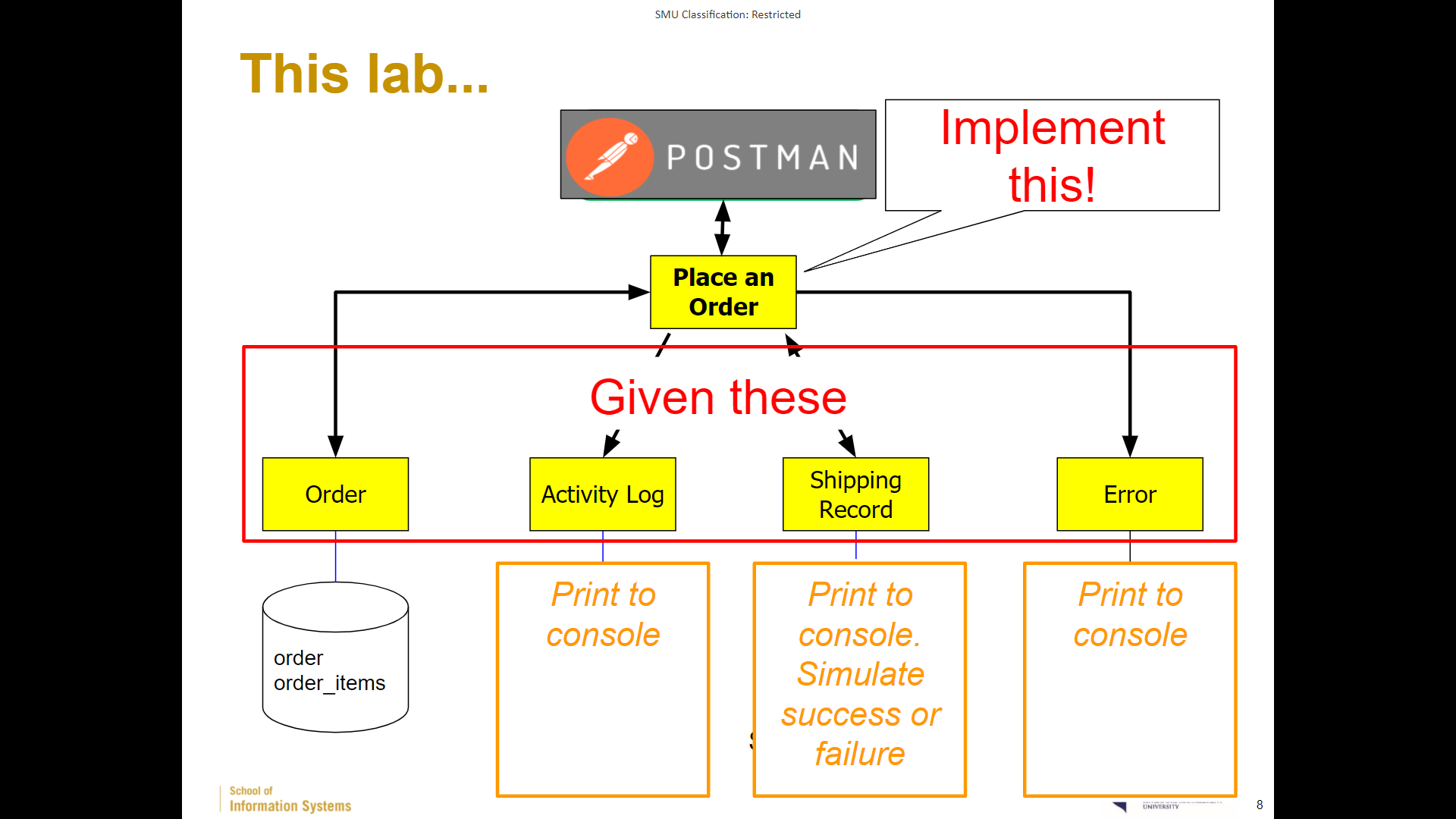
[Part 5: Test place\_order.py 8](#_heading=h.2s8eyo1)

[Learning Points 11](#_heading=h.3rdcrjn)

# Learning Objectives

* How to invoke REST APIs using Python
* Implement a composite microservice to orchestrate atomic microservices to fulfill the “place order” business process.

# Introduction



The scenario is to implement the composite microservice “Place an order” following the above design.

1. Create a new order based on the request from a client to the **Order** microservice.
2. Log the order creation activity via the **Activity Log** microservice, no matter if it's successful or not.
3. If order creation fails, inform the **Error** microservice to handle the failure; Otherwise, notify the **Shipping Record** microservice about the new order.
4. If shipping record creation fails, inform the Error microservice.
5. Return details of the new order and shipping record or their error status to the client.

# Composite Microservice “Place Order”

## Part 1: Preparation

1. Create a working folder for this lab. E.g. C:\ESD\Labs\PlaceOrder\
2. Unzip all the given **resource.zip** into the working folder.   
   You should see the following files
   1. activity\_log.py
   2. book.py
   3. error.py
   4. ESD.postman.json
   5. invokes.py
   6. order.py
   7. order.sql
   8. place\_order.py
   9. shipping\_record.py

## Part 2: HTTP GET/POST in Python - invokes.py

1. Review invokes.py. It has a function **invoke\_http()** that allows you to make HTTP calls to Web APIs.

| invokes.py |
| --- |
| import requests  SUPPORTED\_HTTP\_METHODS = set([  "GET", "OPTIONS", "HEAD", "POST", "PUT", "PATCH", "DELETE"  ])  def **invoke\_http**(url, method='GET', json=None, \*\*kwargs):  """A simple wrapper for requests methods.  **url**: the url of the http service;  **method**: the http method;  **data**: the JSON input when needed by the http method;  **return**: the JSON reply content from the http service  if the call succeeds;  otherwise, return a JSON object with a "code" name-value pair.  """  … |

1. Line 01: **import requests**

The script requires the python module ‘requests’[[1]](#footnote-0). Let’s install it.

**python -m pip install requests**

## Part 3: What is CORS?

1. Open book.py in VSCode. You will notice the following new lines have been added.

| from flask import Flask, request, jsonify  **from flask\_cors import CORS**  from flask\_sqlalchemy import SQLAlchemy  from os import environ  app = Flask(\_\_name\_\_)  **CORS(app)**  … |
| --- |

1. Quote [Flask-Cors 3.0.10 documentation](https://flask-cors.readthedocs.io/en/latest/#:~:text=Usage,those%20who%20prefer%20this%20approach).  
    *This package exposes a Flask extension which by default enables CORS support on all routes, for all origins and methods. …*
   1. We need CORS to allow Cross-Origin Resource Sharing. Cross-origin resource sharing (CORS) is a mechanism for integrating applications.
   2. For example, <http://localhost/index.html> displays all books by making an AJAX call to the book microservice <http://localhost:5001/book>. The web page and microservice are said to be from different origins because of the different ports. In real life, the UI and microservices can even be on different hosts (e.g. <http://zoko.com/index.html>, <http://esd.com:5001/book>).
   3. When Javascript accesses resources (microservice) on a different host and/or port, this is known as cross-origins access. As part of Internet security, this is NOT allowed normally.
2. Hence, by adding the above highlighted lines of code, we are **explicitly allowing cross-origins access to our microservices** to prepare for the web pages of the bookstore that we’ll develop in a future lab.
3. Let’sinstall the module **“flask\_cors”**.  
     
    python -m pip install flask\_cors
4. If you open the given index\_vue.html (labeled “[Front-end for Lab 5](https://drive.google.com/file/d/1Pd5iM5sbvyf5YZ3bJAwc_H19dh0tkrUF/view)” under Content - Week 5) in your browser, you should be able to view the list of books correctly.
   1. Note the HTML file assumes your book microservice’s URL is http://localhost:5000/book

Side note, you can comment out those highlighted lines of CORS code and refresh the HTML to see what happens in the browser’s developer console. Remember to uncomment them after you try!

## Part 4: Preparing the atomic microservices

The python script files for Activity Log, Error, Order and Shipping Record microservices are given. Before the microservices can be executed, there are some setups to be done.

Note:

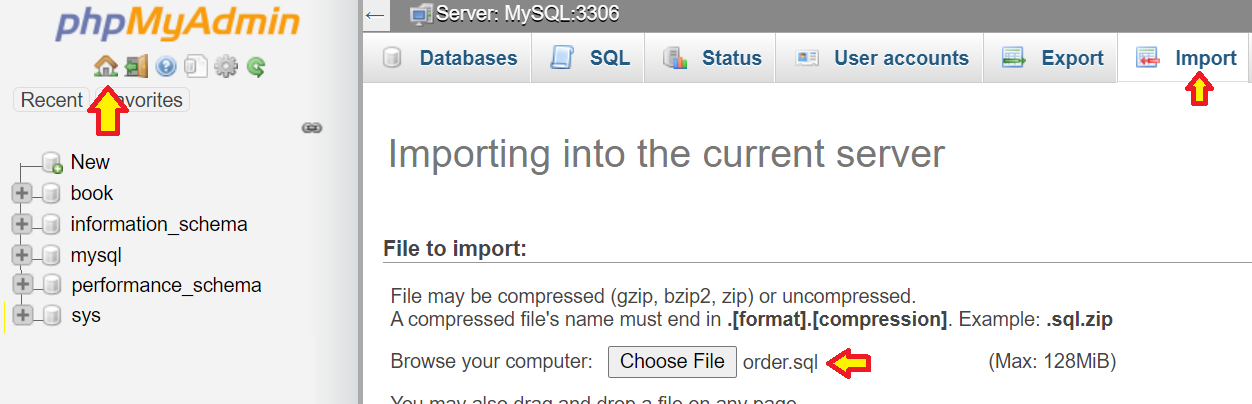
* You may refer to the [API documentation](https://drive.google.com/drive/folders/1sxgjviMF2QsjIHh31rlVIQNlGGksOGA4)[[2]](#footnote-1) to understand the inputs required and the expected output of each microservice.
* The python script **shipping\_record.py** simulates a shipping process success or failure based on the given customer ID. If **customer\_id** contains “ERROR”, it will simulate a shipping process failure. Otherwise, it will simulate success.

### 

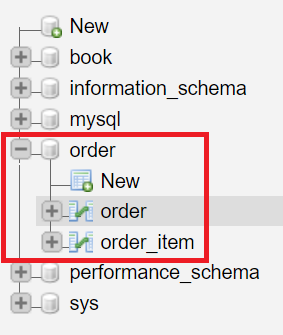
### 4.1: Create MySQL tables for the Order microservice

**order.py** uses two tables “order” and “order\_item” in MySQL database.

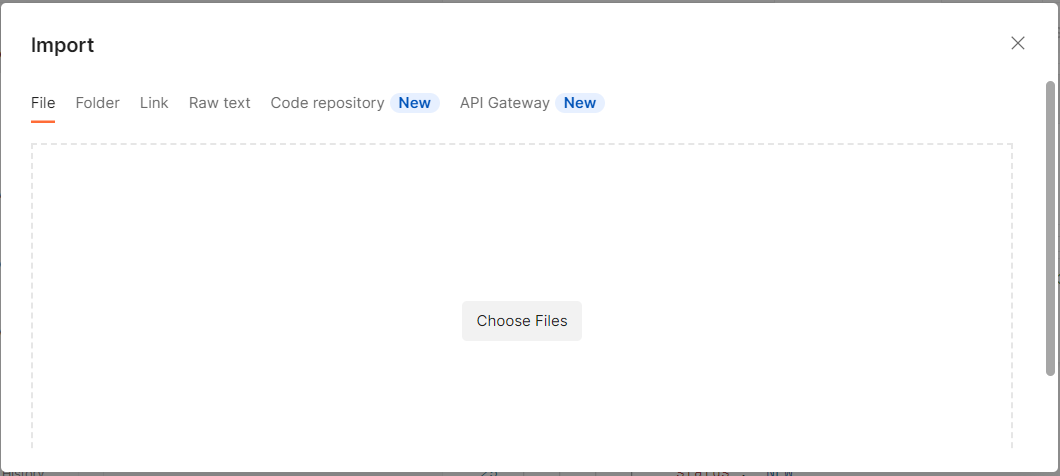
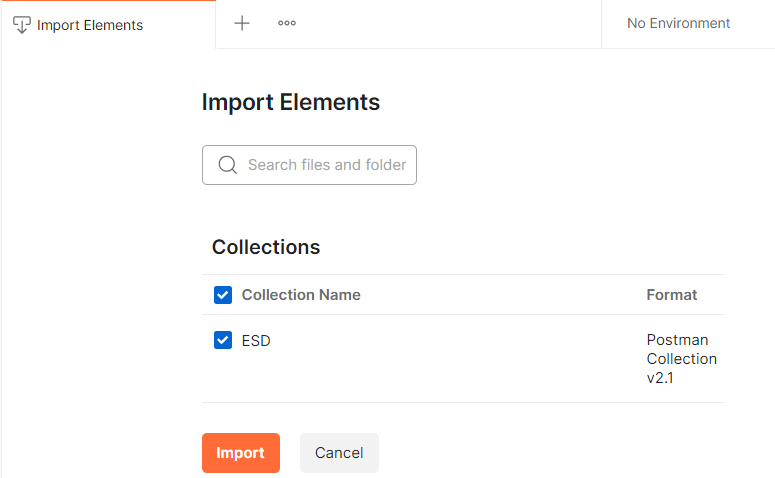
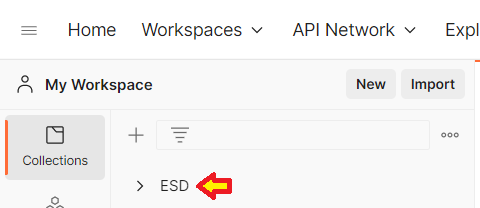
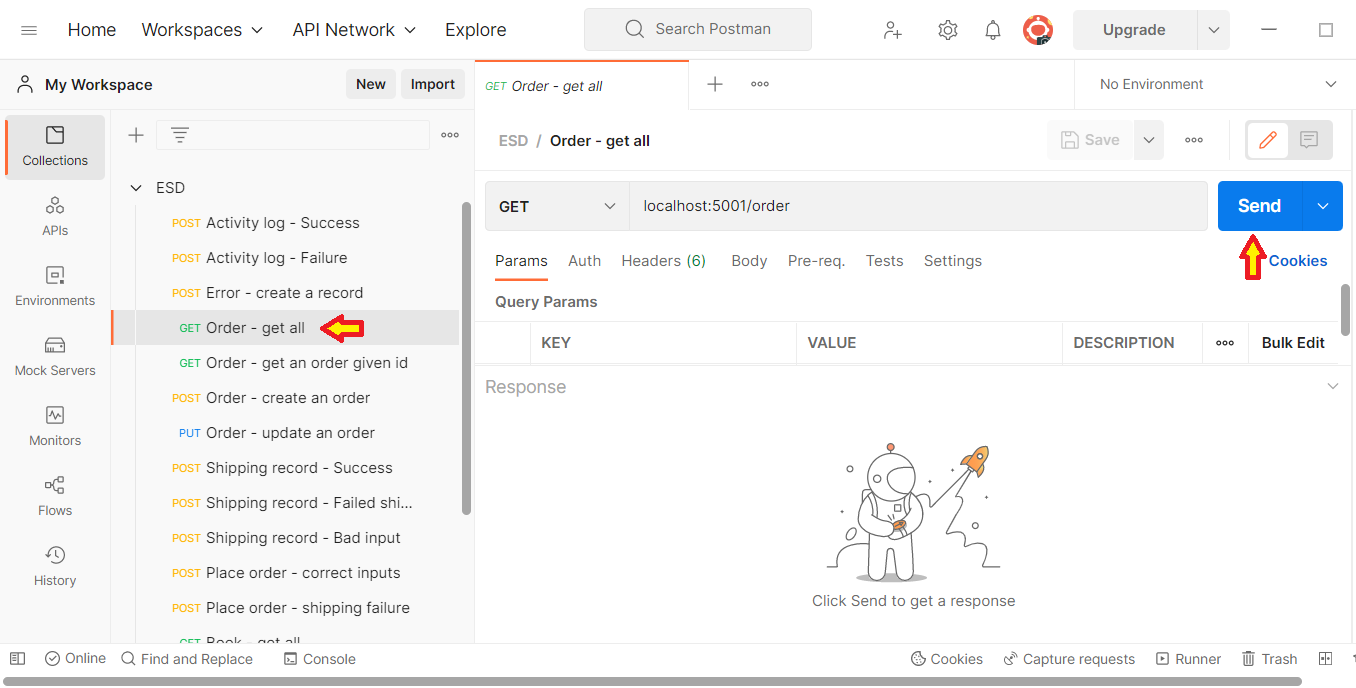
1. Let’s create the two tables
   1. phpMyAdmin > Click “home” icon > tab “Import”
   2. Choose File > the given “order.sql”
   3. Click “Go”



If done successfully,

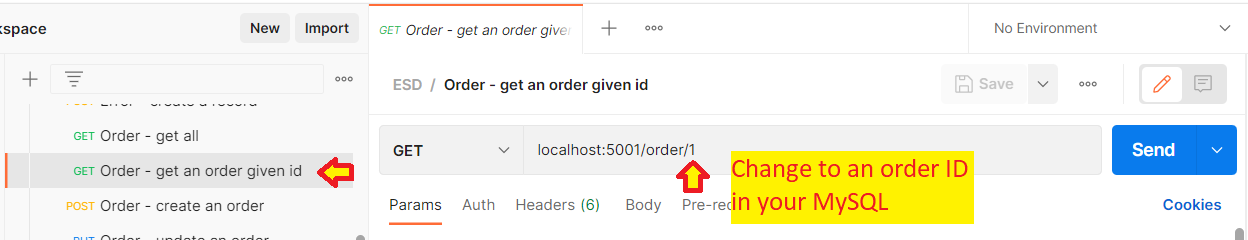


### 4.2: Run and try the microservices

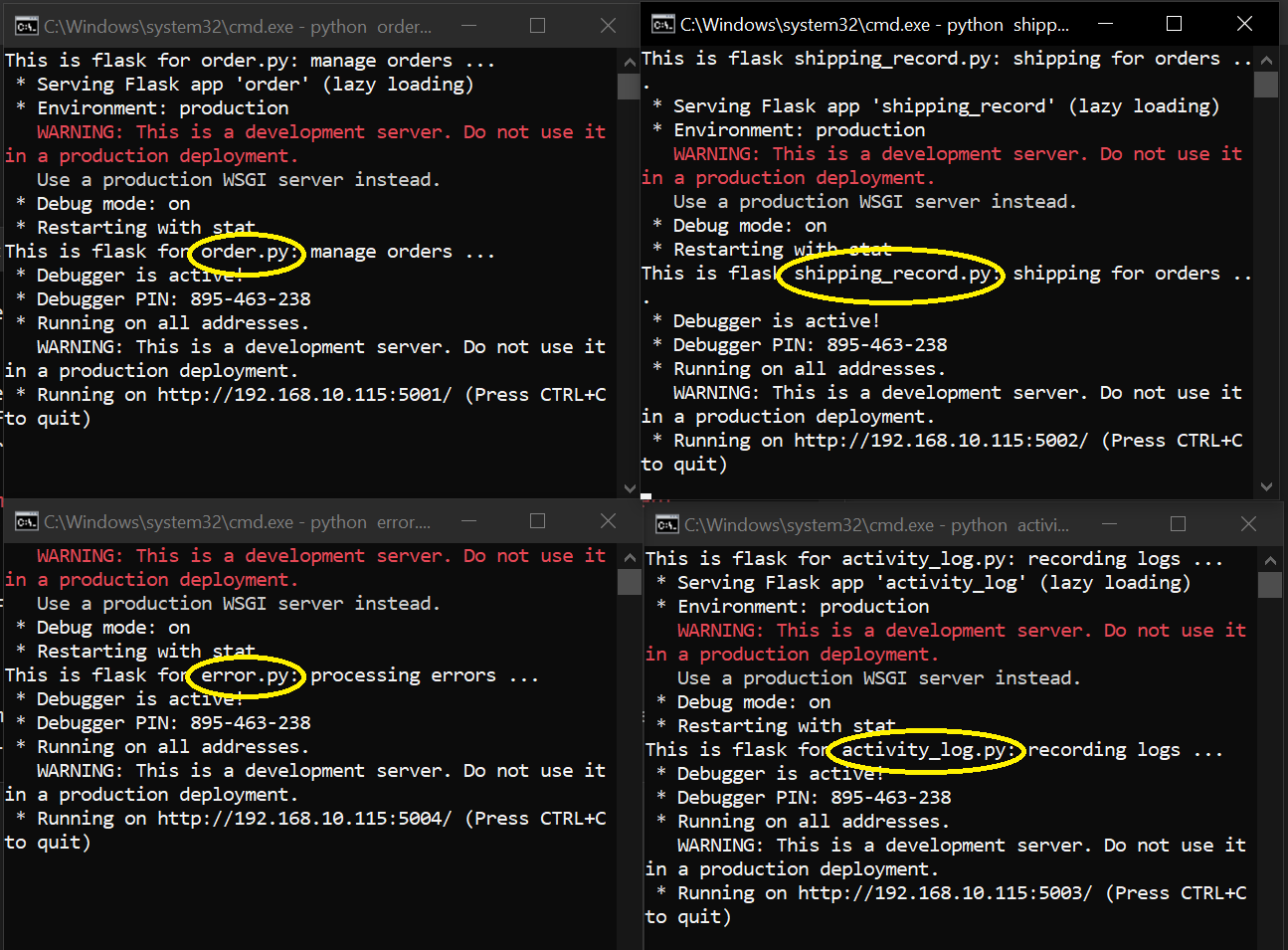
1. Import the given “ESD.postman.json” into Postman
   1. Postman >> menu “File” > Import, and you will see this:  
      
   2. Click “Choose Files” > select the given “ESD.postman.json”
   3. Click “Import”  
      
   4. If done correctly, you should see a “ESD” list under the tab “Collections”.  
      **
2. Run order.py
3. In Postman, send the “Order - get all” request  
   
4. If you encounter errors, check that WAMP server is running and the database connection string is correct; user id, password, port

| app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+mysqlconnector://**root**@localhost:**3306**/order'  **# MAMP has a default password for the *root* account. # Please check your MAMP’s MySQL port number as it may have been changed.** # app.config['SQLALCHEMY\_DATABASE\_URI'] =  # 'mysql+mysqlconnector://**root:root**@localhost:**3306**/order' |
| --- |

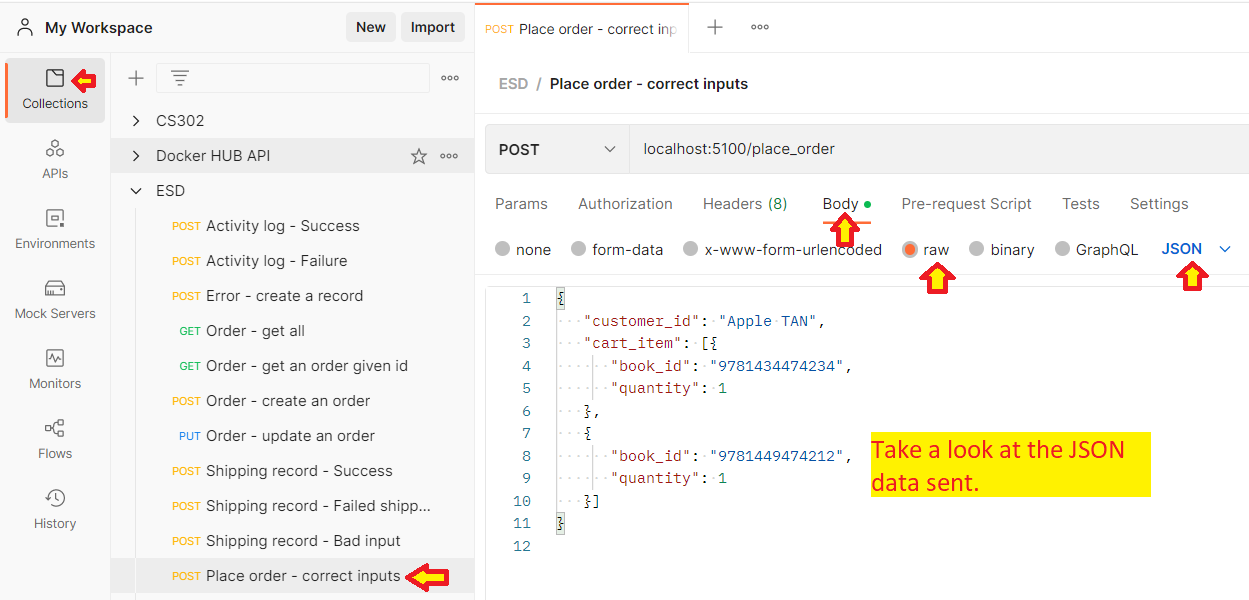
Note:

1. During the lab, if you encounter issues with a particular microservice, you can test it with the given Postman requests.
2. **After class**, for each of the Postman requests do the following:
   1. What is the HTTP method? What is the JSON data sent?
   2. Send the request to test each microservice.
      1. Remember to run the microservice first!
      2. You may need to modify the Postman request according to your data.   
         E.g. “Order - get an order given id”, to test successful response, change the URL to point to a valid order ID in your database.  
         
      3. Check that the inputs/outputs of each microservice match the API documentation.

## Part 5: Test place\_order.py

1. Run **activity\_log.py**, **error.py**, **order.py**, **shipping\_record.py** simultaneously in 4 **separate** CMD windows (or bash terminal on macOS).  
   
2. Open a **5th** CMD window and run place\_order.py.

**python place\_order.py**

1. Go to Postman > tab “Collections”> ESD > **Place order - correct inputs**  
   
2. Click “Send” to test a successful placing of order.

| Expected response in Postman   * order\_id and date/time values may be different |
| --- |
| {  **"code": 201,**  "data": {  "order\_result": {  "code": 201,  "data": {  "created": "Thu, 21 Jan 2021 14:56:59 GMT",  "customer\_id": "Apple TAN",  "modified": "Thu, 21 Jan 2021 14:56:59 GMT",  "order\_id": 12,  "order\_item": [  {  "book\_id": "9781434474234",  "item\_id": 23,  "order\_id": 12,  "quantity": 1  },  {  "book\_id": "9781449474212",  "item\_id": 24,  "order\_id": 12,  "quantity": 1  }  ],  "status": "NEW"  }  },  "shipping\_result": {  "code": 201,  "data": {  "order\_id": 12  },  "message": "Simulated success in shipping record creation."  }  }  } |

| Expected output in CMD window running place\_order.py   * order\_id and date/time values may be different. |
| --- |
| Received an order in JSON: {'customer\_id': 'Apple TAN', 'cart\_item': [{'book\_id': '9781434474234', 'quantity': 1}, {'book\_id': '9781449474212', 'quantity': 1}]}  -----Invoking order microservice-----  order\_result: {'code': 201, 'data': {'created': 'Wed, 08 Feb 2023 18:57:10 GMT', 'customer\_id': 'Apple TAN', 'modified': 'Wed, 08 Feb 2023 18:57:10 GMT', 'order\_id': 15, 'order\_item': [{'book\_id': '9781434474234', 'item\_id': 29, 'order\_id': 15, 'quantity': 1}, {'book\_id': '9781449474212', 'item\_id': 30, 'order\_id': 15, 'quantity': 1}], 'status': 'NEW'}}  -----Invoking activity\_log microservice-----  Order sent to activity log.  -----Invoking shipping\_record microservice-----  shipping\_result: {'code': 201, 'data': {'order\_id': 15}, 'message': 'Simulated success in shipping record creation.'}  127.0.0.1 - - [08/Feb/2023 18:57:13] "POST /place\_order HTTP/1.1" 201 - |

1. Use the given Postman requests to test the following scenarios:

| **Postman requests** | **Scenario** |
| --- | --- |
| Place order - correct inputs | WAMP stopped. order.py running. |
| WAMP running. order.py stopped. |
| Place order - shipping failure | Simulate shipping record failure by sending a request whose customer\_id contains “ERROR”. |

1. Check that the response matches the API documentation.

# Learning Points

* Invoke REST APIs in python.
* Implement composite microservice
  + Orchestrated architectural pattern.
  + A controller that manages atomic microservices to fulfill a business process.
* API documentation specifies the expected inputs and outputs of a service, making it clear for others to use your services.

1. For more details on the “requests” module and the code in function invoke\_http(), please refer to <https://www.w3schools.com/python/module_requests.asp>, <https://requests.readthedocs.io/en/latest/api/> [↑](#footnote-ref-0)
2. API documentation of the microservices can be found in <https://drive.google.com/drive/folders/1sxgjviMF2QsjIHh31rlVIQNlGGksOGA4> [↑](#footnote-ref-1)