Criterion C

Index of techniques

technique	Explanation
JSON (Javascript object notation)	The data is formatted into a JSON format through the ‘jsonify’ method in the Flask library so that it easily readable by the client but also easy to process.
string.split() to alter objects for use and storage	A javascript function similar to the function in Java was used to split up a string which was used to display the items from the web service so that it did not have to reload the data
2-dimensional arrays of objects	Several arrays were used throughout the project. Many were used to store JSON objects that contained their own storage of data.
Parsing data entry	Declare data types for users to input and prevent incorrect data such as the use of HTML input type ‘email’ to prevent other input, also forces input for important data entries.
Dictionary objects	Unlisted dictionary objects were used to store values so that they could be called for use to display their text without order, they were also converted to JSON objects through methods to be further used.
Utilized Library methods	The Flask Python library was used to create URLs and web service for the program to connect to and store data. The BotUI javascript library was used to create the front end.
Inserting data into a file	After data is parsed, it is written into the app.py file where it is stored after changing its format and adding it to an array.

Style of web service

The solution utilizes an online Python RESTful API and a frontend javascript and shell to store and access the data for the client and customers. I chose to create a restful API format for my web service because they provide a **great deal of flexibility in terms of data handling**, as well, they are stateless, therefore being more able to handle large volumes of users. I believe that these functions give it **more potential to develop** if the service grows in the future. The different calls to different URLs used in the RESTful format API allow the client to 'GET' (view data), 'POST' (add data), 'PUT' (modify data), and 'DELETE' (remove data).

```python
#delete an item, must autheticate
@app.route("/items/<int:item_id>", methods=["DELETE"])
@auth.login_required
def delete_item(item_id):
    item = [item for item in items if item["id"] == item_id]
    items.remove(item[0])
    return jsonify({"deleted": True})
```

```python
#create an item, must authenticate
@app.route("/items", methods=["POST"])
@auth.login_required
def create_item():
    #format of item created by method
    item = {
        "id": items[-1]["id"] + 1,
        "name": request.json.get("name", ""),
        "info": request.json.get("info", ""),
        "unavailable": False
    }
    #add item to items array and return item
    items.append(item)
    return jsonify({"item": item})
```

```python
#GET calls
#calling get to /items gets list of items
@app.route("/items", methods=["GET"])
def get_items():
    x = {"items": items}
    return jsonify(x)
```

(examples of when each operation is used)

Choice of Python libraries

I utilized the flask python library to create the API which stores information. This is because the flask library maintains the barebones of a web structure with its methods while allowing the developer more freedom as well as more insight into how the program worked than a higher level library such as Django. I feel that this was more appropriate for this project as I

was able to modify some basic functions as the data structure which stored the items, giving me **more customization on how each methods** worked.

```
#import modules from flask for api
from flask import *
```

Using the flask library I am able to launch this program online using an online hosting service using a Gunicorn web server.

```
HTTP/1.1 200 OK
Connection: keep-alive
Server: gunicorn/19.9.0
Date: Mon, 04 Mar 2019 11:13:01 GMT
Content-Type: application/json
Content-Length: 148
Via: 1.1 vegur
```

The library allows the program to securely operate through the HyperText Transfer Protocol Secure (HTTPS) and requires authentication for modification of items or removal of orders. The program has separate URL links for each of the functions which **allow the client to easily modify information stored in the program by accessing different links.** This is extremely **useful for adding different functions to the web service**, as a separate URL can simply be added for every different function that the client wishes to add in the future.

```
@app.route("/orders",
```

In order to prevent modification of the item by an unauthorized user, I used the library flask_httpauth's HTTPBasicAuth method in order to enforce a password on every call to the web service.

```
from flask_httpauth import HTTPBasicAuth
```

```
#username and password required for viewing and modifying database
@auth.get_password
def authenticate(u):
    if u in users:
        return users.get(u)
    return None

#message for wrong username/password
@auth.error_handler
def unauthorized():
    return make_response(jsonify({"error": "Access Denied"}))
```

```
#modify items, must authenticate
@app.route("/items/<int:item_id>", methods=["PUT"])
@auth.login_required
```

JSON Objects and Data structures

The library also allows the objects to be stored in the javascript object notation (JSON) format.

```
items = [
    {
        "id": 1,
        "name": u"Barite",
        "info": u">90% BaSO4",
        "unavailable": False
    },
    {
        "id": 2,
        "name": u"Silica Sand",
        "info": u">99.5% SiO2",
        "unavailable": False
    }
]
```

(items stored in JSON format)

```
@app.route("/items", methods=["POST"])
@auth.login_required
def create_item():
    #format of item created by method
    item = {
        "id": items[-1]["id"] + 1,
        "name": request.json.get("name", ""),
        "info": request.json.get("info", ""),
        "unavailable": False
    }
    #add item to items array and return item
    items.append(item)
    return jsonify({"item": item})
```

(adding new information so that it is easily readable by the client and easily accessible to code)

This method of storing the item allows the **association of multiple pieces of information about the product to be stored in a two-dimensional array** so that any piece can be accessed by the client (eg. information and availability of an item). Information can be inserted and written as well as deleted from the program. Using a **for loop, the program allows the modification of items stored in the array by iterating through the array and altering the information attached to the item that shares the same id as the item selected.**

```
@app.route("/items/<int:item_id>", methods=["PUT"])
@auth.login_required
def modify_item(item_id):
    item = [item for item in items if item["id"] == item_id]
    item[0]["name"] = request.json.get("name", item[0]["name"])
    item[0]["info"] = request.json.get("info", item[0]["info"])
    item[0]["unavailable"] = request.json.get("unavailable", item[0]["unavailable"])
    return jsonify({"item": item[0]})
```

(the client is able to modify item information calling the “put” method to /items/item#)

Essentially the program performs a **search through the items listed and modifies the one selected.** By creating new JSON objects, **the program allows the client to add as many items as needed and remove them at will.**

The separate client modification file

As of client request, the modification program was removed from the javascript and put in a format that the client felt was more appropriate. In order to separate customer and client use of the API, I created a separate program for the client to access information than the customer so that it **lowers the chance a customer can enter and modify items.** The customer accesses the information through a **chatbot interface created using javascript** while the client accesses the API **using a shell script**. The shell script was made for the client's system (Unix on a MacBook air). Through shellcode, the client is able to **modify orders and items extremely quickly**. In order to prevent items from being modified or orders being removed by an unauthorized party, these actions require the client to a**uthenticate themselves through a login**. The client is prompted by the code to enter the necessary fields, making it easy and fast for them to modify items.

```
#!/bin/bash

echo to modify item enter m, to delete item enter d, to add new item enter a, to
echo to place orders enter po, to view orders enter vo
read varname

if [ $varname == 's' ]
then
        curl -i https://ancient-bastion-34942.herokuapp.com/items
fi
```

(code behind client access)

```
to modify item enter m, to delete item enter d, to add new item enter a, to see items enter s
to place orders enter po, to view orders enter vo
a
please enter in name of item to add
example item
please enter in description
this is an example
please enter in username
clientuser
please enter in password
samplepassword
HTTP/1.1 200 OK
Connection: keep-alive
Server: gunicorn/19.9.0
Date: Mon, 04 Mar 2019 11:57:33 GMT
Content-Type: application/json
Content-Length: 88
Via: 1.1 vegur

{"item":{"id":3,"info":"this is an example","name":"example item","unavailable":false}}
Done
```

(client is given text prompts to quickly input data)

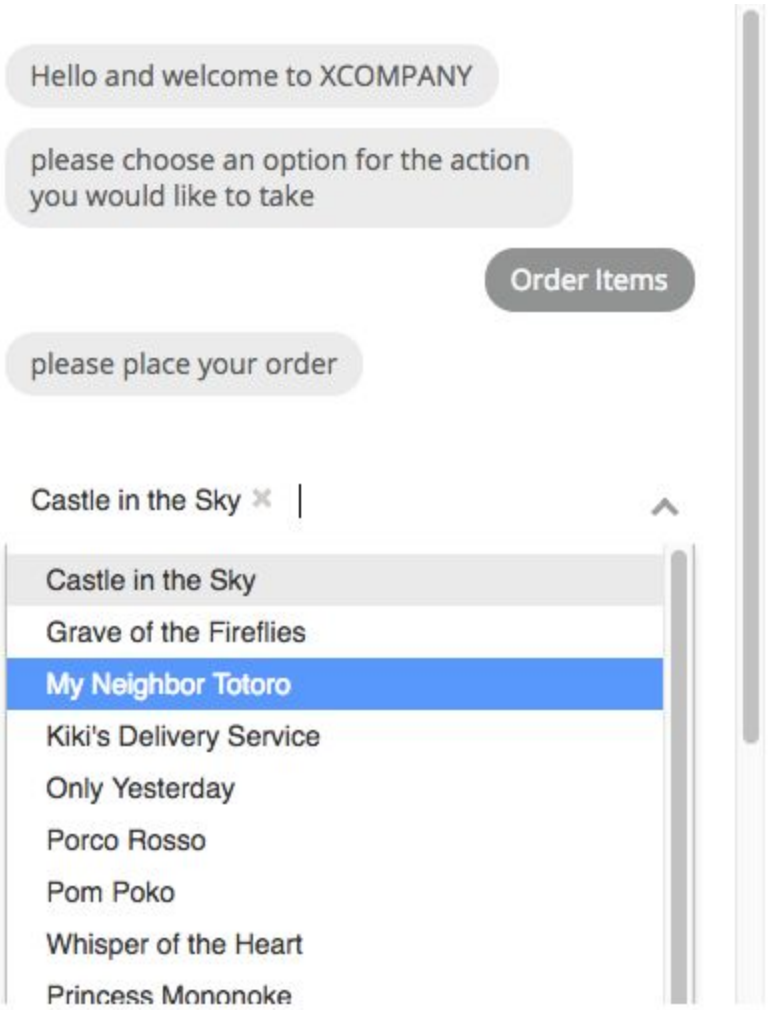
Javascript library and functions

I used the “botui” framework, an open source javascript framework to create the appearance of the customer interface. The interface gives customers a friendlier approach in purchasing goods and helps them fill out information. The chatbot **prompts users for information** and **provides them with options in a user-friendly manner**. The information that is **loaded from the API in the JSON format is processed in javascript** in order to take the information. After taking the orders of the customer, the information is **parsed and sent to the web service** through the ‘POST’ method, allowing the client to easily **access or modify this information.** The **information is modified by the program to be stored.**

```
function post(item, info){ //calls 'POST' to api to post items
  var itemstr = "";
  //function to load items from api
  var request = new XMLHttpRequest();
    var url = 'https://ancient-bastion-34942.herokuapp.com/orders'//orders url of api
    request.open('POST', url);
    request.setRequestHeader("Content-Type", "application/json");
    var data = {"item":item, "info":info };
    request.onreadystatechange = function () {
    }
    request.send(data);
    botui.message.add({
      content: "thank you for your order." //thanks the customer
    })
}
```

In order to load the items of the items into a scroll down menu, the items are split from the display string (loaded earlier for viewing) into an array where after iterating, **is reformatted then added to a new array so that it can be used by the botui framework.** This method worked well in that it created a drop-down menu that the client could update instead of a text prompt that could lead to more data entry error.

```
var temp = list.split("+=+"); //split string into items
for (i=0;i<temp.length-1;i++){ //puts items into menu
var temp_dict = {value: temp[i].substring(0,2), text: temp[i]}; //formats in the form of a dictionary
choices.push(temp_dict); //adds names of items as options from drop down menu
}
botui.action.select({
    action: {
      placeholder : "Select An Option",
      multipleselect : true,
      value: choices[0].value, // default selected value is first item in choices
      options : choices,
      button: {
        icon: 'check',
        label: 'Select'
      }
    }
```



(drop-down menu loading a large number of names from an example list put in the back end)

Word count: 1092

Sources:

Anon, <input>: The Input (Form Input) element. *MDN Web Docs*. Available at: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input [Accessed January 4, 2019].

Anon, BotUI. *BotUI*. Available at: https://docs.botui.org/ [Accessed January 4, 2019].

Anon, Welcome. *Welcome | Flask (A Python Microframework)*. Available at: http://flask.pocoo.org/ [Accessed January 4, 2019].

Anon, Welcome to Flask-HTTPAuth's documentation!¶. *Welcome to Flask-HTTPAuth's documentation! - Flask-HTTPAuth documentation*. Available at: https://flask-httpauth.readthedocs.io/en/latest/ [Accessed January 4, 2019].

Anon, *[A GNU head]*. Available at: https://www.gnu.org/software/bash/manual/bash.html [Accessed January 4, 2019].

Anon, Shell Scripting Tutorial. *Shell Scripting RSS*. Available at: https://www.shellscript.sh/ [Accessed January 4, 2019].

Cook, C.E., 2005. *Blue Pelican Java*, College Station: Virtualbookworm.com.

Horstmann, C.S., 2015. *Big Java*, New York: Wiley.