Project Report

Rest API for garment article price (GAP)

Rafat Khan
rafatk@kth.se

Table of Contents

Revision History

Name	Date	Comment	Version
Rafat Khan	17-Mar-2022	Initial Version	1.0

1. Introduction

1.1 Purpose

The purpose of this document is to present a detailed description of the GAP. It will explain the purpose and features of this application, what the application will do, design, implementation, testing process and the constraints under which it must operate. This document is intended for the business and technical teams and also potential developers.

1.2 Document Conventions

This Document was created based on the IEEE template for System Requirement Specification Documents.

1.3 Intended Audience and Reading Suggestions

- ✓ Business and technical teams.
- ✓ Programmers who will work on the project by further developing it or fix bugs.

1.4 Project Scope

GAP is a web based Rest API for garment article price retrieval, which will be developed to fetch article price based on different parameters.

1.4.1 Overall Description

The purpose of this project is to implement a REST API. I will build the API server and a client program to call the API. The implementation will be simple, with no specific design pattern, security and encryption process. I will use python3 as the programming language and SQLite as the database. The server will run using the default address and serve only the GET request from the client. After receiving a request, the server will fetch data from the database by executing query and return it in JSON format. The client program only contain one method, that will make an API call with three parameters and will show the response.

1.5 Product Perspective

Clients of GAP will be able to make API calls with selected parameters that will return the price of the selected article.

1.6 Product Features

Data Fetch Operation

The client will be able to search and find article prices based on article, color, and week. Fetched data might be kept as session data for as long as it's available, which will make the data easy to access and prevent the API from being overloaded.

1.7 User Classes and Characteristics

There will be only one class of users. The users will use the API for fetching article price from the database.

1.8 Operating Environment

Server: Microsoft Azure/ Amazon Web Server/ Google Cloud or similar
Client:
Device: Any
OS: Any
Browser: Any Browser (Chrome, Firefox or Safari is preferable)

1.9 Design and Implementation Constraints

Both the client and server must be connected to the internet in order to use the API.

2. System Features

In this section, we demonstrate highlights of the application, how they can be used, and the results they can produce for the client.

2.1 Client Features

REQ-GAP-1: Fetch Article Price for REST Client

Description

Any client can access the API and search for the price of the selected article using parameters like article, color and/or week.

Stimulus/Response Sequences

In the client application of GAP, User will have to follow the following direction:

Action	Performed By	Response	Performed By
Call the GAP API service with required parameter values	Client	Article details price with given parameter values will be shown	API Server

Functional Requirements

In the current scope, a user can fetch the price of an article based on three parameters, article, color and week. Within the parameters, article and week is mandatory. Color can be used as an additional filtering criteria.

2.2 Server Features

REQ-GAP-2: Return Article Price by REST Service

Description

While an API call is made by the client, the server will fetch the relevant data from the database and return it in JSON format.

Stimulus/Response Sequences

Action	Performed By	Response	Performed By
Request for pulling article price	Client	Server will return relevant data.	Server

Functional Requirements

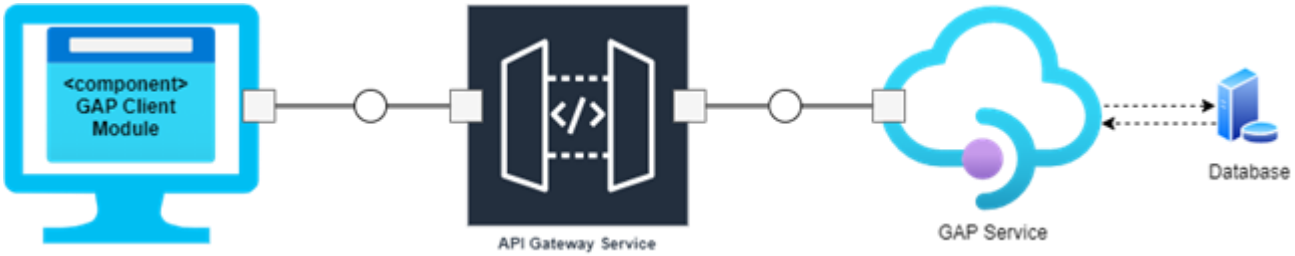
While client creates a request by calling the API, server will fetch the data from the database and return it to the client in JSON format, which will be used for serializing and transmitting structured data over network connection.

3. System Architecture

3.1 Overview

To find article prices, the client system will allow the user to make an API call. An option will be provided to select the value of some predefined parameter.

If the client makes an API call, the server will retrieve relevant data from the database based on the parameter values and return it to the client in JSON format.



3.2 Development Technology

Development Language: Python 3
Architecture: Representational State Transfer (REST)
Framework: Flask
Database: SQLite

4. Implementation & Code Review

4.1 REST Client

To write code that interacts with REST APIs, I have used python requests to send HTTP requests. This library abstracts away the complexities of making HTTP requests.

In this iteration, only GET method has been implemented. GET is one of the most common HTTP methods that we use when working with REST APIs. This method allows us to retrieve resources from a given API.

To complete the API call, I am running an API server in localhost, using the port 5000. This code calls requests.get() to send a GET request to the server by calling to the price function using three parameters, which responds with the price item with the provided parameter values. Then I have called .json() on the response object to view the data that came back from the API.

Implementation of the function is as follows:

```
def get_article_price(article, color, week):
    parameters = {'article': article, 'color': color, 'week': week}
    api_url = "http://127.0.0.1:5000/price"
    try:
        response = requests.get(api_url,parameters)
        if(response.status_code == 200):
            print(response.json())
        else:
```

```
        print("An error occurred")
    except:
      print("An exception occurred. Can not connect to the server.")
```

The response data is formatted as JSON, a key-value store similar to a Python dictionary. It's a very popular data format and the de facto interchange format for most REST APIs.

4.1.1 Error Handling

To verify if the API server is accessible from the client end, I have used try except block to handle selected exceptions. The try block lets us test a block of code for errors and except block let us handle the error. In future, this can be enhanced to detect specific kind of exceptions.

Another problem that can arise in case of faulty response by the server. Usually the server reply contains a status code that indicate the status of the request. In our scenario, only get operation is being called, therefore we can expect only success code (200) as response. Any other response might indicate an error.

4.2 REST Service

This is the server. In this implementation, I have used Flask-RESTful web framework. Flask-RESTful is an extension for Flask that adds support for quickly building REST APIs. It is a lightweight abstraction that works with our existing ORM/libraries. Flask-RESTful encourages best practices with minimal setup.

Implementation of the server is as follows:

Code	Explanation
from flask import Flask, request, jsonify, session from flask_restful import Resource, Api from sqlalchemy import create_engine from json import dumps from flask_restful import reqparse	Import the classes
db_connect = create_engine('sqlite:///article.db') app = Flask(__name__) api = Api(app)	Connection request with the database Create an instance of the class

class ArticlePrice(Resource):
  def __init__(self):
  self.reqparse = reqparse.RequestParser()
  self.reqparse.add_argument('article', type = int, default='')
  self.reqparse.add_argument('color', type = int, default='')
  self.reqparse.add_argument('week', type = int, default='')
  super(ArticlePrice, self).__init__()
	Defining parameters using RequestParser()

def get(self):
  args = self.reqparse.parse_args()
  conn = db_connect.connect() # connect to database
  query = """
  Select i.article_name,c.article_color_name,p.week,p.article_price
  From article_price p
  Inner Join article_item i on i.article_id=p.article_id
  Inner Join article_color c on c.article_color_id=p.article_color_id
  Where p.article_id=? and p.article_color_id=? and
	Find the parameters
	Connect to SQLite database
	
	SQL query to fetch data

p.week=? """ data = conn.execute(query,(args['article'],args['color'],args['week'])) result = {'data': [dict(zip(tuple (data.keys()) ,i)) for i in data.cursor]} return jsonify(result)	Performs query and returns json result Make your output JSON serializable
api.add_resource(ArticlePrice, '/price')	Decorator to tell Flask what URL should trigger the function
if __name__ == '__main__': app.run()	Run the Flask application (default port)

4.2.1 Error Handling

For handeling the error and exceptions, the query execution process has been modified and try except block has been introduced.

```
try:
    data = conn.execute(query,(args['article'],args['color'],args['week']))
    result = {'data': [dict(zip(tuple (data.keys()) ,i)) for i in data.cursor]}
    return jsonify(result)
except DoesNotExist:
    raise DataNotExistsError
except Exception:
    raise InternalServerError
```

Moreover, a python file named error.py has been created which keeps a list of all possible kind of error and exceptions along with status code and custom message.

5. Testing

To test the REST Client call, I have executed the client.py program using Jupyter Notebook and for testing out GET method in REST Service section, I have used a service called Postman. Postman is an application used for API testing. It is an HTTP client that tests HTTP requests, utilizing a graphical user interface, through which we obtain different types of responses that need to be subsequently validated.

In both case, the same API will be called.

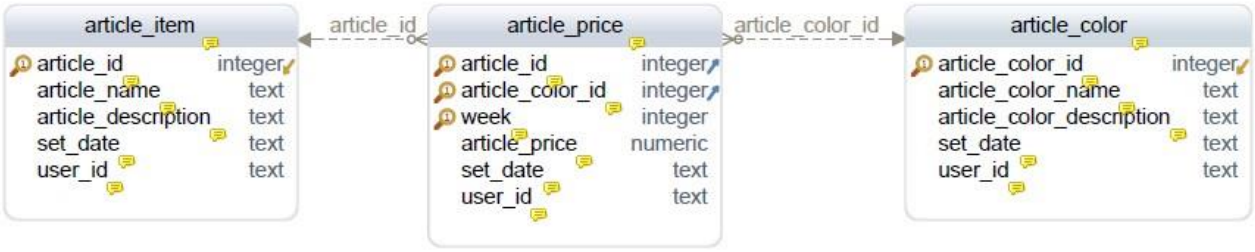
Test Case ID	TEST-GAP-1	Requirement ID	REQ-GAP-1
Executed By	Rafat Khan	Execution Date	17-Mar-2022
Test Case Description: We will check if we can make the API call from the client program by calling the get_article_price() function with different given parameter values and receive appropriate response.			
Test Data	Expected Result	Actual Result	Pass/Fail
get_article_price(article=1,color=1,week=1)	API call: http://127.0.0.1:5000/price?article=1&color=1&week=1 Data: Single article information including price	API call: http://127.0.0.1:5000/price?article=1&color=1&week=1 Data: Single article information including	Pass

	for <article-id> 1 and <color-id> 1 and <week> 1.	price for <article-id> 1 and <color-id> 1 and <week> 1	
get_article_price(article=1,color=0,week=1)	API call: http://127.0.0.1:5000/price?article=1&color=0&week=1 Data: Multiple article information including price for <article-id> 1 and <week> 1. Data will contain all colors.	API call: http://127.0.0.1:5000/price?article=1&color=0&week=1 Data: Multiple article information including price for <article-id> 1 and <week> 1. Data will contain all colors.	Pass
get_article_price(article=1,color=0,week=0)	API call: http://127.0.0.1:5000/price?article=1&color=0&week=0 No data.	API call: http://127.0.0.1:5000/price?article=1&color=0&week=0 No data.	Pass
get_article_price(article=0,color=0,week=0)	API call: http://127.0.0.1:5000/price?article=0&color=0&week=0 No data.	API call: http://127.0.0.1:5000/price?article=0&color=0&week=0 No data.	Pass

Test Case ID	TEST-GAP-1	Requirement ID	REQ-GAP-2
Executed By	Rafat Khan	Execution Date	17-Mar-2022
Test Case Description: We will check if the API call works with different given parameter values.			
Test Data	Expected Result	Actual Result	Pass/Fail
http://127.0.0.1:5000/price?article=1&color=1&week=1	Single article information including price for <article-id> 1 and <color-id> 1 and <week> 1.	Single article information including price for <article-id> 1 and <color-id> 1 and <week> 1.	Pass
http://127.0.0.1:5000/price?article=1&color=0&week=1	Multiple article information including price for <article-id> 1 and <week> 1. Data will contain all colors.	Multiple article information including price for <article-id> 1 and <week> 1. Data will contain all colors.	Pass
http://127.0.0.1:5000/price?article=1&color=0&week=0	Empty data set.	Empty data set.	Pass
http://127.0.0.1:5000/price?article=0&color=0&week=0	Empty data set.	Empty data set.	Pass

6. Database Design

To work with the current implementation, I have created a SQLite database with three tables. The article_item table holds the information about articles, article_color table holds the list of colors and the article_price table holds the price by article_id, article_color_id and week.



7. Further Extensions

At the moment, we are using only a GET request with three parameters. In order to extend further, such as changing the parameter pattern, new parameters must be defined in both client and server implementations. To minimize further work, possible parameters can be defined and the solution should return values that are based on default parameter values.

8. Managing Heavy Load

If the number of requests for the same articles is high, e.g., 1000 times a day, etc., what can we do make the rest API efficient?

Server-side cache
In this request, I would make a single call to my server to "fetch" all the data for all sources. The data would then be cached on the server. The client would then have the same REST endpoints as before, except there wouldn't be much waiting between calls since my server already has the data and just has to feed it to the client.

Pros: Still easy to implement on the client side, but without the latency issues.
Cons: A bit more involved server side, and the first call could take long time to execute.

Client-side cache
This scenario is similar to the previous one except the client only ever makes one request to the server: give me all of the data. From here it's the client's responsibility to save the data and use it appropriately.

Pros: Easy server implementation, very speedy after the first call.
Cons: First call will be very slow, more complicated client-side implementation.

Multiple Endpoints
This can be another solution of this problem. In this approach, I would have multiple endpoints on my server: one to get the list of articles with price, one to get only the price etc.

9. Deployment

Azure
To deploy the solution at Microsoft Azure, I will create a web app using Azure App Service in using the Azure CLI. Then, by configuring a Git remote in my local repository that points at Azure, I can push my application code from the local repository to Azure. Either the Azure portal or the Azure CLI can be used to retrieve the URL of the remote repository and the Git credentials required for configuration.

AWS

For this purpose, I will use AWS App Runner service. From my GitHub repository, I will fork the repository to my GitHub account. Then, I will clone the repository to my AWS CloudShell console and navigate to the cloned directory. Using the Pipeline GitHub.json file in the cloned repo directory, I will create the pipeline that is invoked when a commit is made.

After that, I will set up the IAM roles necessary for App Runner. IAM is used by App Runner to interact with other AWS services. Now I need to sign in to the AWS console to setup the App Runner service. As a repository type, I will select Container Registry, provider as Amazon ECR, and set the Image repository as python-app and the Image tag as latest. Then I will set up the deployment, service and security configuration.

In both cases, for database instance, I will setup and install SQLite server using SQLite server image from the cloud marketplace.

Appendix A: Glossary

System	PMS Application.
GAP	The name of the application. Stands for Garment Article Price.
Users	General users who will use the REST Client application.
Server	REST Service.
SQLite	Database.