Note to TAs: Updated to final Sprint docs. Some endpoints in the graphs might be changed due to billing issues, please reach out to us for specific links.

Project ZyfiJade

E6156 - Topics in SW Engineering Management: Cloud Computing Team and Project Management System

Focal Point: Ordonez Chaguay, Jordan, jio2108
Team Members:Yuyang Li, Jordan Israel Ordonez Chaguay ,Zhixuan Zhang, Yao Jia, Faquan Wang

Introduction

Within a manufacturing company it can be difficult to keep track of the following

- The formation of clients' profiles.
- Creating and tracking products orders, and managing related information..
- Allocating and replenishing raw materials.

ZyfiJade is a cloud-native application that implements functions for managing the orders and its materials.

Personas and Roles

For our project there are 2 personas Employee and Client.

Employee can take on the following roles:

1. Salesman is someone conducting a sell order for their respective company on behalf of a client.
2. Manager is someone in charge of reviewing orders for their respective company placed by the salesman.
3. Purchaser is someone in charge of replenishing the stock for a particular raw material for their respective company.

A Client:

1. Non-employee who has placed an order to be fulfilled.

Key Features

- The salesman:
    - Order is applied by them
    - Responsibility is to sell the product for their respective company
    - They are connected to the client
    - Through an API call to the Order Microservice they put the order for a client ID, number of goods, etc
    - Produce a profile for the client and give it to the client.
    - Update the profile for the client
- Manager
    - The responsibility of the manager
        - Review orders
            - Through an API call to the Order Microservice their able to confirm the order regardless of the number of finished goods
            - Another field: Is_finished
                - First case: Finished goods(num) > the number placed in the order→ checks off Is_finished
                - Second case: Finished goods(num) <= the number placed in the order →
                    - Subtract the quantity from the raw material and add the finished goods no matter what.
                    - Then checks off Is_Finished
- Purchaser
    - Responsibility:
        - Through an API call to the Product microservice they query and adds the raw material quantity

- Client
    - Gets the profile from the salesman and uses this profile to get their history order
    - Queries the client database
- Databases:
    - Client Database
    - Order Database
    - Product Database

Classes:

1. The **salesman**:
    a. Class Salesman
        i. Functions
            1. def __initi__(self, id, name)
                a. self.id = id
                b. self.name = name
            2. def connectToOrderMicroservice(self):
                a. # returns a connection to the order microservice
            3. def placeOrderForAClient(self, client_id, order):
                a. # returns an client_id and order_id
            4. def connectToClientDatabase(self):
            5. def createClientProfile(self, client_id):
                a. # return 1 or 0 for successful/non successful
            6. def updateClientProfile(self, client_id):
                a. # return 1 or 0 for successful/non successful
            7. def retrieveSuccessfulOrderClientID(self):
                a. # returns the client id for a successful order
        ii. Fields
2. The manager
    a. Class Manager
        i. Functions

```
class Manager :
def __init__(self, id, name) -> None:
self.id = id
self.name = name
def getOrderList(Client_id) :
orderList = request_get(Client_id, ...)
def confirmOrder(Order_id) :
comfirm = request_post(id, True or False)
def getProductNum (Product_id) :
productNumber = request_get(ProductId,...)
def isFinished(Order_id) :
if getProductNum(Order_id) > number_placed_intheOrder:
      return True
return False
def getNumberPlaced (Product_id):
      number_placed_intheOrder = request.get(Product_id, …)
```

        ii. Fields
        id
        name

3. Client
    a. Class Client
        I. Functions

```
class Client :
def __init__(self, id, name) -> None:
        self.id = id
        self.name = name
def getHistoryOrder(id, profile) :
        HistoryOrder = request_get({id, profile}, …)
```

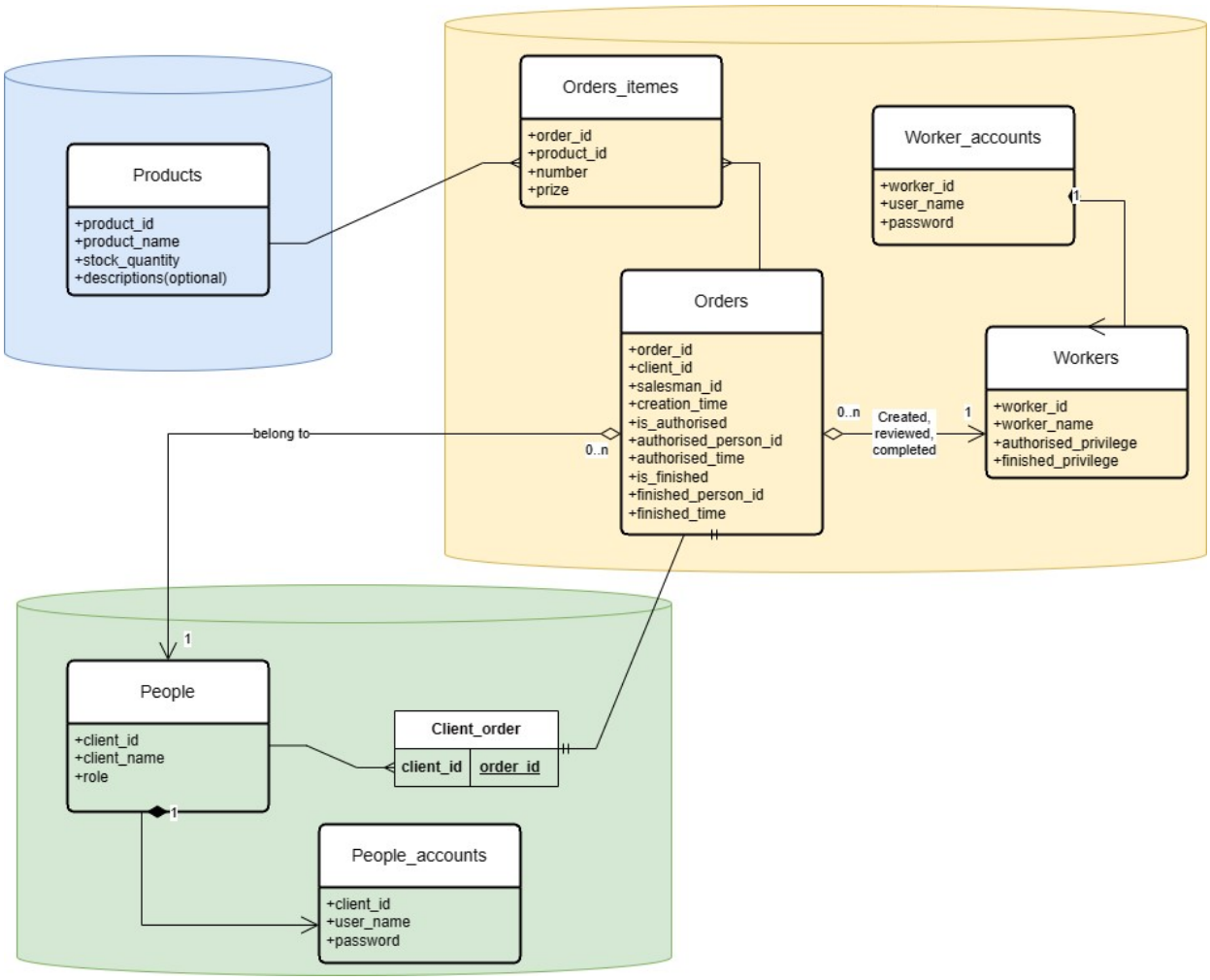
        ii.Fields
            id
            name
4. Purchaser
    a. Class Purchaser
        i. Functions:
            ①**def __init__(self,id,name):**
                self.id = id
                self.name = name
            ②**def get_raw(self,product_id):**
                while(!get_product(product_id,self.id)):
                get_product(product_id, self.id)
            ③**def add_raw(self,product_id):**
                while (!set_product(product_id, self.id)):
                set_product(product_id, value, self.id)

            ④**def get_product(product_id,self.id)**
                return boolen
            ⑤**def set_product(product_id, self.id)**
                return boolen

        ii.Fields
            ①id
            ②name

Domain Model

A domain model documents and exposes the “concepts” or “business objects” that users interact with and manipulate to use a system to accomplish tasks, goals, user stories, … The following is an initial overview of ZyfiJade’s domain model.



The three colours represent three different services. The rounded square represents the preparation of a relational database(MySQL) and the square square indicates the preparation of a non-relational database(redis).

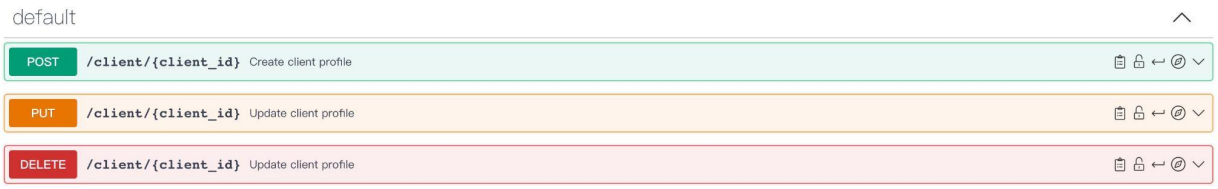
For the Client service, we have implemented anti-normalisation and push mechanisms. When the customer wants to view order information, we display only the order ID on the initial page. Further details are retrieved from MySQL only after the customer taps on the order ID. The order ID is already stored in Redis, utilizing its memory-based storage mechanism to minimize page load times for customers. This is achieved by reducing wait times through foreign key joins, thereby decreasing pressure on the server.

Resource Paths

ZyfiJade exposes a set of REST resources. The following is the initial set of resource paths:

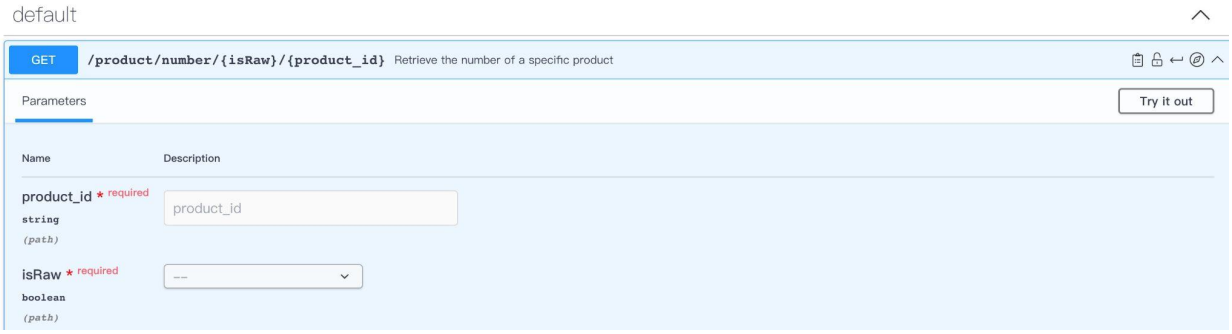
- Login Sections:

- Client :

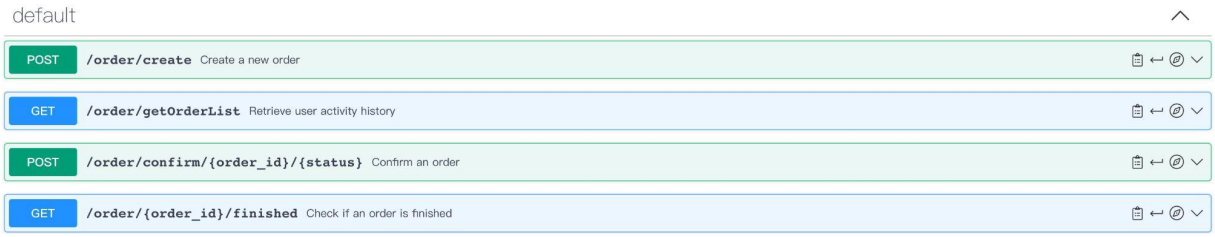


- Connect to Microservice:

- Product:



- Order

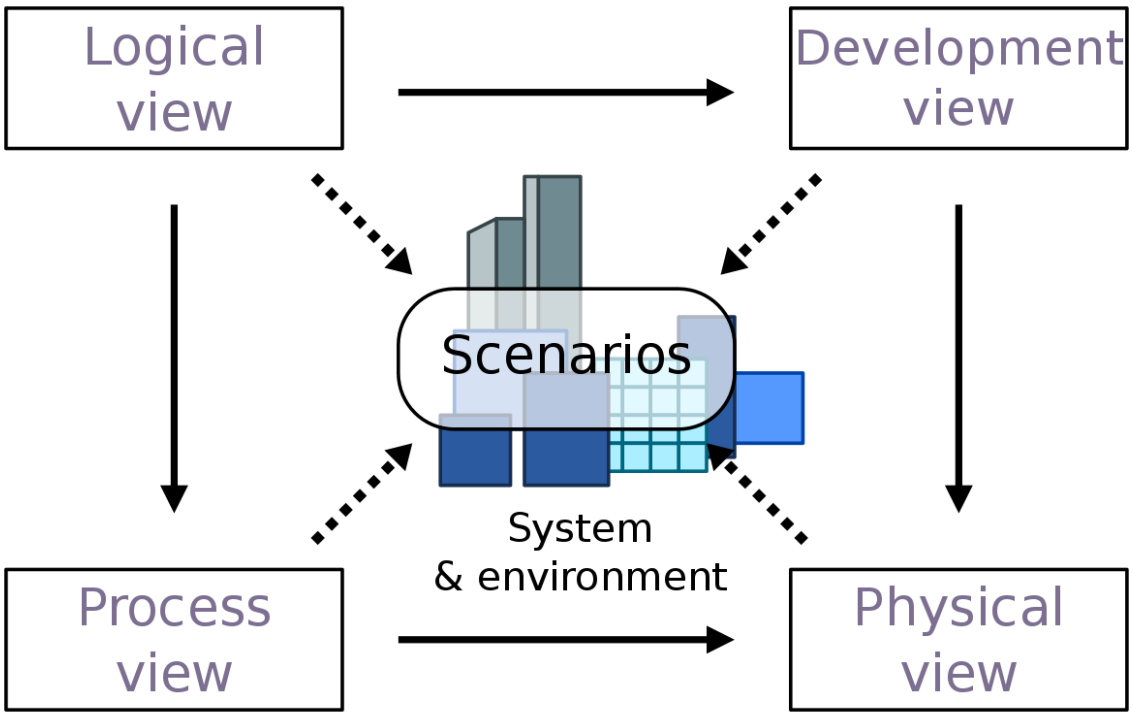


<dff Todo>
Document query requirements, pagination, linked data, HATEOAS, … …
</dff Todo>

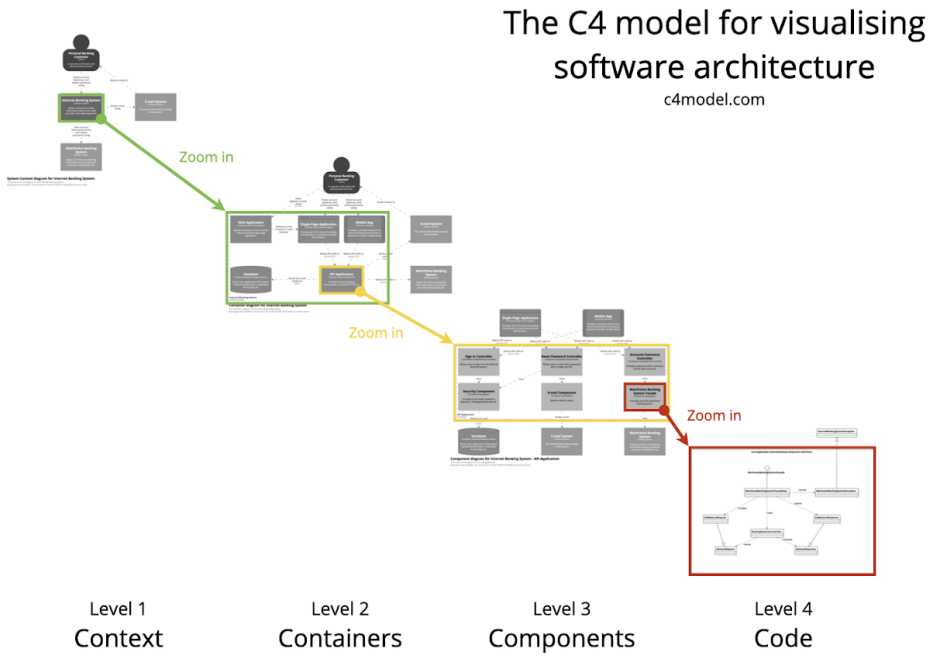
Architecture

Architecture Model

The 4+1 Architecture View Model provides a framework for making architecture decisions and documenting the architecture. ZyfiJade uses a subset of the approach and will be more complete over time.

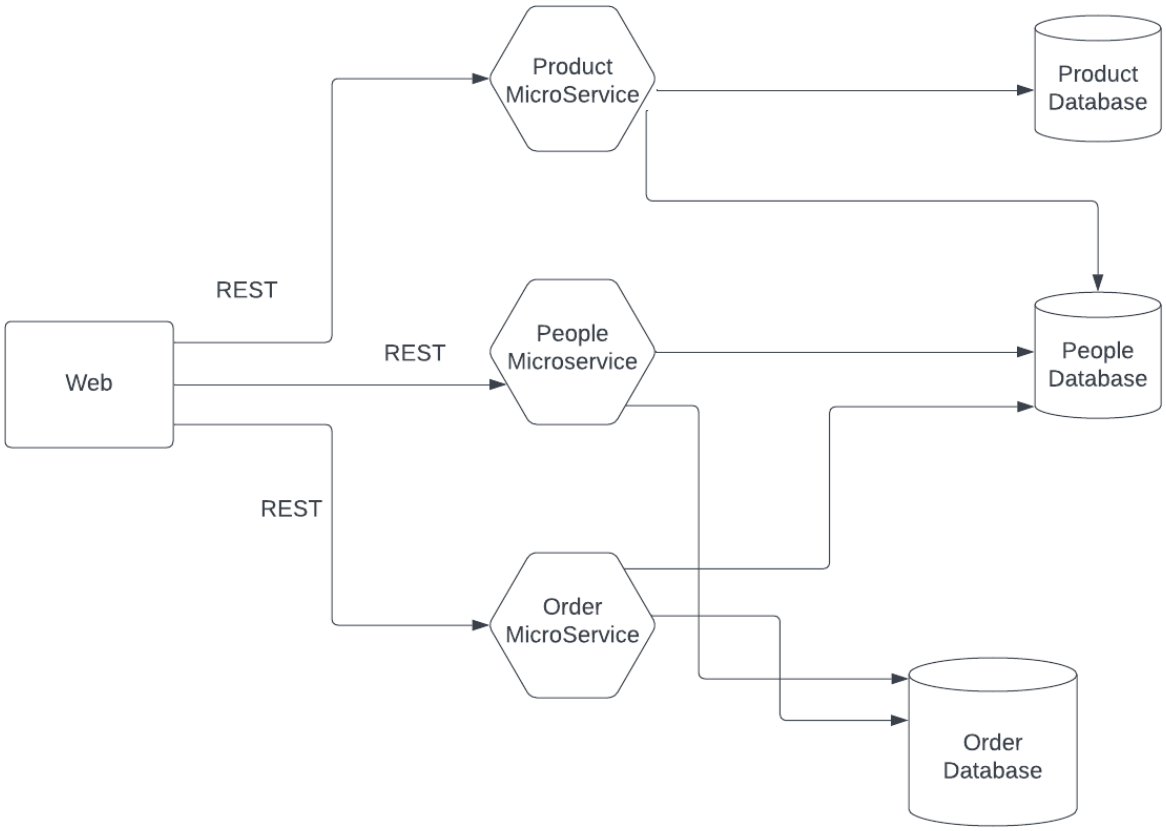


The C4-Model provides notation for visualizing architecture. ZyfiJade will adopt in future documents.



Logical View

A basic concept in my logical view is following the [Hexagonal Architecture].



Microservices explanations:
People:

- Where hosted: We have spun up an **EC2** instance. Here is the current endpoint http://54.226.116.254:8080/clients
- Main Function: Handle User Logins and Roles, expose different APIs and functions for users of different persona/roles to perform different tasks.
- MiddleWare necessary: database middleware, API gateway.
-

Order:

- Where hosted: We have created a **docker image** which is pushed on DuckerHub and deployed at GCP's Kubernetes Engine. Endpoint at http://34.171.158.6:8080/orders.
- Main Function: CRUD of Orders, and Confirmation of Orders, RPCs to Workers.
- MiddleWare necessary: Asynchronous data streaming middleware(for high throughput), database middleware, API gateway, (optional: Remote procedure call (RPC) middleware).

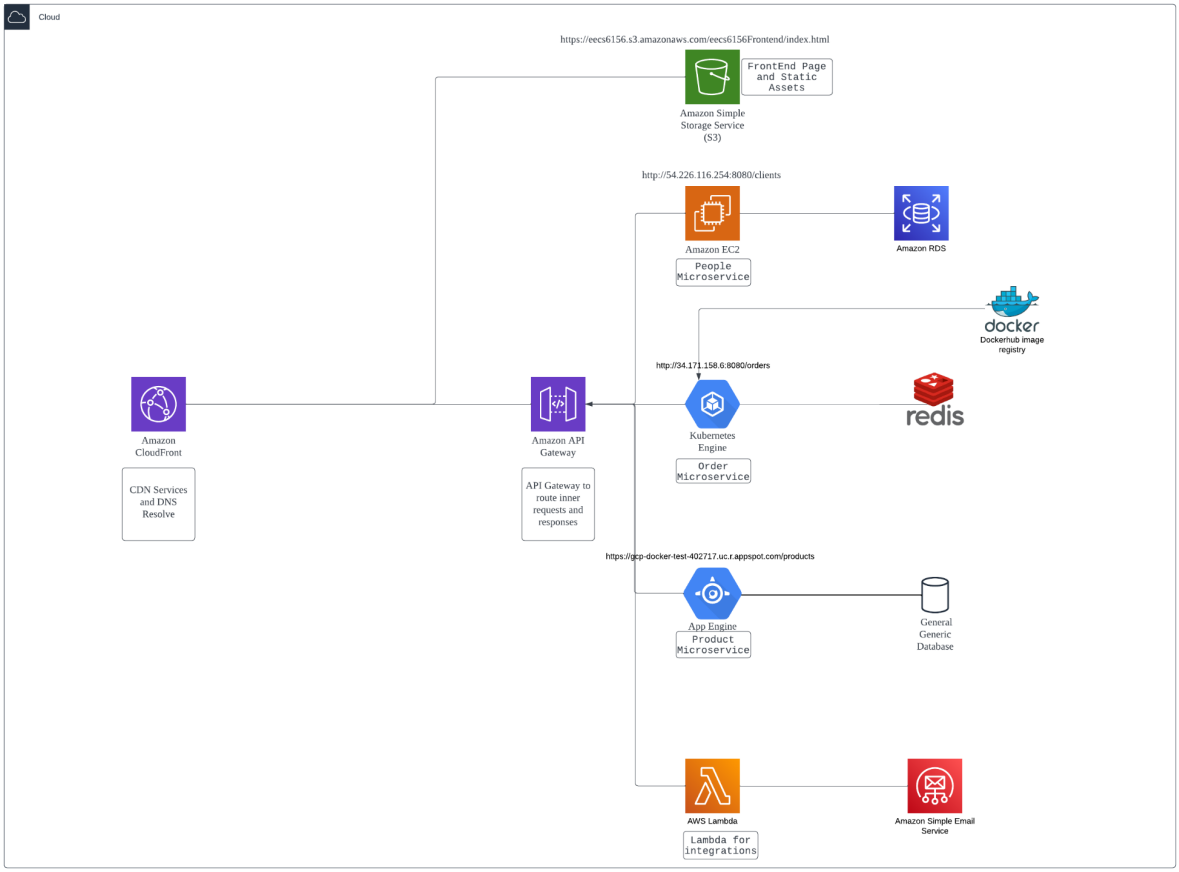
Products:

- Where hosted: **GCP's PaaS:** We used google's SDK to deploy the existing code, endpoint is at https://gcp-docker-test-402717.uc.r.appspot.com/products.
- Main Function: CRUD of Products, constant monitoring of the Stock number.

- MiddleWare necessary: database middleware, API gateway, Asynchronous data streaming middleware(for high throughput), Message-oriented middleware(Notifications when stock is low).

Physical View

This is the draft, in-progress physical view of the first few milestones.



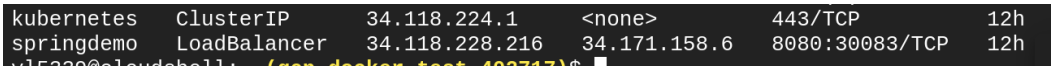
- Our project has a Frontend-Backend Architecture, the architecture can be divided into these four parts listed below.
    - Frontend: Responsible for user interaction, consisting of a web page and static assets.
    - Backend: Responsible for processing business logic and data storage/access, consisting of containerized applications, traditional virtual machines, and PaaS.
    - Storage: Used to store application data, using Amazon S3, Amazon RDS services and containerized redis.

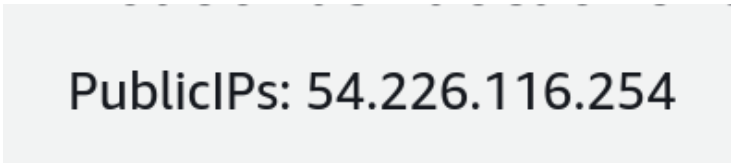
    - Networking: Used to connect frontend, backend, and storage resources, using Amazon and Amazon CloudFront(CDN) services.
- Whilst API Gateway acts as the middleman between the frontend and backend, on which authentication and authorization services are supported.

Codebases

- Project Board: https://github.com/users/Jordonez123/projects/1
- Order Microservice: Gear-dev-sudo/OrderMicroservice-REST (github.com)
- People Microservice Projects · 6156-GoodsInventoryManagementSystem (github.com)
- Frontend: https://github.com/GROJOEY/6156-GoodsInventoryManagementSystem
- Product Microservice: Gear-dev-sudo/ProductMicroservice (github.com)

Exception: Process 2 is still runningDeployment: Current EndPoints and URLs for Resources

- **FrontEnd / S3:** **https://eecs6156.s3.amazonaws.com/eecs6156Frontend/index.html**
- **Kubernetes / Google Kubernetes Engine**: Order Microservice on http://34.171.158.6:8080/orders
    - 

- **VPS / AWS EC2**: People Microservice on http://54.226.116.254:8080/clients(Not constantly on, If you want access, contact me at yl5339@columbia.edu, I'll get the VPS spinning for your inspection)
    - 

- **Paas / Google APP Engine :** Product Microservice on https://gcp-docker-test-402717.uc.r.appspot.com/products

Milestone Checklist

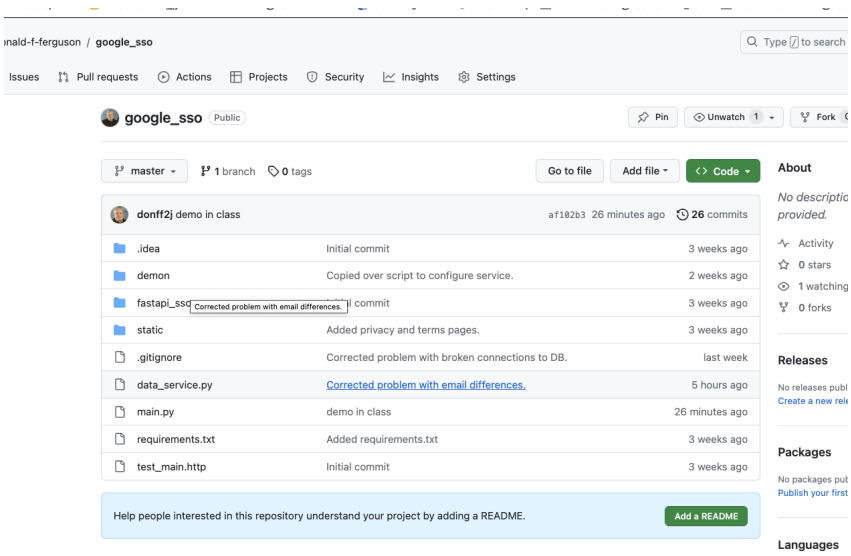
Milestone 2

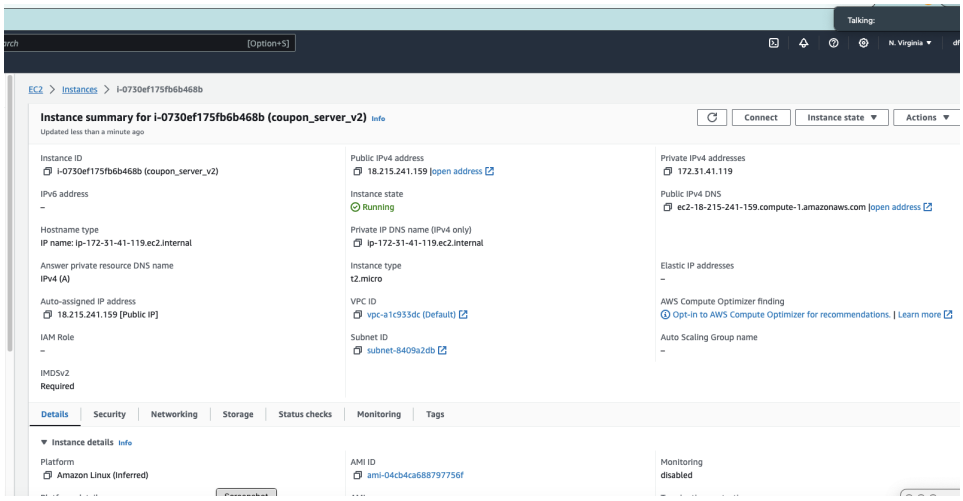
Class Member Microservice

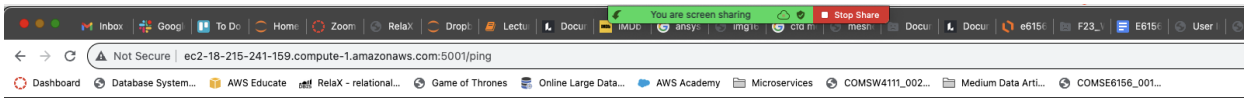
Tasks

- [ ] Create GitHub repo for microservice
- [ ] Create EC2 instance for microservice
- [ ] Implement microservice
- [ ] Deploy microservice on EC2

Artifacts







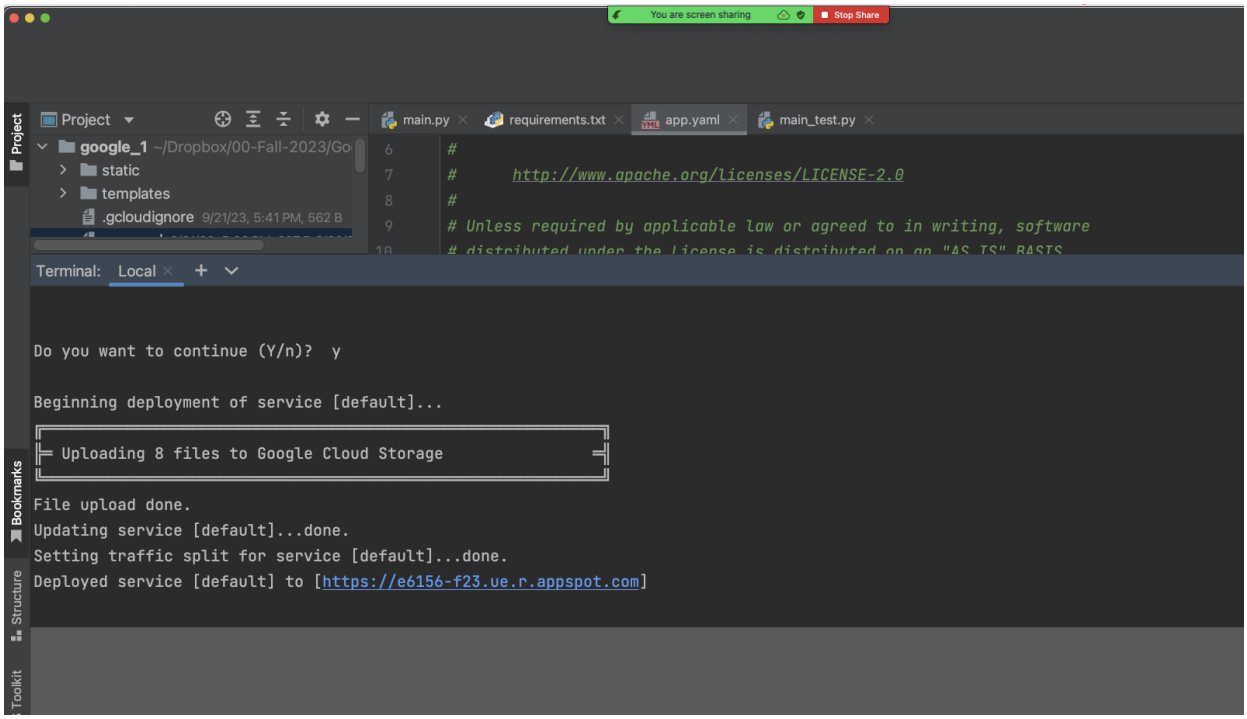
Pong.

Cool Stuff Microservice

Tasks

- [ ] Create GitHub repo for microservice
- [ ] Implement microservice
- [ ] Deploy microservice on Google App Engine

Artifacts

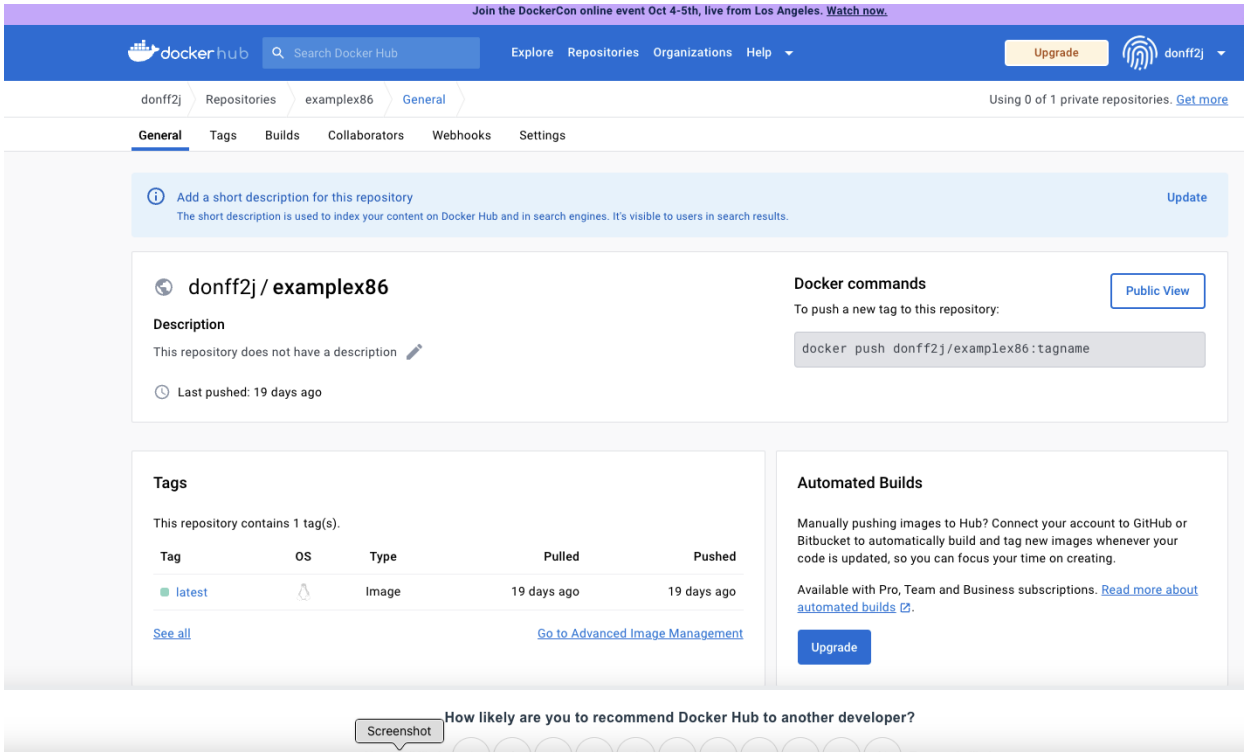


Not So Cool Stuff Microservice

Tasks

- [ ] Create GitHub repo
- [ ] Implement microservice
- [ ] Test container locally
- [ ] Push to Docker Hub
- [ ] … …

Artifacts

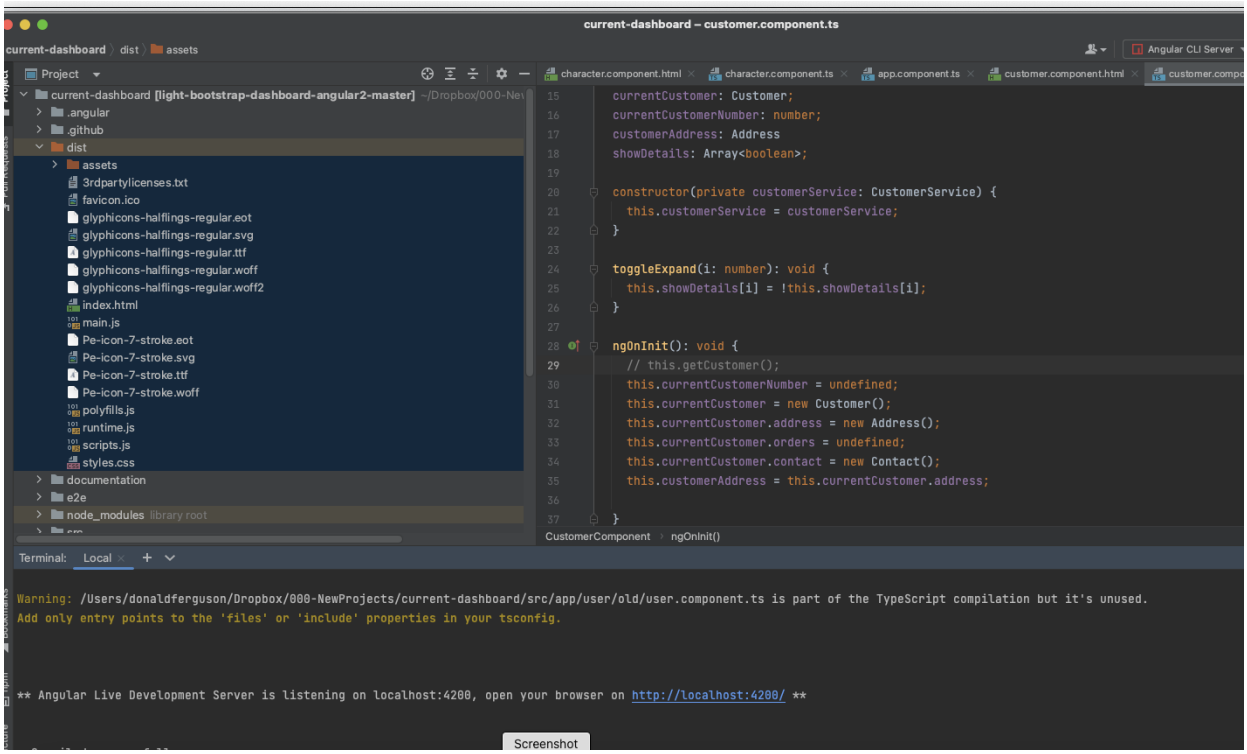


Static Web Site

Tasks

- [ ] GitHub repo
- [ ] Create S3 bucket
- [ ] Develop web front end
- [x] ~~Deploy to S3~~

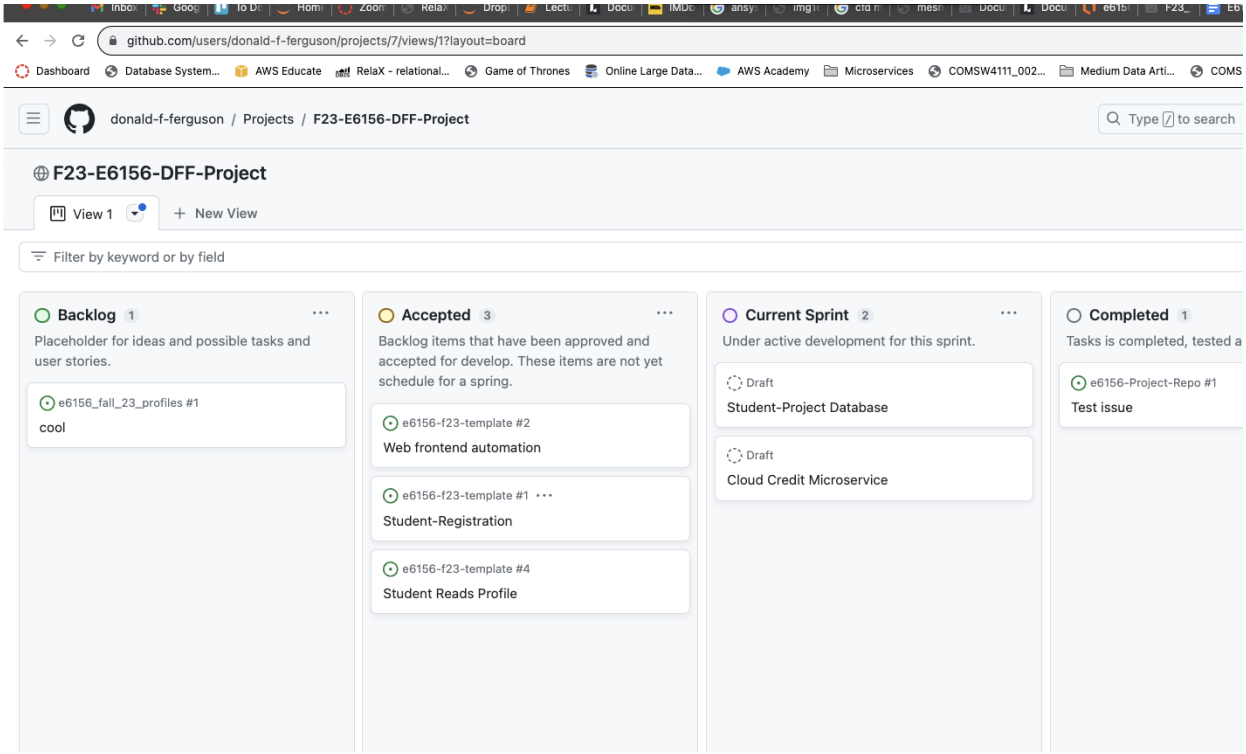
Artifacts



Agile Project

Tasks

- [ ] Create GitHub project
- [ ] Invite members
- [ ] Link repository
- [ ] Create Kanban with issues
- [ ] Initial … …



Sprint 2 Updates:

Videos Location:
https://drive.google.com/drive/folders/1u6aaL-AXJzUBg2-4beQR4pPkgMt6cdN3?usp=drive_link

Repo Location:
https://github.com/Jordonez123/ZyfiJadesomew1nd/peopleservice (github.com)

Enhance Microservices Implementation:

People Microservice

1, people service

Url: http://34.132.102.197/**

```
Get  /peopleInfo/{username}  //Get By User Name

Post  /peopleInfo                  //Get people list by Role

{
   "role": "salesman"
}

Get  /peopleAccount              //Get User By Id
Request Body like:
{
          "id": 1
}

Get  /peopleAccount/all        //Get All Users

Get  /peopleAccount/usernameAndPassword
                                   //Get By username and password
Request Body like:
{
          "username": "Ava",
          "password": null,
}

Put /peopleAccount          //update User By Id
Request Body like:
{
          "id": 1,
          "username": "Ava",
          "password": null,
          "role": "salesman"
}
```

```
Delete /peopleAccount          //delete User By Id
Request Body like:
{
            "id": 1
}

Post   /peopleAccount          //insert new user
Request Body like:
{
            "username": "Ava",
            "password": null,
            "role": "salesman"
}
```

Response Body Like:

```
{
   "flag": 1101,                    // Flag code
   "data": {                        // data
      "id": 1,
      "username": "faquan",
      "password": null,           // for any request, not return password automatically
      "role": "manager"
   }
}
```

Flag code

```
GET_SUCCESS = 1101;    //for all get request
GET_FAILED = 1100;         //for all get request,  means get nothing
POST_SUCCESS = 1201;
POST_FAILED = 1200;
DELETE_SUCCESS = 1301;
DELETE_FAILED = 1300;
UPDATE_SUCCESS = 1401;
UPDATE_FAILED = 1400;
```

Product MicroService

Get /products //Get product By Id
Request Body like:

Unset


```
{
  "_embedded": {
    "products": [
      {
        "productName": "chip",
        "stockQuantity": 2500,
        "type": "electric products",
        "description": "Im sb",
        "_links": {
          "self": {
            "href": "http://localhost:8080/products/1"
          },
          "product": {
            "href": "http://localhost:8080/products/1"
          }
        }
      },
      {
        "productName": "fries",
        "stockQuantity": 2500,
        "type": "food",
        "description": "Im sb",
        "_links": {
          "self": {
            "href": "http://localhost:8080/products/2"
          },
          "product": {
            "href": "http://localhost:8080/products/2"
          }
        }
      },
```

```
    {
      "productName": "sb",
      "stockQuantity": 2500,
      "type": "person",
      "description": "Im sb",
      "_links": {
        "self": {
          "href": "http://localhost:8080/products/3"
        },
        "product": {
          "href": "http://localhost:8080/products/3"
        }
      }
    },
    {
      "productName": "hub",
      "stockQuantity": 2500,
      "type": "electric products",
      "description": "Im sb",
      "_links": {
        "self": {
          "href": "http://localhost:8080/products/4"
        },
        "product": {
          "href": "http://localhost:8080/products/4"
        }
      }
    }
  ]
},
"_links": {
  "self": {
    "href": "http://localhost:8080/products?page=0&size=10"
  },
  "profile": {
```

```
      "href": "http://localhost:8080/profile/products"
    }
  },
  "page": {
    "size": 10,
    "totalElements": 4,
    "totalPages": 1,
    "number": 0
  }
}
```

Get /products/{id} //Get Product by ID

Request Body like:

Unset

```
{
  "productName": "string",
  "stockQuantity": 0,
  "type": "string",
  "description": "string",
  "_links": {
    "self": {
      "href": "http://localhost:8080/products/2"
    },
    "product": {
      "href": "http://localhost:8080/products/2"
    }
  }
}
```

Put /products/{id} //update Product By Id

Request Body like:

Unset

```
{
  "productName": "string",
  "stockQuantity": 0,
  "type": "string",
  "description": "string",
  "_links": {
    "self": {
      "href": "http://localhost:8080/products/2"
    },
    "product": {
      "href": "http://localhost:8080/products/2"
    }
  }
}
```

Delete /products/{id} //delete Product By Id

Request Body like:

Unset

```
{
  "productName": "sb",
  "stockQuantity": 2500,
  "type": "person",
  "description": "Im sb",
  "_links": {
    "self": {
      "href": "http://localhost:8080/products/3"
    },
```

```
    "product": {
      "href": "http://localhost:8080/products/3"
    }
  }
}
```

Post /products //insert new Product

Request Body like:

Unset

```
{
  "productName": "string",
  "stockQuantity": 0,
  "type": "string",
  "description": "string",
  "_links": {
    "self": {
      "href": "http://localhost:8080/products/6"
    },
    "product": {
      "href": "http://localhost:8080/products/6"
    }
  }
}
```

Flag code

```
GET_SUCCESS = 1101;    //for all get request
GET_FAILED = 1100;     //for all get request,  means get nothing
POST_SUCCESS = 1201;
```

```
POST_FAILED = 1200;
DELETE_SUCCESS = 1301;
DELETE_FAILED = 1300;
UPDATE_SUCCESS = 1401;
UPDATE_FAILED = 1400;
```

Order Microservice

GET
/orders
This method returns all orders
This method Does not have pagination.

ResponseBodyLike:

Unset


```
{
 "flag": 1101,
 "data": [
  {
   "orderId": 1,
   "productId": 1,
   "number": 2,
   "price": 3,
   "clientId": 4,
   "salesmanId": 5,

   "creationTime": "1123-04-11T22:31:53.000+00:00",
   "authorisedPersonId": 2,
   "authorisedTime": "2222-04-11T21:31:53.000+00:00",
   "finishedPersonId": 213,
   "finishedTime": "2023-11-24T07:22:31.000+00:00",
   "authorised": false,
   "finished": true
  },
    {
      "orderId": 1,
      "productId": 1,
      "number": 2,
      "price": 3,
      "clientId": 4,
      "salesmanId": 5,
      "creationTime": "1123-04-11T22:31:53.000+00:00",
      "authorisedPersonId": 2,
      "authorisedTime": "2222-04-11T21:31:53.000+00:00",
      "finishedPersonId": 213,
```

```
      "finishedTime": "2023-11-24T07:22:31.000+00:00",
      "authorised": false,
      "finished": true
    }]
}
```

GET

/orders/{id}

This method returns the order of a specific id

ResponseBodyLike:

```
{
  "flag": 1101,
  "data": {
    "orderId": 7,
    "productId": 1,
    "number": 2,
    "price": 3,
    "clientId": 4,
    "salesmanId": 5,
    "creationTime": "2023-11-24T15:24:01.000+00:00",
    "authorisedPersonId": 99,
    "authorisedTime": "2023-11-24T15:56:43.000+00:00",
    "finishedPersonId": null,
    "finishedTime": null,
    "authorised": true,
    "finished": false
  }
}
```

GET

/orderPG

This method returns all orders

This method Does have pagination.

Takes in two params: page (from0), size (from 1)

LIKE:

http://localhost:8080/ordersPG?page=0&size=1

ResponseBodyLike:

```
{
 "flag":1101,
 "data":{
```

```
  "content": [
   {
    "orderId": 1,
    "productId": 1,
    "number": 2,
    "price": 3,
    "clientId": 4,
    "salesmanId": 5,
    "creationTime": "1123-04-11T22:31:53.000+00:00",
    "authorisedPersonId": 2,
    "authorisedTime": "2222-04-11T21:31:53.000+00:00",
    "finishedPersonId": 213,
    "finishedTime": "2023-11-24T07:22:31.000+00:00",
    "authorised": false,
    "finished": true
   }
  ],
  "pageable": {
   "pageNumber": 0,
   "pageSize": 1,
   "sort": {
    "sorted": false,
    "unsorted": true,
    "empty": true
   },
   "offset": 0,
   "paged": true,
   "unpaged": false
  },
  "totalPages": 8,
  "totalElements": 8,
  "last": false,
  "first": true,
  "size": 1,
  "number": 0,
  "sort": {
   "sorted": false,
   "unsorted": true,
   "empty": true
  },
  "numberOfElements": 1,
  "empty": false
 }
}
```

PUT

/orders/{orderid}

RequestBodyLike:


```
{
 "orderId": 8,
 "productId": 345,
 "number": 345,
 "price": 345,
 "clientId": 0,
 "salesmanId": 0,
 "creationTime": "2023-11-24T16:14:43.582Z",
 "authorisedPersonId": 0,
 "authorisedTime": "2023-11-24T16:14:43.582Z",
 "finishedPersonId": 0,
 "finishedTime": "2023-11-24T16:14:43.582Z",
 "authorised": true,
 "finished": true
}
```


or


```
{
  "orderId": 8,
  "productId": 345,
  "number": 345,
  "price": 345,
  "clientId": 0,
  "salesmanId": 0
}
```


ResponseBodyLike:


```
{
  "flag": 1401,
  "data": {
    "orderId": 8,
    "productId": 1,
    "number": 2,
    "price": 3,
    "clientId": 4,
    "salesmanId": 5,
    "creationTime": "2023-11-24T15:24:01.000+00:00",
    "authorisedPersonId": 99,
    "authorisedTime": "2023-11-24T15:56:43.000+00:00",
    "finishedPersonId": null,
    "finishedTime": null,
    "authorised": true,
    "finished": false
  }
}
```

PUT
/orders/finish/{orderId}/{finishedPersonId}
//FInish an order
ResponseBodyLike:

```
{
 "flag": 1401,
 "data": true
}
```

PUT
/orders/authorize/{orderId}/{authPersonId}
//Authorize an order
ResponseBodyLike:

```
{
 "flag": 1401,
 "data": true
}
```

Response body Like

Unset

```
{
 "flag": 1401,
 "data": {
  "orderId": 8,
  "productId": 345,
  "number": 345,
  "price": 345,
  "clientId": 0,
  "salesmanId": 0,
```

```
    "creationTime": null,
    "authorisedPersonId": null,
    "authorisedTime": null,
    "finishedPersonId": null,
    "finishedTime": null,
    "authorised": false,
    "finished": false
   }
  }
```

POST
/orders
This method create a new Order
Request body Like:

```
{
  "productId": 45142,
  "number": 123124,
  "price": 234234,
  "clientId": 234,
  "salesmanId": 234
}
```

Response Body Like:

```
{
  "flag": 1201,
  "data": {
    "orderId": 10,
    "productId": 45142,
    "number": 123124,
    "price": 234234,
    "clientId": 234,
    "salesmanId": 234,
    "creationTime": "2023-11-24T16:03:23.376+00:00",
    "authorisedPersonId": null,
    "authorisedTime": null,
    "finishedPersonId": null,
    "finishedTime": null,
    "authorised": false,
    "finished": false
  }
}
```

DELETE

/orders/{orderId}

Delete a order.

Response Body Like:


```
{
  "flag": 1201,
  "data": {
    "orderId": 10,
    "productId": 45142,
    "number": 123124,
    "price": 234234,
    "clientId": 234,
    "salesmanId": 234,
    "creationTime": "2023-11-24T16:03:23.376+00:00",
    "authorisedPersonId": null,
    "authorisedTime": null,
    "finishedPersonId": null,
    "finishedTime": null,
    "authorised": false,
    "finished": false
  }
}
```


Flag code

```
GET_SUCCESS = 1101;    //for all get request
GET_FAILED = 1100;        //for all get request,  means get nothing
POST_SUCCESS = 1201;
POST_FAILED = 1200;
DELETE_SUCCESS = 1301;
DELETE_FAILED = 1300;
UPDATE_SUCCESS = 1401;
UPDATE_FAILED = 1400;
```

Events, Notifications, Pub/Sub:

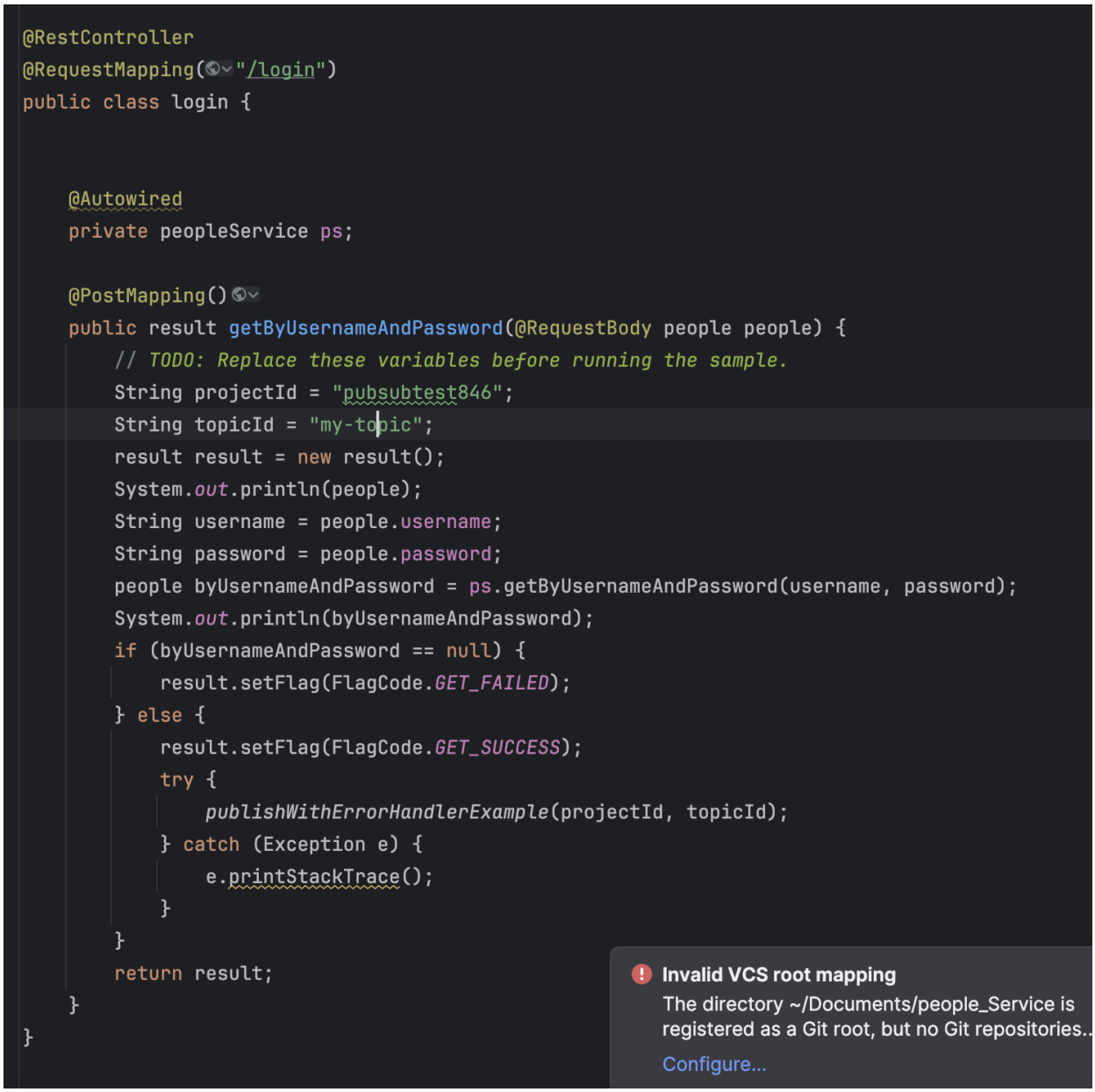
Events:

Take a screenshot of your code publishing events to any queue systems (SNS, Google Pub/Sub).

SetUp: Configuration of the topic and project.

```
public static void main(String... args) throws Exception {
    String projectId = "pubsubtest846";
    String topicId = "my-topic";

    publisherExample(projectId, topicId);
}
```

The code that calls the publication of the event to the topic.



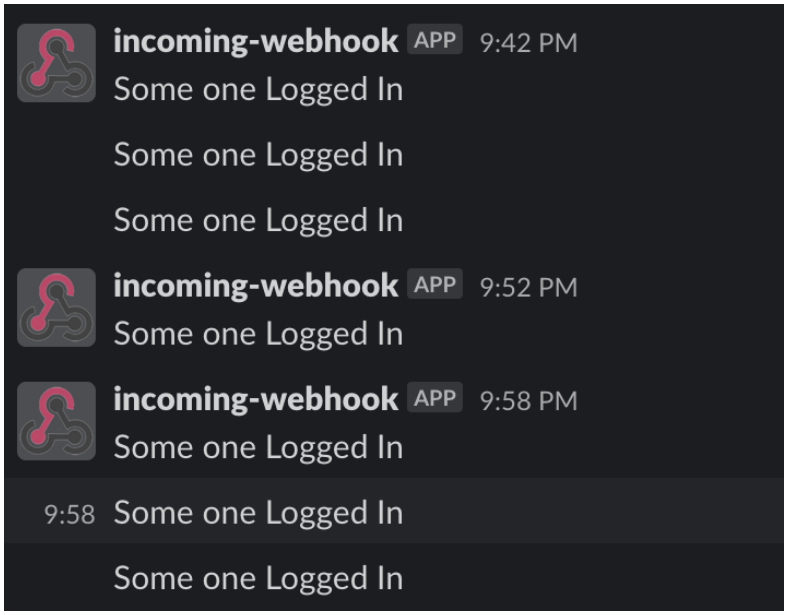
PUBSUB:

The People microservice sends events to the lambda function and the picture below shows the Lambda function handling the published events, and it in turn, sends notifications to slack(By using a webHook).

The lambda function that subscribes to the event in the pipeline.

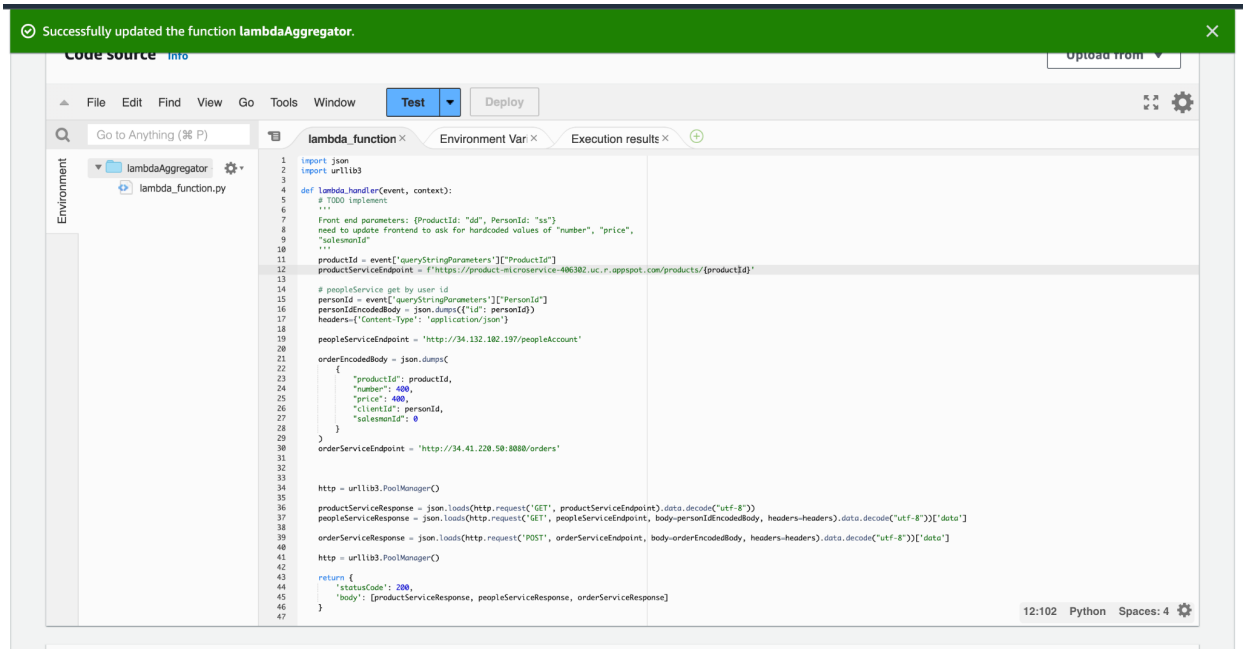
```
project_id = "pubsubtest846"
subscription_id = "my-topic"
# Number of seconds the subscriber should listen for messages
#timeout = 5.0
# Webhook URL provided by Slack
webhook_url = "https://hooks.slack.com/services/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"

# Define the payload as a dictionary
payload = {
    "text": "Some one Logged In"}
```

When there's a login event, a notification is posted.



Composition/Aggregators:

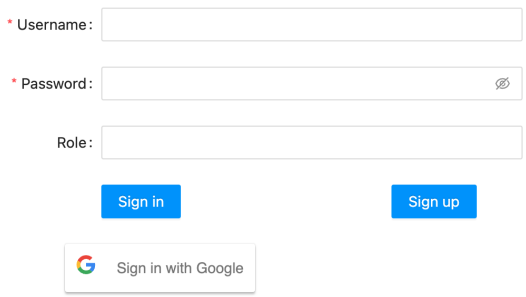
lambdaAggregatorEndpoint:
https://vjvraka4zxpetzucol3y776tim0qwmob.lambda-url.us-east-1.on.aws/
The aggregator gathers the PersonId and ProductId when called, putting them in a procedure that joins the data from the Product and the Person microservice and returns the aggregated data.



Note that the code shows that this Aggregator Aggregates the info from Product and People Microservices, Aggregates the info(in an Async manner) and returns the aggregated information back.

SSO:

Realizing Google Signal Sign On, users can press a button and login/logOut with their google account. We will create an account using their account and GoogleId.



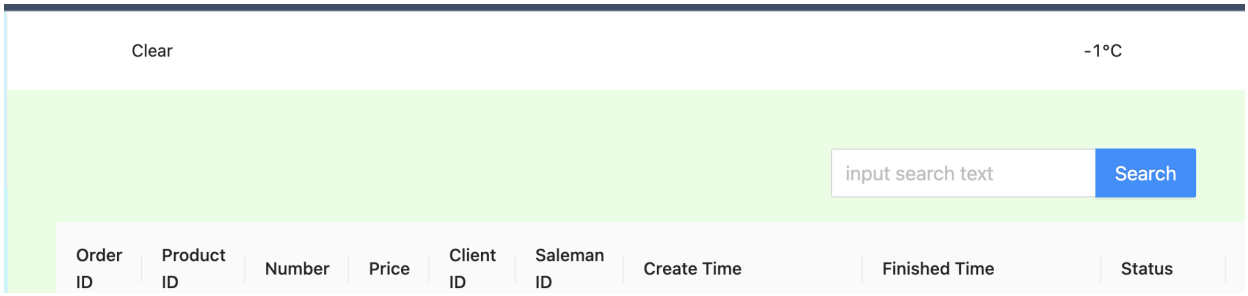
Users can also SignUp or login by {Username,Password,Role} normally.
Deploy a table to display all user's information.

Name	ID	Role
jiayao	1	saleman
jiayaoewqew	2	saleman
jiayaoewqewdwdsamfkofmep	3	buyer
wfq	4	sleeper
lyy	5	salesman
zzx	6	buyer
kelven	7	sales
kelven	8	sales
jiayao1122	9	student
jiayao123sew	13	abc

< 1 2 3 >

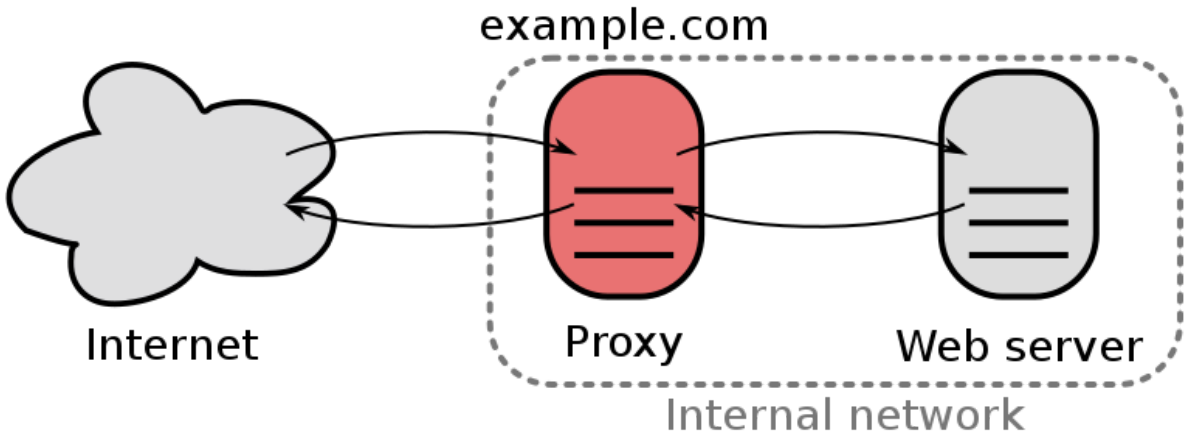
External API:

We use an external API to get real time weather information and display it on our website.



MiddleWare:

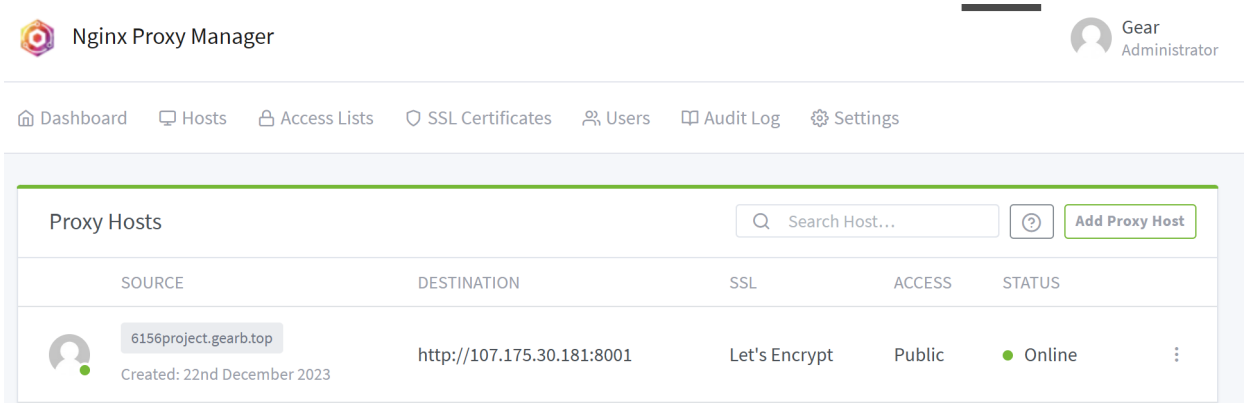
ReverseProxy

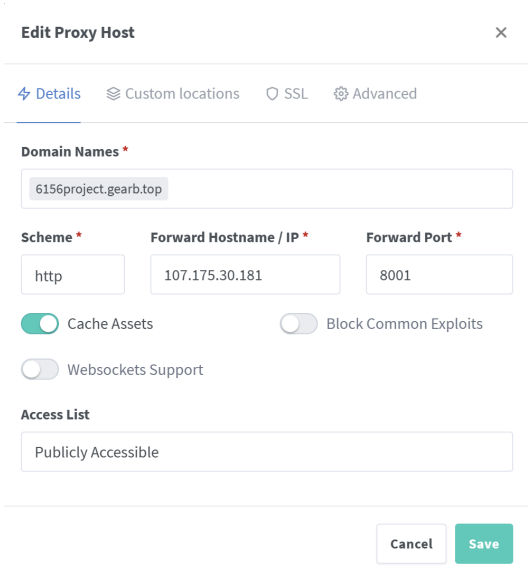


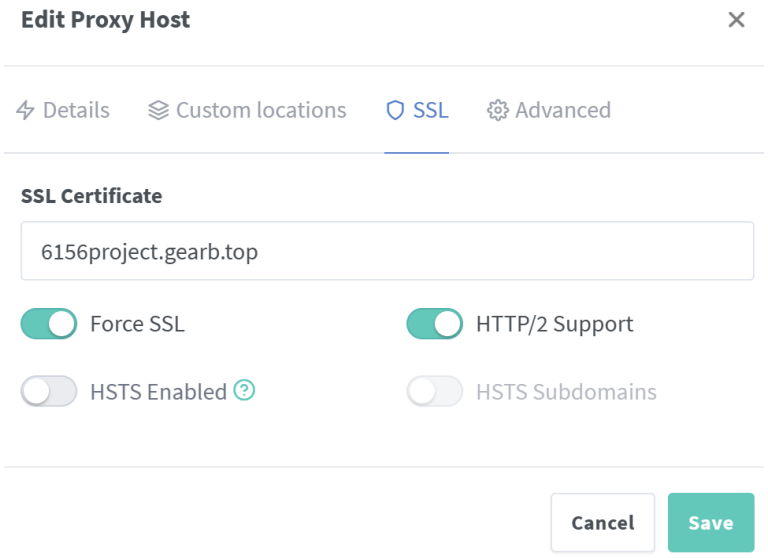
In the context of cybersecurity, a reverse proxy is a crucial component for enhancing security and managing client traffic to a server. When you expose the actual ports (like 80/8080) of your production machine directly to the internet, it becomes vulnerable to attacks and unauthorized access. Here's how a reverse proxy helps mitigate these risks:

- Security: A reverse proxy acts as a shield between the internet and your server. It intercepts all requests coming from the internet and then forwards them to the internal server. This means the actual IP address and port of the server are hidden from the outside world, significantly reducing the surface for attacks.
- Caching: A reverse proxy can cache content, reducing the load on the server and improving response times for clients. This means frequently accessed resources are served faster, enhancing the user experience.
- SSL Termination: It can handle the encryption and decryption of SSL/TLS traffic (HTTPS), offloading this task from the web server. This simplifies the SSL management and improves performance as the web server is freed from this computationally intensive task.
- Content Filtering/URL Rewriting: A reverse proxy can modify the requests and responses on-the-fly. It can filter out harmful content and rewrite URLs to ensure that the internal structure of the application is not exposed.

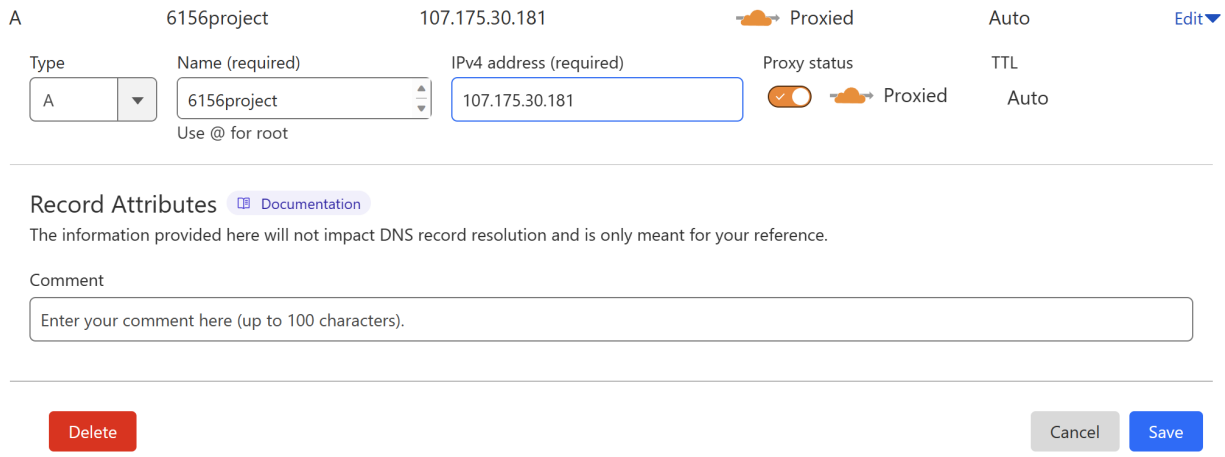
In our project, the host server of the dynamic web page with NodeJS is reverse proxied using NginxProxyManager, with the Self-signed SSL (for demonstration purposes) as HTTPS credentials. The host machine of the NginxProxyManager runs the software in docker, providing reliable service.


Nginx Proxy Manager
Gear
Administrator
Dashboard
Hosts
Access Lists
SSL Certificates
Users
Audit Log
Settings
Proxy Hosts
Search Host...
Add Proxy Host
SOURCE
DESTINATION
SSL
ACCESS
STATUS
6156project.gearb.top
Created: 22nd December 2023
http://107.175.30.181:8001
Let's Encrypt
Public
Online


Edit Proxy Host
Details
Custom locations
SSL
Advanced
Domain Names *
6156project.gearb.top
Scheme *
Forward Hostname / IP *
Forward Port *
http
107.175.30.181
8001
Cache Assets
Block Common Exploits
Websockets Support
Access List
Publicly Accessible
Cancel
Save


Edit Proxy Host
Details
Custom locations
SSL
Advanced
SSL Certificate
6156project.gearb.top
Force SSL
HTTP/2 Support
HSTS Enabled
HSTS Subdomains
Cancel
Save

On top of that, we deployed CloudFlare CDN on the reverse proxied url to ensure maximum safety and access speeds.



API GateWay

The API for the Zyfijade project 1.0.0 OAS 2.0

[Base URL: /]

API to interact with the Zyfijade microservices

Schemes

HTTPS

PEOPLE

GET /peopleInfo/{username} get by username

POST /peopleInfo get people list by role

PUT /peopleAccount update user by Id

DELETE /peopleAccount delete user by Id

POST /peopleAccount insert new user

POST /peopleAccount/id get user by id

GET /peopleAccount/all get all users

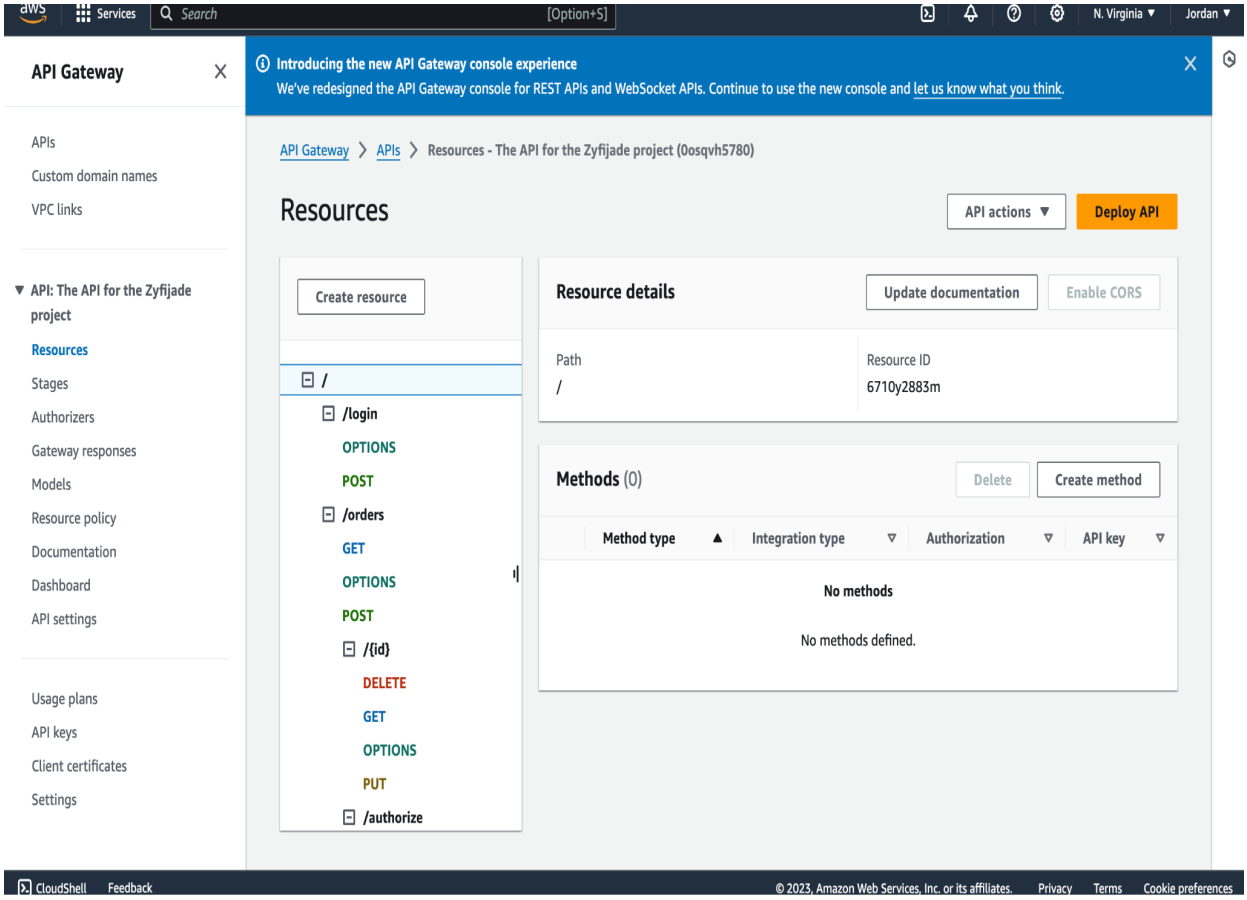
POST /login get by username and password

PRODUCTS

GET	/products	get all products
POST	/products	insert new product
GET	/products/{id}	get product by id
DELETE	/products/{id}	delete product by Id
PUT	/products/{id}	update product by Id

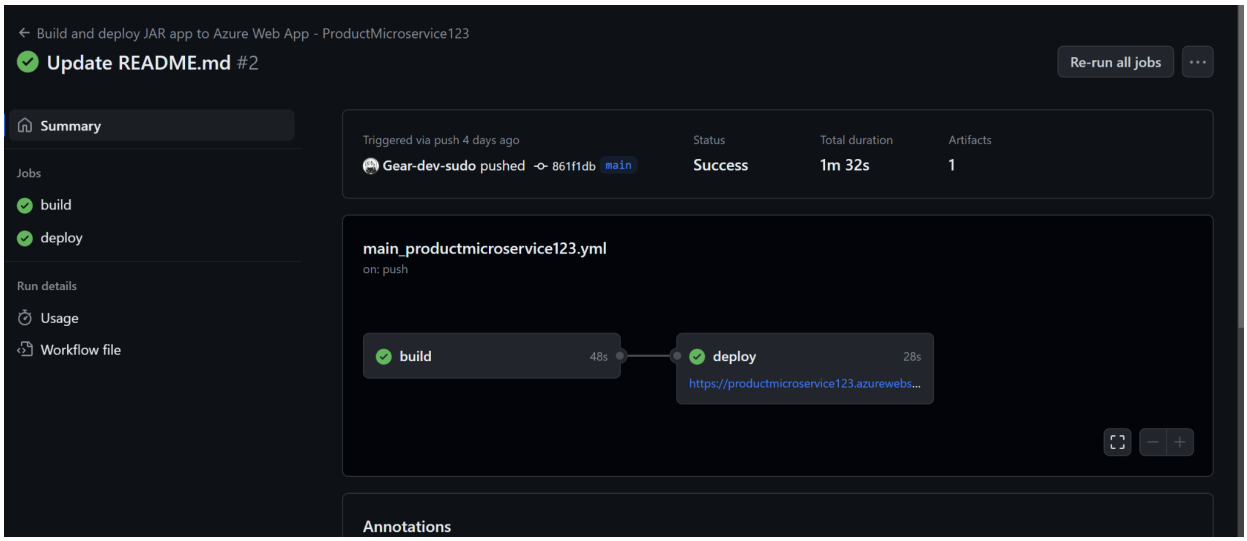
ORDERS

GET	/orders	returns all orders. Does not have pagination
POST	/orders	
GET	/orders/{id}	
PUT	/orders/{id}	
DELETE	/orders/{id}	
GET	/ordersPG	
PUT	/orders/finish/{orderId}/{finishedPersonId}	
PUT	/orders/authorize/{orderId}/{authPersonId}	



CI/CD

CICD is an important modern-day DevOps Practice. In terms of our specific project, the modifications are made to the production server(compiled, tested, deployed) whenever there's a github **PUSH EVENT**.



Specifically:

1. The Github Action triggers when a push is made.
2. Configures java-17 and Maven
3. Installs Dependencies, compiles the source code. ```mvn clean install ```
4. Deploy to Azure SpringApp.

As a result, when a dev pushed the code, the changes are made to the production environment automatically if it works, if it doesn't, theres notifications sent via email to the devs, and the changes are not deployed.

Infrastructure as code

Infrastructure as code or IaC, is also an integral part of the modern-day DevOps practices, since the control of deployment and configuration or Access Control of the resources on the cloud can be modded by using code only, the deployment process can be streamlined and simplified. In our case, we use terraform as a tool to implement this.

Specifically:

1. Configure the credentials of the GCP, import to terraform.
2. Specify the resources we would like to use, i.e. ports and locations.
3. Use terraform cli to validate the configuration file.
4. Call terraform deploy
5. When we need to shut the services down, call terraform to destroy this resource.

```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "4.51.0"
    }
  }
}

provider "google" {
  credentials = file("ordermicroservice-72b3ecbe463a.json")

  project = "ordermicroservice"
  region  = "us-central1"
  zone    = "us-central1-c"
}
resource "google_cloud_run_service_iam_member" "allow_unauthenticated" {
  service = google_cloud_run_service.default.name
  location = google_cloud_run_service.default.location
  role = "roles/run.invoker"
  member = "allUsers"
}
resource "google_cloud_run_service" "default" {
  name     = "cloudrun-srv"
  location = "us-central1"

  template {
    spec {
      containers {
        image = "ianli233/my-spring-boot-appv2:latest"

        ports {
          container_port = 8080
        }
      }
    }
  }

  traffic {
    percent         = 100
    latest_revision = true
  }
}

```

GraphQL

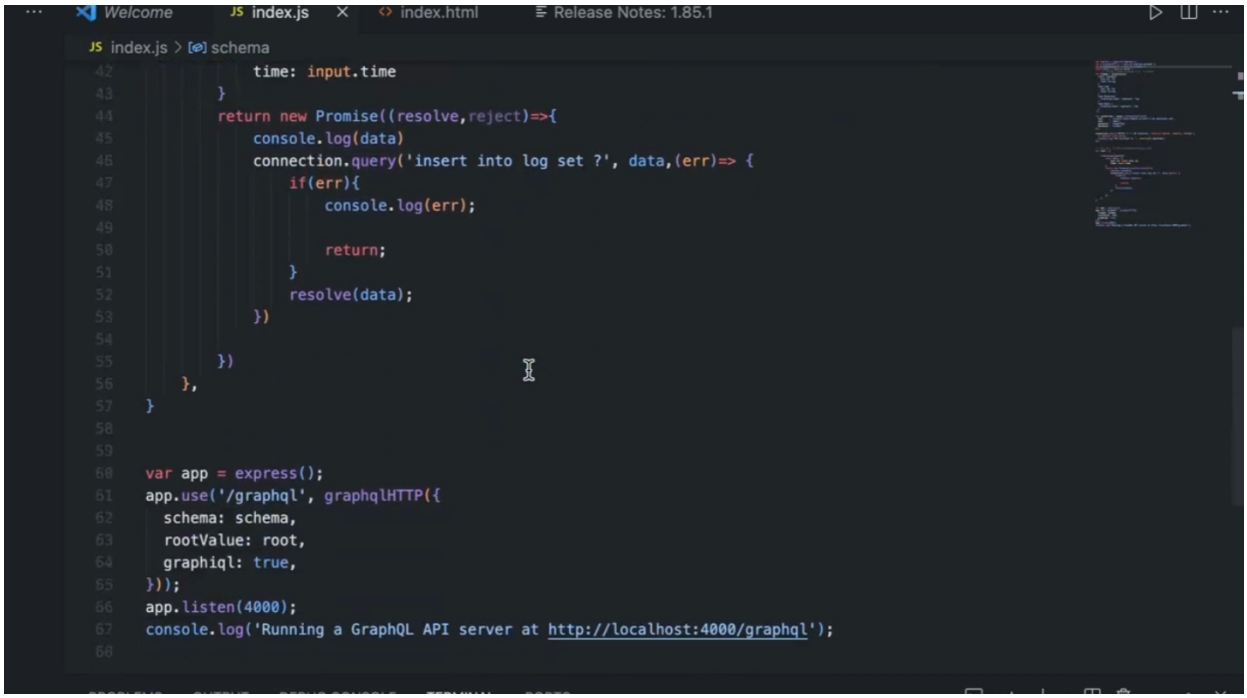
GraphQL is both a query language for APIs and a runtime for your data queries. GraphQL provides a complete set of easy-to-understand descriptions of the data in your API, allowing the client to get exactly the data it needs without any redundancy, making it easier for the API to evolve over time, and for building powerful developer tools.

That is how we implement this technology.

1, Scene Description

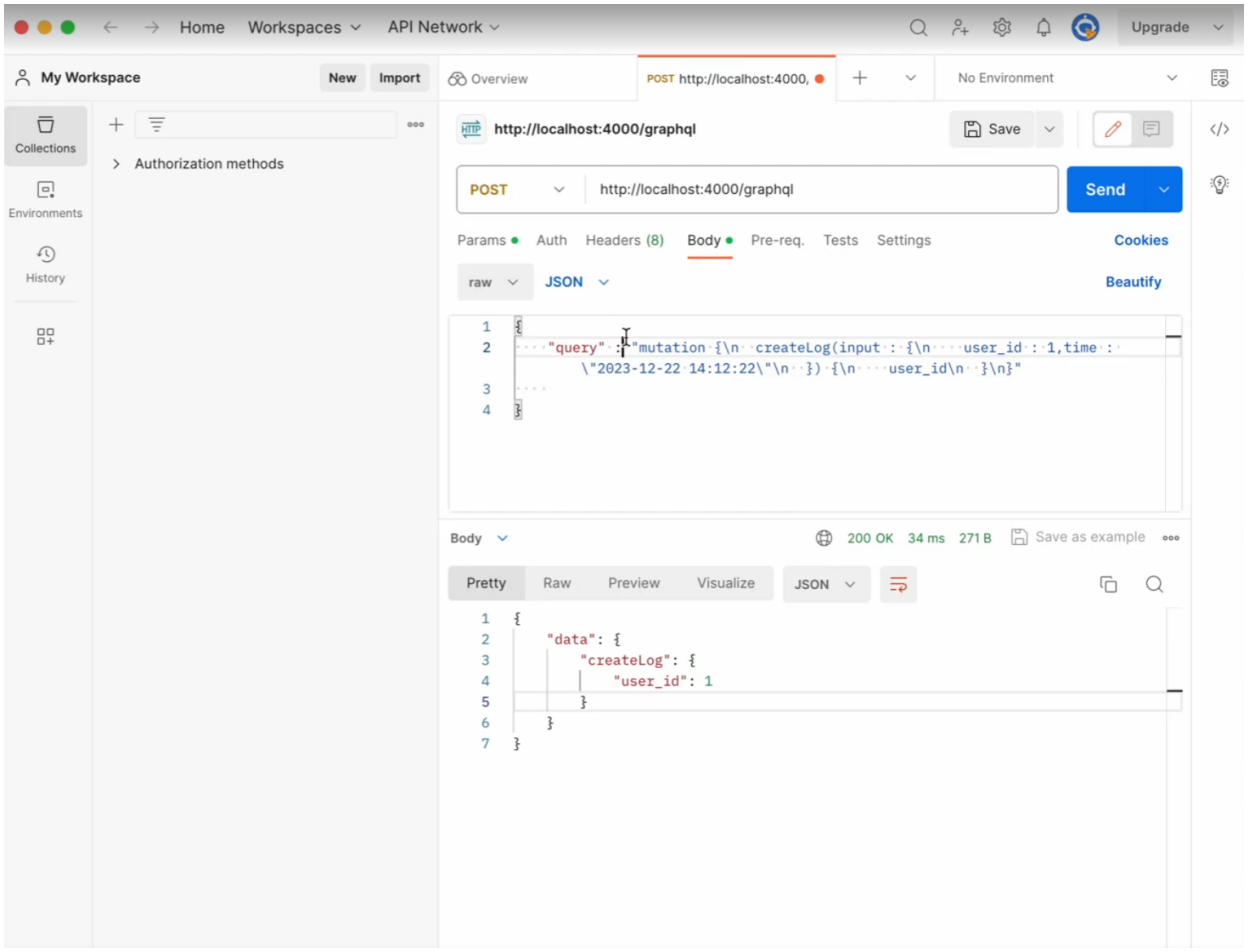
We would like to make a backup of user logins. The traditional approach requires the support of a back-end interface to manipulate the database. However, login records are more flexible and do not require much join operations on multiple tables. graphql is a good fit.

2, Service Interface Construction

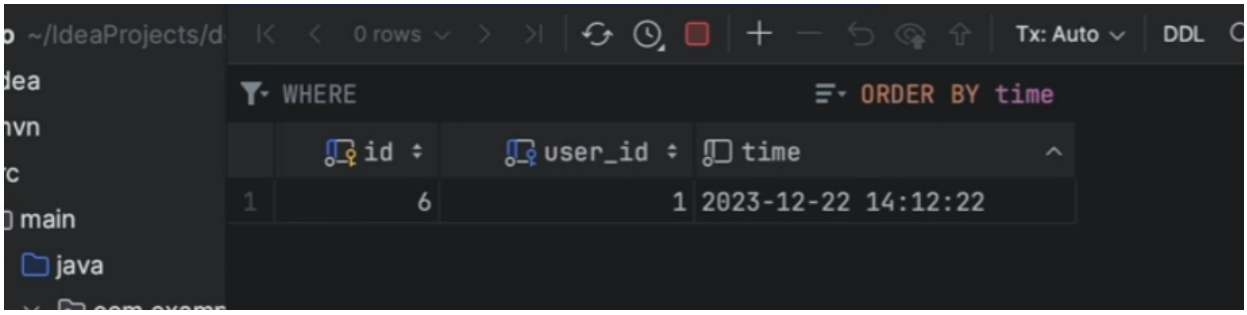


3, Test

Use postman post pretend a user login.



Result:
It's worth clarifying that user_id is a foreign key, corresponding to the id in the people table. The time is the real time at the time of login.



This way, the front-end is free to use JavaScript to modify anything in the database.

Webserver/Webpage IP:http://107.175.30.181:8001

- The deployment with CDN and the reverse proxy url: https://project2.gearb.top/