

Project Report for the Cloud Computing course project

Groups members [Muscas, Marco; Amininodoushan, Negin]

Project title: [Product authentication system]

Deviations from the Project proposal

- 1 Using Azure instead of Amazon AWS. We had no choice due to the unavailability of infrastructure to deploy a Kubernetes cluster; also, the only available zone for student subscription accounts would have been in the eastern US. Azure allowed for solving the availability problem and better positioning.
- 2 Since we wanted to put more time and effort on the backend and development of our web application, which is one of the major parts of cloud computing course, we did not implement a frontend for our web application.
- 3 The previous version of the proposal referred to a system to check the authenticity of a product via the serial number. This could be implemented by answering the question “is this serial number in our database or not?”. This seemed to simplify the project as a whole, so we decided to add support for registration of users and administrators, with their own reserved operations.
It allowed us to explore a bit more in depth the Flask library and the MongoEngine.

In this new version the users will be able to create an account through our API, register products and see what and when they have registered so far.

Description of the problem addressed

Here you should provide the description of the problem addressed (it comes from the project proposal, it could be more detailed)

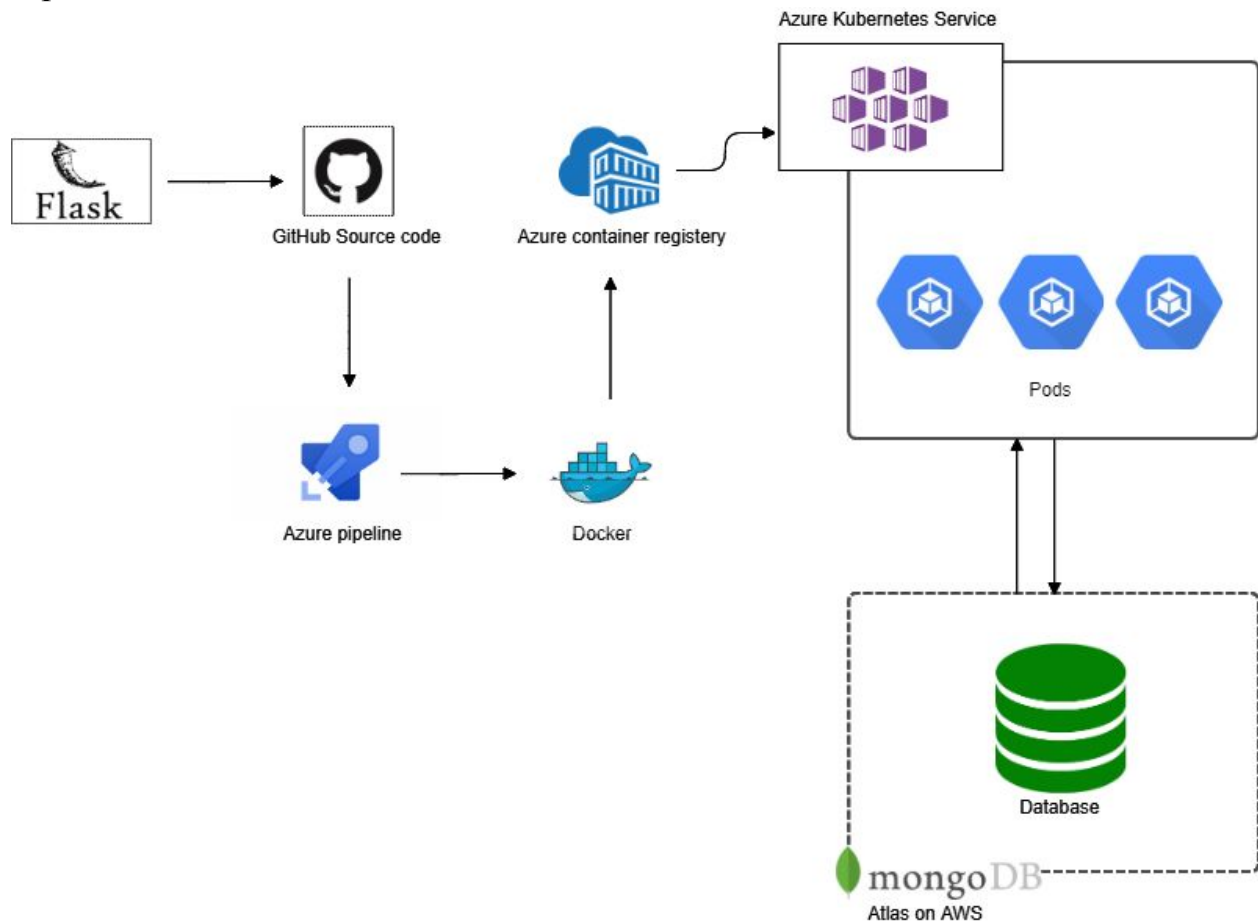
Being assured of the authenticity of a product is one of the major concerns for consumers especially when they are looking for online shopping. Our goal is to provide a verification originality web service to help end users track the authenticity of their purchased product by using its serial number.

Our web application, not only allows users to track the originality of their purchase but also allows the possible admins to interact with it and visualize or extend the dataset.

Design of the solution

Here you should describe and motivate the design of your solution

In the following picture, you can see the architecture of our design. We will also explain the beneficial features of each framework.



Flask: For developing the backend of our web application, we used the Flask framework. Flask is a lightweight web application framework, which is designed to make getting started quick and easy, with the ability to scale up to complex applications.

MongoDB Atlas: For storing our data, we used the AWS MongoDB Atlas, which is a management service for MongoDB and provides all features of MongoDB and takes the responsibility of hosting, patching, managing and securing your MongoDB cluster.

Some beneficial features of MongoDB consists of:

- Built-in replication for availability and tolerating data center failure
- Security features for accessing data
- Choice of cloud providers, regions and billing options
- Fine-grained monitoring and ease of scaling

Docker: Docker provides the ability to package and run an application in a loosely isolated environment called a container. By using containerization, we can make sure that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code. It means that we can focus on writing code without worrying about the system that it will finally be running on.

We created a docker image locally with the Docker Engine on our machine and set up a Docker Image repository on Azure Container Registry.

Azure Kubernetes cluster: At the end, we will deploy our application in Azure Kubernetes Cluster. Azure Kubernetes Service (AKS) is Microsoft Azure's managed Kubernetes solution. It competes with both Google Kubernetes Engine (GKE) and Amazon Elastic Kubernetes Service (EKS). Some beneficial features of AKS consist of:

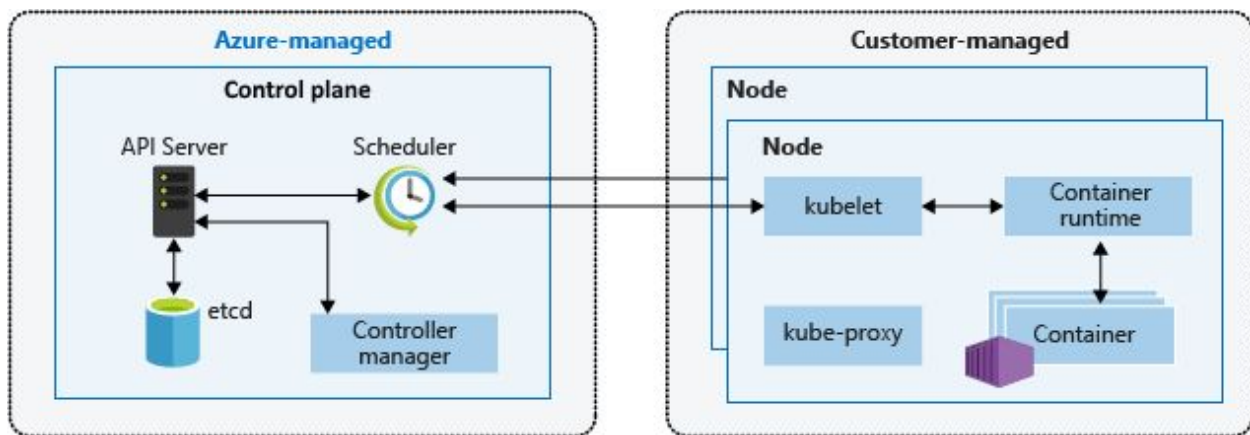
- Elastic scalability
- Enterprise-grade security
- Pay for compute only - no minimum monthly charge
- Agile project management support
- Support for automatic creation of a CD pipeline.

Kubernetes cluster consists of two components: control plane nodes, which provide core Kubernetes services and orchestration of application workloads and the nodes

that run the application workloads. The control plane includes the following components:

- kube-apiserver: provides interaction for management tools
- etcd: maintains the state of the Kubernetes configuration and cluster
- kube-scheduler : schedules the workload on nodes
- kube-controller-manager: supervises smaller controllers

Below, you can see the architecture of an Azure Kubernetes cluster



Description of the Implementation of the solution

Here you should provide a detailed description and motivation for the design of your solutions

1) Database part:

For the database part we will be using MongoDB Atlas which is a global cloud database for modern applications, is distributed and secured by default and is available as a fully managed service on AWS, Azure and Google Cloud.

The screenshot shows the MongoDB Atlas console interface. At the top, the 'Cloud Provider & Region' section is active, displaying 'AWS, Frankfurt (eu-central-1)' as the selected option. Below this, there are three tabs: 'aws', 'Google Cloud', and 'Azure'. The 'aws' tab is selected and highlighted with a green border. Underneath, there's a 'Recommended region' section with three columns: 'NORTH AMERICA', 'EUROPE', and 'ASIA'. In the 'EUROPE' column, 'Frankfurt (eu-central-1)' is selected and highlighted with a green border. Below this, there's an 'AUSTRALIA' section with 'Sydney (ap-southeast-2)' listed. The 'Cluster Tier' section shows 'M0 Sandbox (Shared RAM, 512 MB Storage)' as the selected tier, with 'Encrypted' status. Below this, there's a 'Connect to MyfirstCluster' section with three steps: 'Setup connection security', 'Choose a connection method', and 'Connect'. The first step is selected. Under '1 Select your driver and version', there are two dropdown menus: 'DRIVER' set to 'Python' and 'VERSION' set to '3.6 or later'. Below this, there's a second step '2 Add your connection string into your application code'. Under this step, there's a checkbox 'Include full driver code example' which is unchecked. Below the checkbox, there's a text area containing the connection string: 'mongodb+srv://Admin1:<password>@myfirstcluster.16yqg.mongodb.net/<dbname>'. To the right of the text area is a 'Copy' button.

For creating a cluster on MongoDB Atlas, we will use aws as the cloud provider and choose the nearest region. We will use M0 cluster tier, which is free and gives us 512 MB storage. By default, MongoDB creates 3 replicas; 1 Primary and 2 Secondary.

In order to connect our flask application to the cluster we need to get the URL to the cluster. After setting the users and whitelisting IP addresses, we should choose the driver to Python version 3.6 or later and then a URL link will be generated which we will use later in our flask code.

2) Creating RESTful API with Python & Flask

In this part, we will use Python and Flask to create our API. In order to connect our database to the web service we will use MongoEngine, which is a document-object mapper for working with MongoDB from Python.

- First we need to import the required libraries:

```
from flask import Flask, jsonify, make_response, request
from flask_mongoengine import MongoEngine
from werkzeug.security import generate_password_hash, check_password_hash
from functools import wraps
import jwt
from datetime import datetime, timedelta
import json
from flask import render_template
import uuid
```

- MongoEngine assumes that the MongoDB instance is running on localhost on port 27017 by default; to change it, we should provide the URL to our cluster that we got in the previous step together with the database name and the password. We put all this information in a configuration file as below.

```
SECRET_KEY="thisiscool"
TESTING=False
DEBUG=True
MONGODB_HOST="mongodb+srv://clc_db_admin:clc2020_nice_password@clusterclc.sfbwf.mongodb.net/clc2020_sys_db?retryWrites=true&w=majority"
```

Let's start our Flask application

- Now we should create the schema and define the fields that our collection will have. We will create a class *User*, which will inherit from *db.document* and have six fields: *id*, *public_id* (an ID is assigned to each user randomly), *email*, *password*, *registered_serial_num* and *is_admin* (If a user is the admin of web application this variable is assigned to True). we also create a function *to_json* to convert the document to json whenever is needed.

```

class User(db.Document):
    _id = db.IntField(primary_key=True)
    public_id = db.StringField(unique=True)
    email = db.StringField(unique=True)
    password = db.StringField()
    registered_serial_numbers = db.ListField(default=[])
    if_admin = db.BooleanField(default=False)

    def to_json(self):
        raw_json = {"public_id": self.public_id,
                    "email": self.email,
                    "password": self.password,
                    "registered_serial_numbers": self.registered_serial_numbers,
                    "if_admin": self.if_admin}
        return raw_json

```

- Another class is Products, which contains product_id, name and image_url fields:

```

class Product(db.Document):
    _id = db.IntField(primary_key=True)
    product_id = db.StringField(required=True, unique=True)
    name = db.StringField(required=True, max_length=100)
    image_url = db.StringField(required=False)

    def to_json(self):
        raw_json = {'product_id': self.product_id,
                    'name': self.name,
                    'image_url': self.image_url}
        return raw_json

```

- The last class is *serialNumber*, which contains *value* (the price of the product), *registration_date*, *registration_user* (the ID of the registering user) and *produc_id* of that product.

```

class SerialNumber(db.Document):
    _id = db.StringField(primary_key=True)
    value = db.StringField(required=True, unique=False)
    registration_date = db.StringField()
    registration_user = db.StringField() # ID of the registering user
    product_id = db.StringField(required=True)

    def to_json(self):
        raw_json = {'value': self.value,
                    'registration_date': self.registration_date,
                    'registration_user': self.registration_user,
                    'product_id': self.product_id}
        return raw_json

```

- User registration:

Registration of a user is pretty basic. The API expects to receive a JSON with new user data and won't directly check for user availability, the email (used as a unique attribute) will make it perform such an operation implicitly.

There is also a particular condition in the code: it is also possible to register new admin accounts. In addition to the email and password required it will check for presence of an admin key sent in the request body, which if corresponds to the one in the configuration file of the application, will result in the account having admin privileges.

```
@app.route('/api/user/register', methods=['POST'])
def register():
    request_content = request.get_json()
    return jsonify({'content': str(request_content)})
    try:
        email = request_content['email']
        password = request_content['password']

        if 'admin_key' in request_content.keys():
            admin_key = request_content['admin_key']

            if admin_key == app.config['ADMIN_REGISTRATION_KEY']:
                User.objects.insert(User(public_id=str(uuid.uuid4()), email=email, password=password,
                                         registered_serial_numbers=[], if_admin=True))

            else:
                User.objects.insert(
                    User(public_id=str(uuid.uuid4()), email=email, password=password, registered_serial_numbers=[]))

    except Exception as e:
        return jsonify({'code': 400, 'message': str(e)})

    return make_response('User registered successfully.', 201)
```

- Now we create a login route that will allow us to take email and password from users and provide authentication.

If the user does not provide email or password or the credentials are not matching the ones existing in the *user* database, error 401 will be returned. Otherwise the user will get a jwt token that would expire after a defined amount of time and with that token the user can be constantly authenticated without using an email and password every single time.


```

@app.route('/api/user/login/', methods = ['POST'])
def login():

    auth = request.get_json()['authorization']

    if not auth:
        return make_response('Auth missing', 401, {'WWW-Authenticate': 'Basic realm="Login required!"'})

    if not auth or ('email' not in auth.keys()) or ('password' not in auth.keys()):
        return make_response('Could not verify', 401, {'WWW-Authenticate': 'Basic realm="Login required!"'})

    email = auth['email']
    password = auth['password']

    user = User.objects(email=email).first()

    if not user:
        return make_response('Could not verify1', 401, {'WWW-Authenticate': 'Basic realm="Login required!"'})

    if password == user.password:
        user_expire_session = datetime.utcnow() + timedelta(minutes = 4)

        print('Login successful')
        token = jwt.encode(
            {'public_id': user.public_id, 'exp': user_expire_session}, app.config['SECRET_KEY'])

        token = token.decode('UTF-8')
        return jsonify({'token': token})

    return make_response('Could not verify2', 401, {'WWW-Authenticate': 'Basic realm="Login required!"'})

```

With the token, we need to authenticate the subsequent requests to other routes that require login. In other words, only the login page will work with HTTP basic authentication and other login-required routes work with tokens.

- Now we need to create two decorator functions that work with the token, one for admins and another for all users.

```

# ----- Decorators -----
# Create a decorator to apply for the endpoints that require admin authentication
def admin_token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = None

        if 'x-access-token' in request.headers:
            token = request.headers['x-access-token']

        if not token:
            return jsonify({'message': 'Token is missing!'}), 401

        try:
            data = jwt.decode(token, app.config['SECRET_KEY'])
            current_user = User.objects(public_id=data['public_id']).first()
        except Exception as e:
            return jsonify({'message': 'Token is invalid!', 'exception': str(e)}), 401

        if current_user.to_json()['if_admin']:
            return f(*args, **kwargs)
        else:
            return jsonify({'message': "You can't access this page"}), 401

    return decorated

```

```

# Create a decorator to apply for the endpoints that require user authentication
def login_token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = None

        if 'x-access-token' in request.headers:
            token = request.headers['x-access-token']

        if not token:
            return jsonify({'message': 'Token is missing!'}), 401

        try:
            data = jwt.decode(token, app.config['SECRET_KEY'])
            current_user = User.objects(public_id = data['public_id']).first()
        except:
            return jsonify({'message': 'Token is invalid!'}), 401

        return f(*args, **kwargs)

    return decorated

```

- The following endpoints will allow the admin to get information or make some changes to the database. The *admin_token_requires* tag shows that only users that are logged in as admins are allowed to reach these endpoints.

The functionality of endpoints are getting all users, getting a user with a specific public id, delete all users and delete a user by Email.

```

@app.route('/api/user/', methods=['GET'])
@admin_token_required
def get_all_users():
    return jsonify({'users': User.objects})

@app.route('/api/user/<public_id>', methods=['GET'])
@admin_token_required
def get_user_by_id(public_id):
    return jsonify({'user': User.objects(public_id=public_id).first()})

@app.route('/api/user/delete_all', methods=['GET'])
@admin_token_required
def delete_all_users():
    User.drop_collection()
    return jsonify({'users': User.objects})

@app.route('/api/user/delete/', methods=['POST'])
@admin_token_required
def delete_user():
    try:
        content = request.get_json()
        email = content['email']

        User.objects(email=email).delete()

    except Exception as e:
        return jsonify({'code': 400, 'message': str(e)})

    return jsonify({'code': 200, 'message': 'ok'})

```

- The product API will allow the Admin to visualize, add or delete products to the products database. Below you can see the snippets of codes for this API.

```
@app.route('/api/product/', methods = ['GET'])
@admin_token_required
def get_all_products():
    return jsonify(Product.objects)

@app.route('/api/product/<product_id>', methods = ['GET'])
@login_token_required
def get_product_by_id(product_id):
    product_object = Product.objects(product_id = product_id).first()
    if product_object:
        return make_response(jsonify(product_object.to_json()), 200)

    return jsonify({'code':404, 'message':"Product not found."})

@app.route('/api/product/add', methods = ['POST'])
@admin_token_required
def insert_new_product():
    try:
        content = request.get_json()
        product_id = content['product_id']
        name = content['name']
        image_url = content['image_url']
        Product.objects.insert(Product(product_id = product_id, name = name, image_url = image_url))

        return jsonify({'code':200, 'message':'ok'})
    except Exception as e:
        return jsonify({'code':400, 'message':str(e)})
```

```
@app.route('/api/product/delete/')
@admin_token_required
def delete_all_products():
    Product.drop_collection()
    return jsonify({'products':Product.objects})

@app.route('/api/product/delete/<product_id>')
@admin_token_required
def delete_product(product_id):
    try:
        Product.objects(product_id = product_id).delete()

    except Exception as e:
        return jsonify({'code':400, 'message':str(e)})

    return jsonify({'code':200})
```

- Finally, the most important endpoint of our web application is where the user can check the serial number of the product. There are also other endpoints for admins to visualize, delete or bulk insert data into the serial number database.

```
@app.route('/api/serial_number/', methods = ['GET'])
@admin_token_required
def get_all_serial_numbers():
    return jsonify({'serial_numbers': SerialNumber.objects[:1000]})

@app.route('/api/serial_number/delete', methods = ['GET'])
@admin_token_required
def delete_all_serial_numbers():
    SerialNumber.drop_collection()
    return jsonify({'serial_numbers': SerialNumber.objects})

@app.route('/api/serial_number/<sn>', methods=["GET"])
@login_token_required
def search_sn(sn):
    serial_object = SerialNumber.objects(value = sn).first()
    if serial_object:
        return make_response(jsonify(serial_object.to_json()), 200)
    else:
        return make_response('', 404)
```

```
@app.route('/api/serial_number/bulk/add', methods = ['POST'])
@admin_token_required
def bulk_add_serial_number():
    try:
        list_of_sn = request.get_json()
        product_list = []

        for content in list_of_sn:

            value = content['value']
            registration_date = content['registration_date']
            registration_user = content['registration_user']
            product_id = content['product_id']

            new_sn = SerialNumber(value = value, registration_date = registration_date, registration_user = registration_user, product_id = product_id)
            product_list.append(new_sn)

        SerialNumber.objects.insert(product_list)

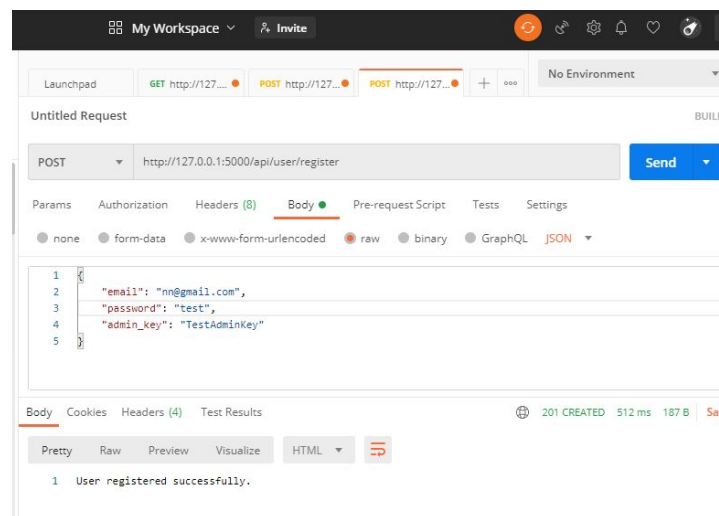
    except Exception as e:
        print(str(e))
        return make_response(str(e), 400)

    return jsonify({'code': 200, 'message': 'ok'})
```

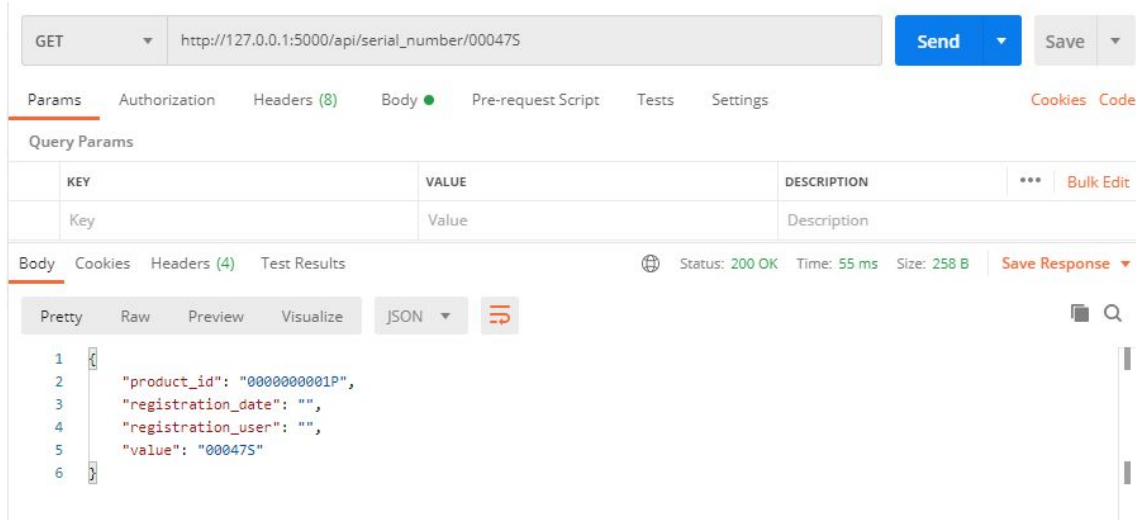
Functional tests of our endpoints:

For testing the functionality of our APIs we will check them by using Postman. We checked all the endpoints to make sure they work properly but in order to not make this report so long, we will put some examples of them below.

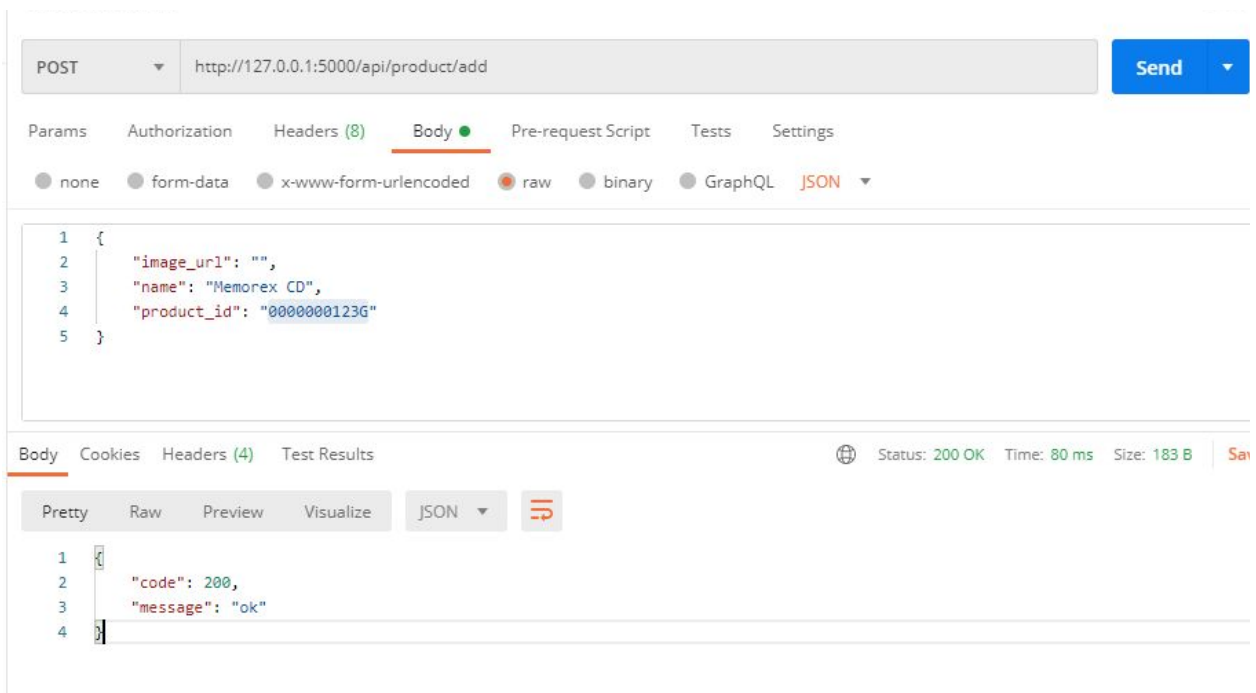
- **Registration:** Here we can see the test for the registration part. The user should send a POST request and provide email and password in JSON format to "api/user/register" path. If the user wants to be registered as Admin, should also provide a key-value pair, the key is "admin_key" and the value is provided in our configuration file, so if a user wants to be registered as admin should be aware of Admin_key. If the JSON inputs are correct, we will see 201 code and "User registered successfully" message.



- **Login:** Below we can see the testing for the login part. Email and password should be sent to "/api/user/login/" route; if the credentials are correct, a token will be returned. To test how the token works, we should add an x-axis-token header and put the token as the value of it. We will use the "api/product/" route which is an admin required route to test if the token is working. If the token is valid we can see the list of all products.



- **Adding a product:** Here we can test adding a product to our product database using "api/product/add" endpoint. The user should provide key-value pairs of *image_url*, *name* and *product_id*. The code 200, shows that the product has been stored in the database.



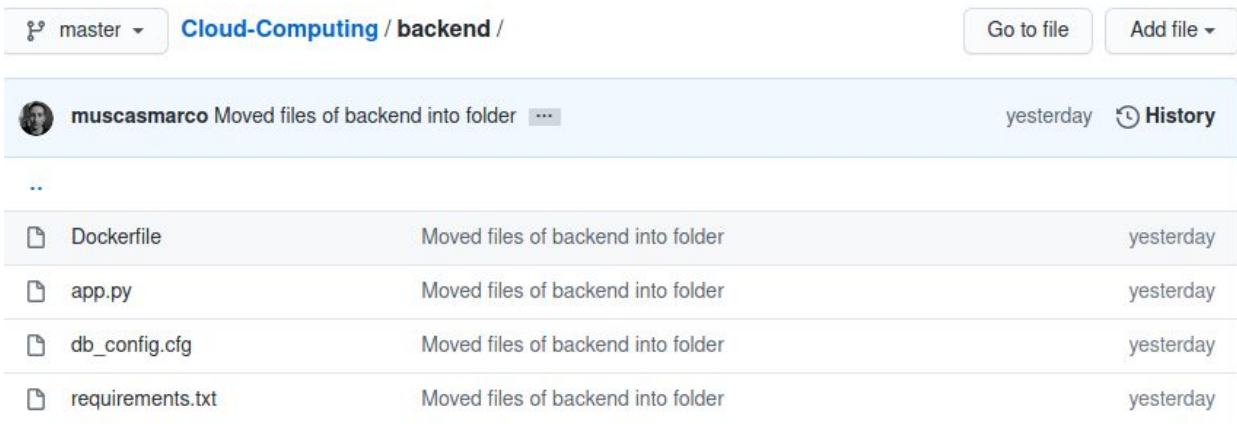
Description of the deployment of your solution

Here you should provide a detailed description of your deployment and/or of the environment you set up for the testing/validation. Motivate your choices.

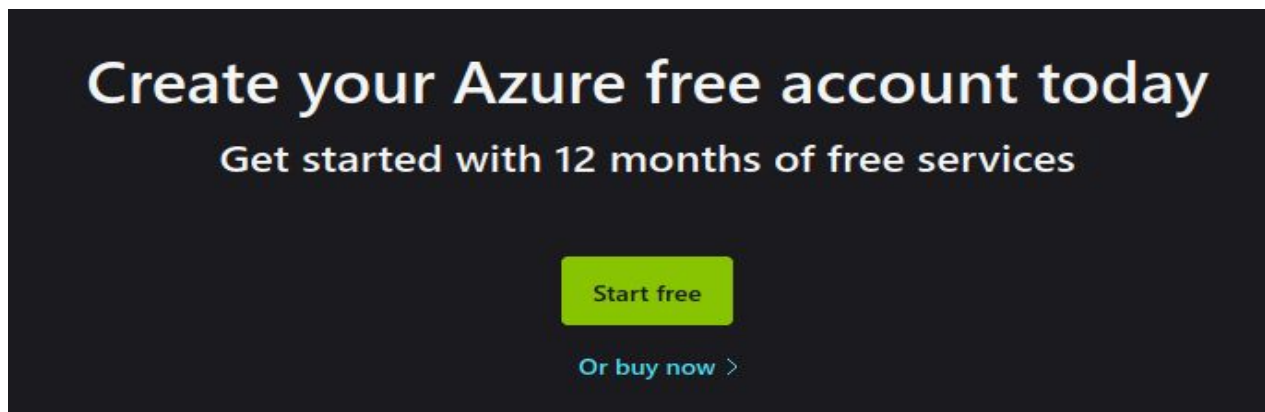
As we saw in the design part, we will use Azure Kubernetes Service (AKS) to deploy our web application. AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure. Our goal in this section is deploying our API on AKS so it runs on an automatically orchestrated cluster of nodes; so let's get started!

1) Preliminary steps:

- ✓ Creating GitHub repository: Since we want to use automated code deployment from GitHub we need to create a GitHub repository and upload app.py, requirements.txt and DockerFile on it.

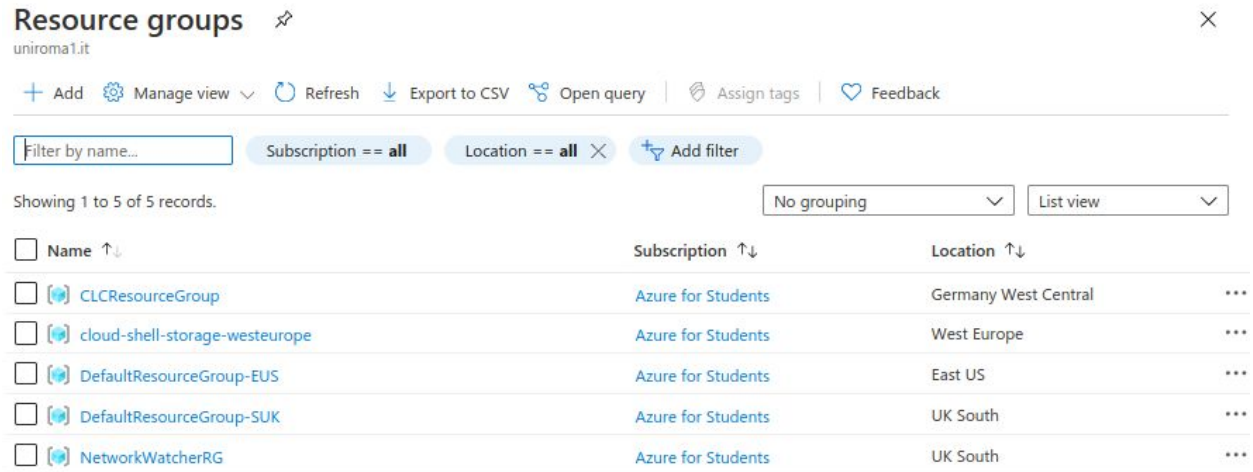


- ✓ Create and activate an Azure account with student subscription



2) Create a resource group on Azure:

Resource group is a container that holds related resources for an Azure solution. As we can see in the below picture, our resource group is CLCResourceGroup and the region is Germany West Central.



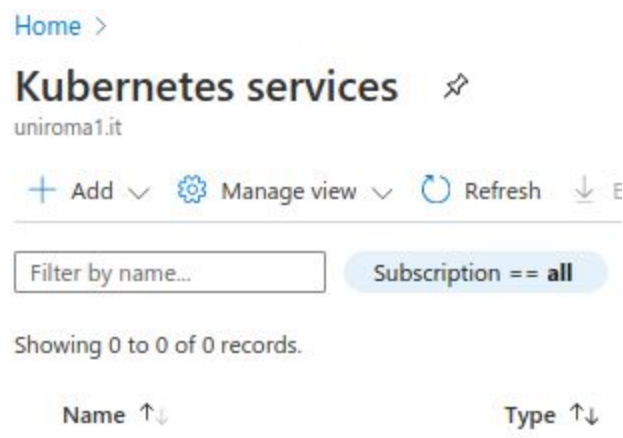
The screenshot shows the 'Resource groups' page in the Azure portal. The page title is 'Resource groups' with a star icon and the user 'uniroma1.it'. Below the title is a toolbar with '+ Add', 'Manage view', 'Refresh', 'Export to CSV', 'Open query', 'Assign tags', and 'Feedback'. A filter bar shows 'Filter by name...' and 'Subscription == all'. The main content area shows 'Showing 1 to 5 of 5 records.' and a table of resource groups.

| Name ↑↓ | Subscription ↑↓ | Location ↑↓ | |
|----------------------------------------------------------|--------------------|----------------------|-----|
| <input type="checkbox"/> CLCResourceGroup | Azure for Students | Germany West Central | ... |
| <input type="checkbox"/> cloud-shell-storage-west europe | Azure for Students | West Europe | ... |
| <input type="checkbox"/> DefaultResourceGroup-EUS | Azure for Students | East US | ... |
| <input type="checkbox"/> DefaultResourceGroup-SUK | Azure for Students | UK South | ... |
| <input type="checkbox"/> NetworkWatcherRG | Azure for Students | UK South | ... |

3) Creating a Kubernetes cluster:

After creating the resource group, we need to create a Kubernetes cluster. The steps to do so are as following:

- ✓ On the Kubernetes services page we need to click add and then Add a Kubernetes cluster.



The screenshot shows the 'Kubernetes services' page in the Azure portal. The page title is 'Kubernetes services' with a star icon and the user 'uniroma1.it'. Below the title is a toolbar with '+ Add', 'Manage view', 'Refresh', and 'E'. A filter bar shows 'Filter by name...' and 'Subscription == all'. The main content area shows 'Showing 0 to 0 of 0 records.' and a table with columns 'Name ↑↓' and 'Type ↑↓'.

| Name ↑↓ | Type ↑↓ |
|---------|---------|
|---------|---------|

- ✓ We need to choose our subscription type and the RG that we created in the previous step.

- ✓ In the cluster details, we need to give a name to our cluster and choose a region that is the closest to where the service will be accessed.
- ✓ In the Primary node pool section, we need to choose the Node size, which reflects the resources allocated to the various VMs and the node count, which is the number of nodes that will be allocated for our cluster. For an Azure Student Subscription the maximum number is 2.

Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Resource group * ⓘ
[Create new](#)

Cluster details

Kubernetes cluster name * ⓘ

Region * ⓘ

Availability zones ⓘ
i No availability zones are available for the location you have selected.
[View locations that support availability zones](#)

Kubernetes version * ⓘ

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to add additional node pools after creating your cluster. [Learn more about node pools in Azure Kubernetes Service](#)

Node size * ⓘ **Standard B2s**
2 vcpus, 4 GiB memory
[Change size](#)

Node count * ⓘ

- ✓ For node pools and Authentication, we leave everything at default settings.

- ✓ In the Networking section, we check the "Enable HTTP application routing" option since our application does not support HTTPS and leave everything else at default

Basics Node pools Authentication **Networking** Integrations Tags Review + create

You can change networking settings for your cluster, including enabling HTTP application routing and configuring your network using either the 'kubenet' or 'Azure CNI' options:

- The **kubenet** networking plug-in creates a new VNet for your cluster using default values.
- The **Azure CNI** networking plug-in allows clusters to use a new or existing VNet with customizable addresses. Application pods are connected directly to the VNet, which allows for native integration with VNet features.

[Learn more about networking in Azure Kubernetes Service](#)

Network configuration ⓘ

☒ kubenet

☐ Azure CNI

DNS name prefix * ⓘ

clc-api-cluster-dns ✓

Traffic routing

Load balancer ⓘ

Standard

Enable HTTP application routing ⓘ

☒

Security

Enable private cluster ⓘ

☐

Set authorized IP ranges ⓘ

☐

Network policy ⓘ

☒ None

☐ Calico

☐ Azure

i The Azure network policy is not compatible with kubenet networking.

- ✓ Integrations and Tags section can be left as they are.
- ✓ In the "Review + Create" section we click Create to let Azure allocate the resources and create our Cluster.

✓ Our cluster is ready!

Deployment is in progress

Deployment name: microsoft.aks-20201015211714 Start time: 10/15/2020, 9:36:49 PM
Subscription: Azure for Students Correlation ID: ebdbf4e2-470a-4e09-9d10-f6c0fd58b3d9
Resource group: CLCResourceGroup

Deployment details (Download)

| Resource | Type | Status | Operation details |
|---------------------------------|-------------------------------------|---------|-----------------------------------|
| clc-api-cluster | Microsoft.ContainerService/manag... | Created | Operation details |
| SolutionDeployment-202010152136 | Microsoft.Resources/deployments | OK | Operation details |

4) Connecting GitHub to our container registry: This method of automated deployment from GitHub allows Azure to fetch the files to build a Docker image and deploy it just as soon as it detects a change in our GitHub repository.

✓ On the cluster dashboard, in the deployment center part, we choose GitHub as the source.

clc-api-cluster | Deployment center (preview) ⚙

Kubernetes service

Search (Ctrl+/)

Diagnose and solve problems

Security

Kubernetes resources

- Namespaces (preview)
- Workloads (preview)
- Services and ingresses (preview)
- Storage (preview)
- Configuration (preview)

Settings

- Node pools
- Configuration
- Scale
- Networking
- Dev Spaces
- Deployment center (preview)

Launch a docker app to Azure Kubernetes cluster in a few quick steps
Configure a DevOps pipeline to deploy your application updates to this Kubernetes cluster

1 Source 2 Repository 3 Application 4 Resources

Select the code location

Azure Repos

Unlimited free private repos

GitHub

Home to the world's largest community of developers


Bitbucket Cloud

Hosted by Atlassian

External Git

Deploy from a public or private Git repo

- ✓ Then we need to authorize Azure to access our GitHub account and choose the repository and branch the project is stored on.



Source Repository Application Resources


Select a repository

My repositories All repositories

Repository *
muscasmarco/clc-serial-number

Branch *
main

- ✓ Azure will automatically detect the application when the Dockerfile is provided.



Source Repository Application Resources

Confirm application settings

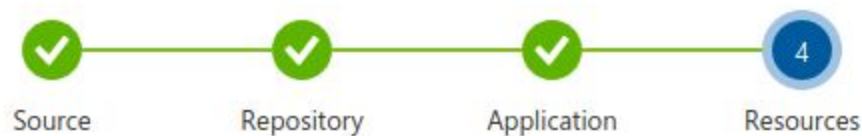
Detected the following Dockerfile

Dockerfile path *
/backend/Dockerfile

[/backend/Dockerfile](#)

```
1 FROM python:3.6.5-alpine
2 WORKDIR /project
3 ADD . /project
4 RUN pip install -r requirements.txt
5 CMD ["python", "app.py"]
6
```

- ✓ In the last section "Resources" we provide a Namespace and a Container Registry reference where the Docker image will be stored. We will use the existing container registry that we created before.



Almost there!

A pipeline is chosen automatically. You can change pipeline settings [here](#).

Namespace

Namespace *

[Create new](#) [Use existing](#)

Name *

clc-api-clustera57f

Container Registry


Container Registry *

[Create new](#) [Use existing](#)

Name *


clccontainerregistry

- ✓ An automatic configuration for Continuous Delivery will start. You can see the results in the following image.

| Namespace | Pipeline | Repository | Updated |
|-------------------------------------|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|
| clc-api-clustera57f | deploytoAksCluster.yml |  Cloud-Computing  master | a few seconds ago (View all runs) |

5) Accessing the application

In order to get the URL to our application, we need to go to the Services and ingresses menu in the cluster dashboard, section Ingresses. In the Hosts section you there is a link starting with the NAMESPACE. The application is accessible through that URL.

 **clc-api-cluster** | Services and ingresses (preview)
Kubernetes service

Access control (IAM)

Tags

Diagnose and solve problems

Security

Kubernetes resources

Namespaces (preview)

Workloads (preview)

Services and ingresses (preview)

Storage (preview)

Configuration (preview)

+ Add

Delete

Refresh


Show labels

Services

Ingresses

Filter by ingress name

Filter by namespace

| <input type="checkbox"/> | Name | Namespace | Class | Hosts | Address | Ports | Age ↓ |
|--------------------------|----------------------------------|---------------------|----------------------------|-----------------------------------------------|------------------------------------------------------------------------------------------------------|-------|-----------|
| <input type="checkbox"/> | clcapiclust-f51f | clc-api-clustera57f | addon-http-application-... | clc-api-clustera57f-clcapiclu | 51.116.141.243  | 80 | 8 minutes |

Test and validation:

- First, we want to check the performance of MongoDB Atlas and our three clusters. For the purpose of testing, we create dummy data to insert into our MongoDB database.

In the dataset, there are 3 main entities: users, products and serial numbers. At the starting point, there is still no relationship between them, like the users will not have registered any serial numbers by default and each valid serial numbers will be up for registration.

1. Generating the data

We used python scripts to generate a dataset for each entity. Namely, we chose to have a "catalogue" of 500 products, the names were sampled from an external dataset of Amazon electronics products that we found online. The code associated to each product was sequential number of 10 digits followed by a 'P'. The serial numbers were created also sequentially, the actual serial number has 5 digits followed by an 'S'.

For both products and serial numbers, not all digits correspond to a valid serial or a valid product. In the case of products that was mainly a memory concern, having $1e15$ serial numbers in total was not possible for a database registered with a free student account.

The fact that we also did not exhaust all 5 digits of a serial was for testing purposes. We split a portion of that such 70% of the possible SNs for each product are valid, while the remaining 30% is not. That is in line with the requirements we proposed in particular in the part about "checking if a serial number is valid or not".

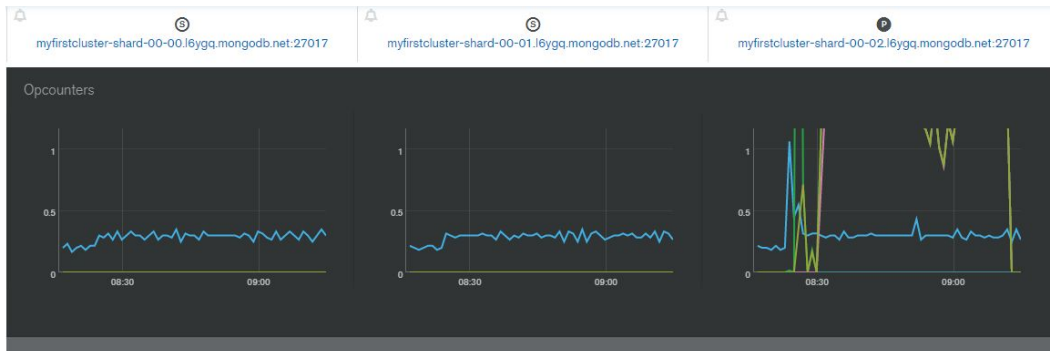
For users, we just created 1000 dummy accounts in the format "userXXX@clc.com" where XXX is a sequential integer, while the password is the same for everyone. New accounts can be created manually obviously.

2. Inserting the data

We used more python scripts to insert the data through the API. These scripts would read the datasets generated at the previous step and send requests to our API, that would in turn save the data in the database.

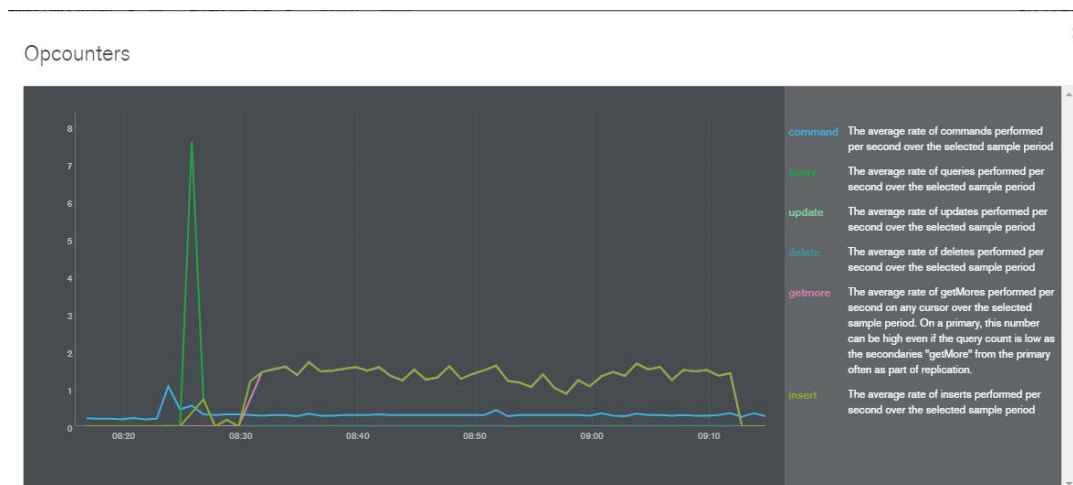
At each step of this "bulk insertion" the scripts would check for response codes from the HTTP replies. Everything went smoothly without need to review the architecture.

As we can see in the following pictures, inserting data in our collections is successful. We can notice that when we insert data we will only write on the primary node (The one on the right) and at the same time, the secondary nodes replicate the primary's oplog. (The light green line shows the average rate of insert)



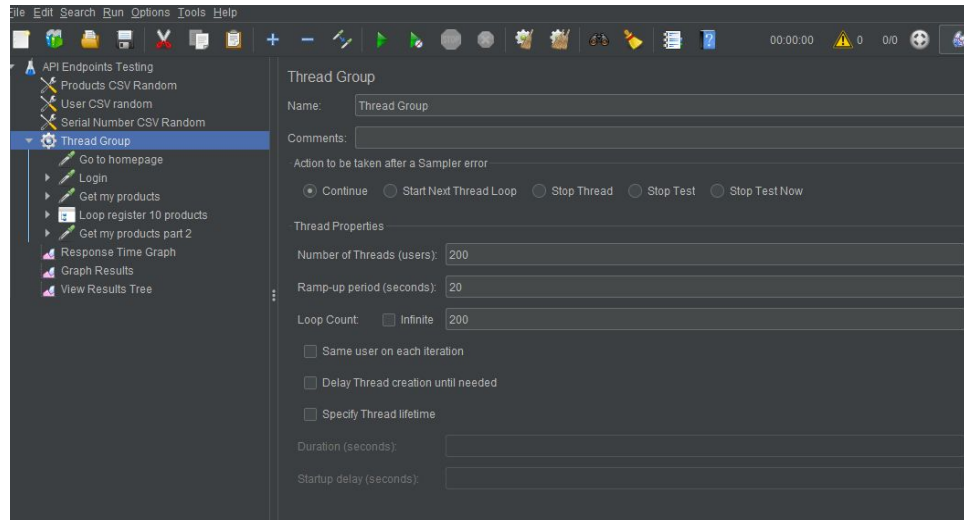
In the following picture, we look more in detail to the writing procedure to the primary node.

If we track the light green line, we'll see that we have three courses of inserting data. The first one belongs to adding product data, which contains 500 data points and was added one row per time. The second course belongs to adding user collection, which contains 1000 data points. We used bulk insert for that and as we can see, even it contained more data than product collection, the insertion was faster due to the bulk insert. The third course belongs to adding the serial number collection, which has 3.5 million data points and we used bulk insert for each 1000 data points at a time.



- For testing the performance and scalability of deployed APIs we use Apache JMeter which is an open source desktop application based on Java. Apache JMeter Load Testing is a crucial tool that determines whether the web application under test can satisfy high load requirements or not. It also helps to analyze the overall server under heavy load.

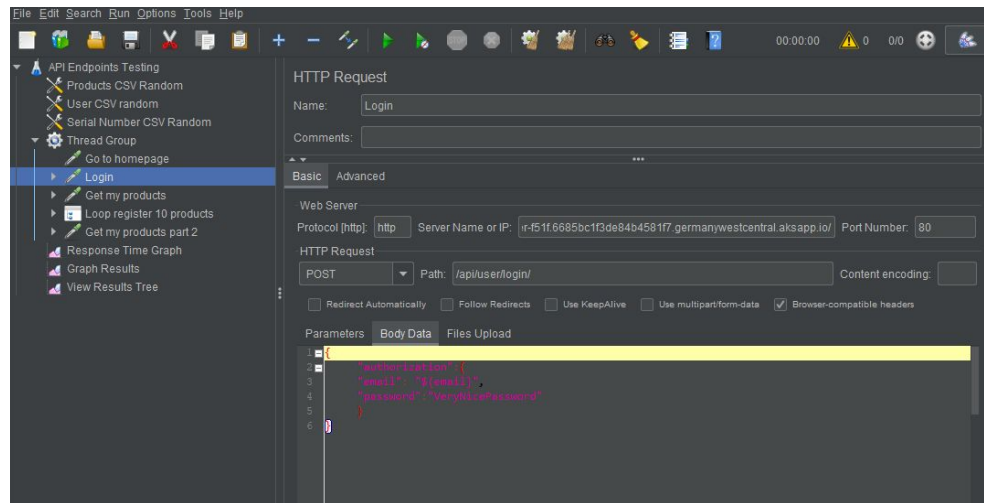
✓ First, we need to add a Thread Group:



Number of Threads: Number of users connects to the target website

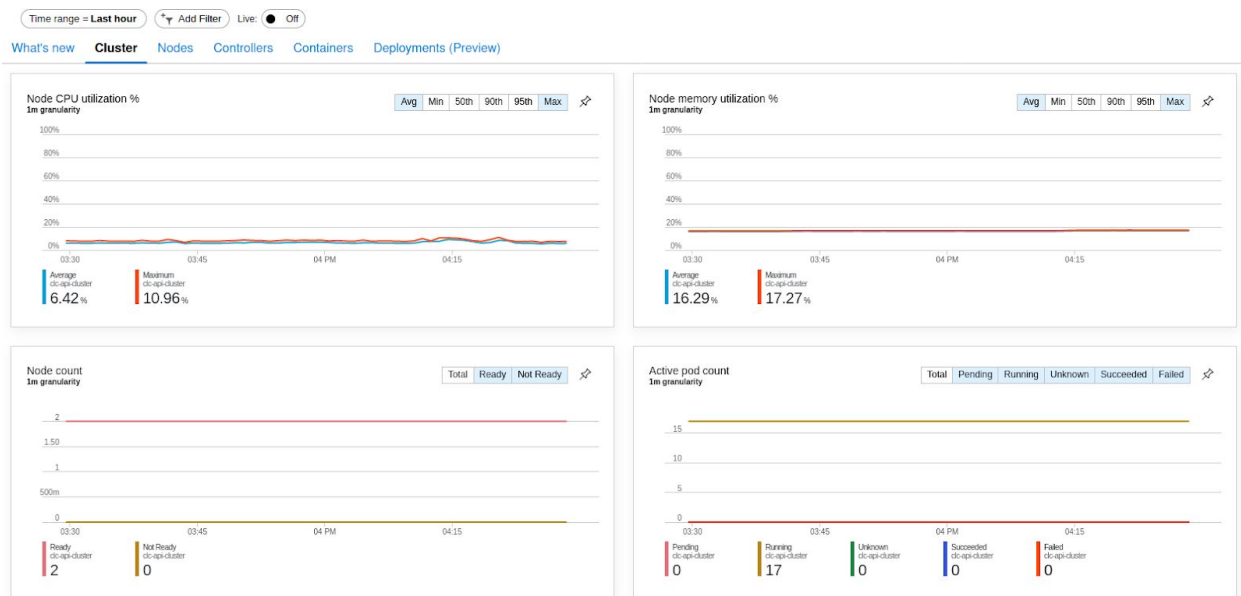
Loop Count: Number of time to execute testing

✓ Next, we need to add a JMeter element, we choose HTTP Request Defaults and then we need to choose HTTP Request from sampler. In an HTTP request page, we need to specify the server name or IP, the port, the type of HTTP request and the path. We performed the load testing for homepage, login and get products endpoints.



More results from the cluster testing:

What we saw from the Azure Dashboard of the cluster:



What was happening in JMeter while testing:

API_testing_jmeter.jmx (/home/marco/Desktop/TestingAPI/API_testing_jmeter.jmx) - Apache JMeter (5.3)

View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

Filename: Browse... Log/Display Only:

| Sample # | Start Time | Thread Name | Label | Sample Time(ms) | Status | Bytes | Sent Bytes |
|----------|--------------|-------------------|-----------------------|-----------------|--------|-------|------------|
| 581 | 15:58:21.164 | Thread Group 1-4 | Add new product | 7959 | ✓ | 401 | 496 |
| 582 | 15:58:22.852 | Thread Group 1-1 | Add new product | 6373 | ✓ | 401 | 496 |
| 583 | 15:58:28.219 | Thread Group 1-5 | Add new product | 1430 | ✓ | 401 | 496 |
| 584 | 15:58:29.649 | Thread Group 1-5 | Get my products pa... | 125 | ✓ | 469 | 450 |
| 585 | 15:58:29.775 | Thread Group 1-5 | Go to homepage | 78 | ✓ | 187 | 193 |
| 586 | 15:58:29.853 | Thread Group 1-5 | Login | 67 | ✓ | 336 | 347 |
| 587 | 15:58:29.920 | Thread Group 1-5 | Get my products | 104 | ✓ | 170 | 450 |
| 588 | 15:58:28.087 | Thread Group 1-7 | Add new product | 2112 | ✓ | 401 | 496 |
| 589 | 15:58:30.199 | Thread Group 1-7 | Get my products pa... | 115 | ✓ | 308 | 450 |
| 590 | 15:58:30.314 | Thread Group 1-7 | Go to homepage | 84 | ✓ | 187 | 193 |
| 591 | 15:58:30.398 | Thread Group 1-7 | Login | 70 | ✓ | 336 | 347 |
| 592 | 15:58:30.469 | Thread Group 1-7 | Get my products | 91 | ✓ | 308 | 450 |
| 593 | 15:58:25.976 | Thread Group 1-2 | Add new product | 5149 | ✓ | 401 | 496 |
| 594 | 15:58:26.318 | Thread Group 1-9 | Add new product | 5098 | ✓ | 401 | 496 |
| 595 | 15:58:26.039 | Thread Group 1-8 | Add new product | 6195 | ✓ | 401 | 496 |
| 596 | 15:58:27.100 | Thread Group 1-6 | Add new product | 5575 | ✓ | 401 | 496 |
| 597 | 15:58:29.226 | Thread Group 1-1 | Add new product | 5196 | ✓ | 401 | 496 |
| 598 | 15:58:30.024 | Thread Group 1-5 | Add new product | 5791 | ✓ | 209 | 496 |
| 599 | 15:58:29.109 | Thread Group 1-3 | Add new product | 6791 | ✓ | 401 | 496 |
| 600 | 15:58:30.560 | Thread Group 1-7 | Add new product | 6098 | ✓ | 209 | 496 |
| 601 | 15:58:31.416 | Thread Group 1-9 | Add new product | 5438 | ✓ | 401 | 496 |
| 602 | 15:58:29.123 | Thread Group 1-4 | Add new product | 7877 | ✓ | 401 | 496 |
| 603 | 15:58:31.125 | Thread Group 1-2 | Add new product | 5941 | ✓ | 401 | 496 |
| 604 | 15:58:29.110 | Thread Group 1-10 | Add new product | 8154 | ✓ | 401 | 496 |
| 605 | 15:58:32.234 | Thread Group 1-8 | Add new product | 5098 | ✓ | 401 | 496 |
| 606 | 15:58:32.675 | Thread Group 1-6 | Add new product | 4730 | ✓ | 401 | 496 |

☒ Scroll automatically? ☐ Child samples? No of Samples: 606 Latest Sample: 4730 Average: 2306 Deviation: 2424

152 2020-10-21 15:58:54.466 INFO o.a.j.m.StandardJMeterEngine: All thread groups have been started
153 2020-10-21 15:58:54.466 INFO o.a.j.m.StandardJMeterEngine: Thread started: Thread group 1-2

The test results are successful despite the limitations given by the platform.