Bachelor’s thesis

Bachelor of Engineering (B.Eng.), Information and Communications Technology

2021

Huy Thanh Nguyen

MICROSERVICES, RESTFUL API AND A USE CASE

BACHELOR’S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor of Engineering (B.Eng.), Information and Communications Technology

2021 | 30 pages

Huy Thanh Nguyen

MICROSERVICES, RESTFUL API AND A USE CASE

Application design has been changing rapidly since 2010. Most of the modern applications have been using microservices architecture design which brings several benefits to the users. This thesis covers the advantages of the microservices approach in designing applications and REST API in microservices. Furthermore, this thesis examines the roles of REST API in the modern data transferring world. The objective of this thesis is to understand the microservices architecture by conducting a literature review and build a functional cloud microservices application with Python as the programming language. Suomen Asiakastieoto Oy is a Finnish company that provides data services for financial, corporate, and risk management besides sales and marketing. The result was a working cloud application that makes requests to the Asiakastieto API and receives the result set in JSON/XML format.

KEYWORDS:

Microservices, REST API, Virtual Machine, Asiakastieto

CONTENTS

FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
Curl	Client Uniform Resource Locator
CSS	Cascading Style Sheets
DevOps	Development and Operations
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IT	Information Technology
JSON	Javascript Object Notation
OS	Operating system
REST	Representational State Transfer
REST API (noun)	Representational State Transfer Application Programming Interface
RESTful API (adjective)	Representational State Transfer-ful Application Programming Interface
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

1 INTRODUCTION

While business has been moving forward rapidly, application architecture has also been evolving quickly to meet business needs. Understanding microservices architecture will help business owners to organize business models and define a process on how business units will communicate with each other.

The reason why microservices and RESTful API were selected as the topic for this thesis is that microservices have been the ideal architecture for corporations looking for continuous delivery. This ideal architecture comprises small modules and each module represents one functionality that is the same in the case study. Even though there are some disadvantages, the benefits of microservices outweigh the inconvenience. It is believed that the microservices architecture will be a standard way to develop clouds in the future.

The goal of this thesis is a working cloud application that can send requests to Asiakastieto API and receive responses. This thesis covers microservices architecture and its background, patterns, and topologies. Furthermore, this thesis will also look into container and container orchestration within this microservices architecture picture. Besides, readers will become acquainted with API, RESTful API, API mechanism, HTTP methods, and CRUD and examine in detail a cloud application that serves a business purpose.

This thesis has two main parts: the theoretical part and the project part. The theoretical part introduces the key concepts The practical part deals with creating an application using microservices architecture explained in the theoretical part. More specifically, this thesis will cover microservices architecture and topologies, containers and container orchestration, REST API, and some comparisons between microservices and containers. The first chapter is the introduction of this thesis. The second chapter of this thesis contains the general structure and topologies of microservices. The general structure of microservices drills down to dividing what the application is needed to be capable of into small self-functioning endpoint nodes. From that, the advantages and disadvantages will be uncovered to give a deeper understanding. Furthermore, containers will also be looked into and a comparison between virtual machines and containers will be made. The third chapter will introduce RESTful API and HTTP. RESTful API uses simple HTTP protocol principles to provide support to create, read, update and delete data. These operations are referred to all together as CRUD Operations. The fourth chapter of this thesis will include a part of the Asiakastieto project which is an application using REST API to make a request to third-party API and return a result set in JSON/XML format. The fifth chapter is the conclusion.

2 MICROSERVICES AND CONTAINERS

2.1 Microservices

Over the past decade, better system architecture has been found. We have been studying the older technologies, integrating new technologies, and witnessing waves of changes in technology that lead to new IT systems benefiting companies and developers.

Continuous delivery is a software development method that helps teams to create working pieces of software quickly. Continuous delivery using the agile manifesto helps the software production process to move from the development environment to the production environment fast, efficiently, and effectively. Continuous delivery also allows us to divide a large architecture system into smaller endpoints that link to each other. Protocols to transfer data between endpoints have been developed by the understanding of how the web operates. Hexagonal architecture which was invented by Alistair Cockburn helps us to separate the layers without dependencies and contamination of user interface with business logic. Hardware virtualization and platform virtualization enable the machine to have the capability to scale up or down. In addition, infrastructure automation provides load balancing and handling traffic for large-scale systems. Some large corporations advocate a small team owning the full lifecycle of all services while others share with the public how to build antifragile systems at a large scale that would have been impossible just a decade ago.

From the need of moving application development toward an agile approach, microservices architecture was born. Many organizations have found that by having a fine-grained microservices architecture, the software can be delivered faster and new technologies adoption is easier. This allows companies, especially large-scale global companies, to make decisions freely on manipulating the system and to respond more quickly to the required changes.

Microservices are a system software development technique that creates a system composed of fine-grained self-functioning modules with application programming interfaces and functionalities [1]. This type of system architecture has gained its popularity in recent year because of the very fact that companies and enterprises are moving toward DevOps and Agile style of developing software systems.

Each service component must be able to communicate with other service components inside the same microservice architecture. A microservice must have the ability to send data to and receive data from other microservices within the same system. The system as a whole processes data that it receives from external endpoints such as web applications, other APIs, and databases. The ability to send and receive data among microservices requires some endpoints which are where APIs come along in this architecture.

For this reason, an interface is going hand in hand with each microservice which makes the API an important factor in this architecture. In order to take advantage of microservices' scaling capabilities, routing between services needs to be set up logically. Because of all these characteristics, the RESTful APIs are the most suitable type of API for building interfaces between services since performance, scalability, simplicity, portability, and modifiability are the main principles behind REST design. Each service is assigned to a general interface that represents the service's functionality.

2.1.1 Patterns

The general structure of microservices drills down to dividing what business needs the application to be capable of into small self-functioning endpoint nodes. Other than that, there should be at least a layer between these nodes and the internet to receive and respond to requests sent by clients usually in HTTP protocol. This layer also sends and receives requests with endpoint nodes separately (which keep these nodes from directly sending requests to each other) as in Figure 1 (below).

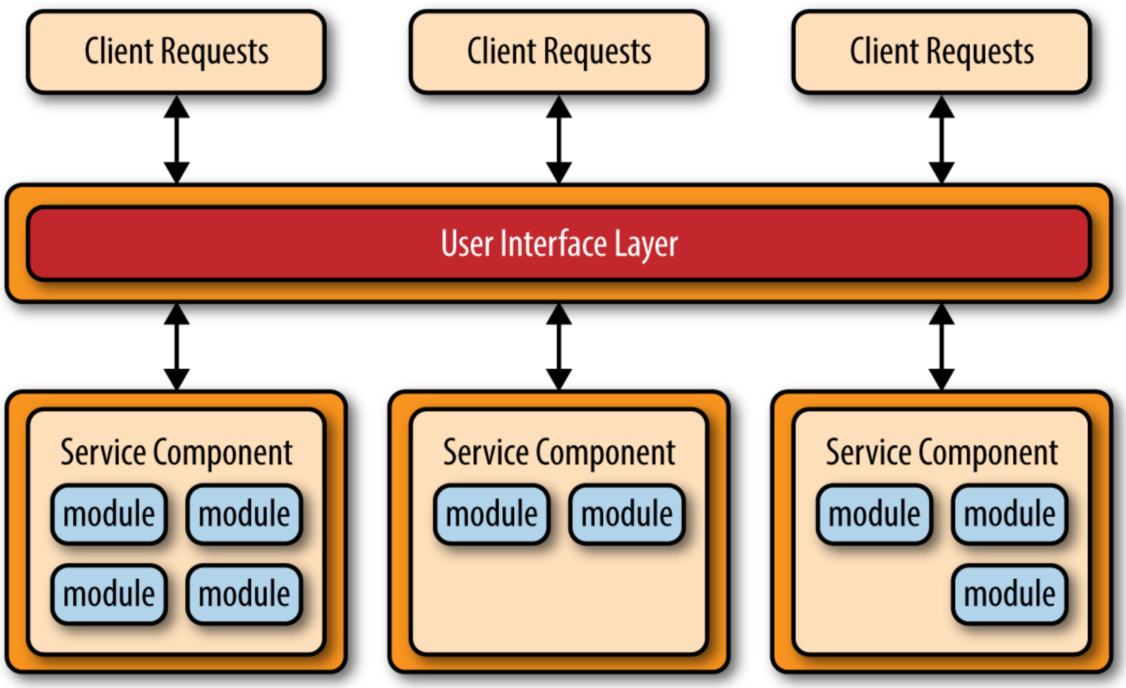


Figure 1. Basic Microservices architecture pattern (O'Reilly 2015).

Each functionality that an endpoint node represents can be a function that stands on its own or it could be a standalone part of a large application that is used in business.

Distributing is one main characteristic of the system built using this architecture which means one endpoint node has the capability to run on its own and could run with or without other endpoint nodes within the same system. This characteristic enables microservice architecture applications to have the ability to scale and early deploy while developing other functionality.

Despite the fact that there is no predetermined way to build a system, there have been many variants of this architecture. Most of these variants were built based on these three topologies that use the following interfaces between clients and back-end processing components:

- Using a RESTful interface
- Using a web app
- Using a(n) message/integration broker

2.1.2 Topologies

Using a RESTful interface

The first topology is using a RESTful interface for exchanging requests with external sources and communicating with endpoint nodes as in Figure 2 (below).

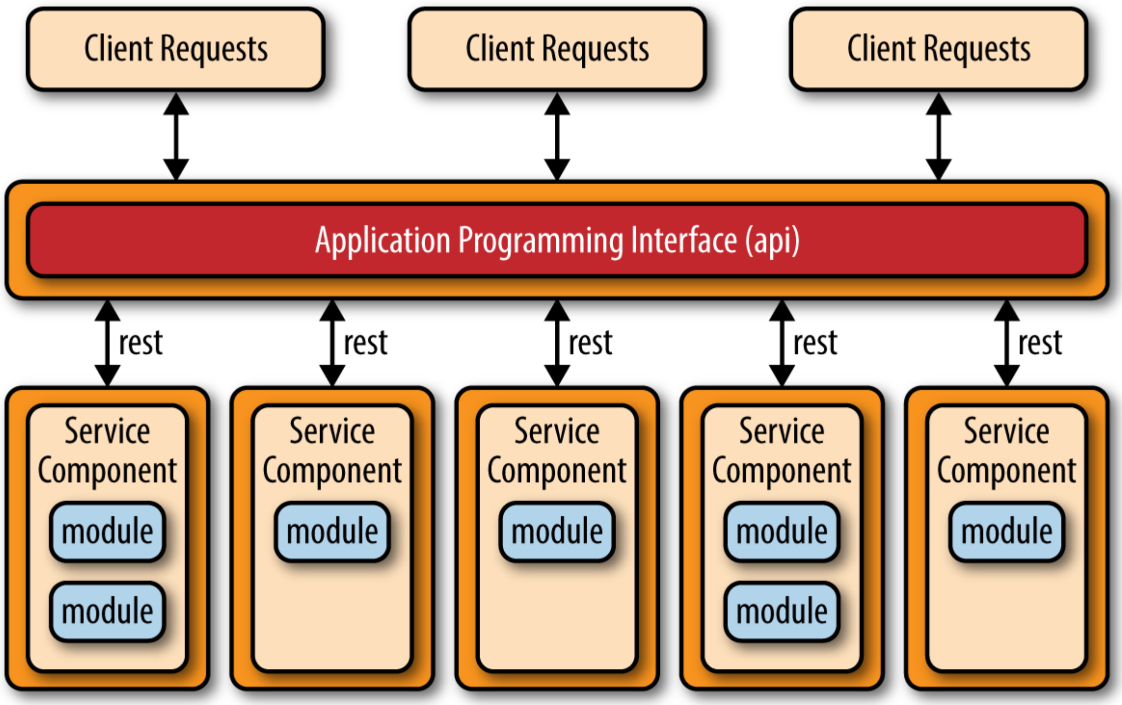


Figure 2. API REST-based topology (O’reilly 2015).

This type of topology uses a RESTful interface layer for receiving and responding to requests with clients and with endpoint nodes. Each endpoint node contains one or two modules that define a small part of a decent business application. These endpoint nodes are fine-grained self-functioning and independent from each other. This type of topology is suitable for servers and exposes endpoints to be used outside of the application. This would only require the front-end developers to know the address/URL of the endpoint, the requirement when sending requests, and what type of information this endpoint returns. One example for this type of topology is the Google Sheets API v4 which have a service endpoint base URL `https://sheets.googleapis.com` with methods that come with it (“batchUpdate”, “create”, “get”, etc.) that do specific function (“applies one or more updates to the spreadsheet”, “creates a spreadsheet”, “returns the newly created spreadsheet”, etc.) (Google Sheets API v4 2019).

Using a web app

The second topology is using a web app with an interface for communicating with clients and endpoint nodes as in Figure 3 (below).

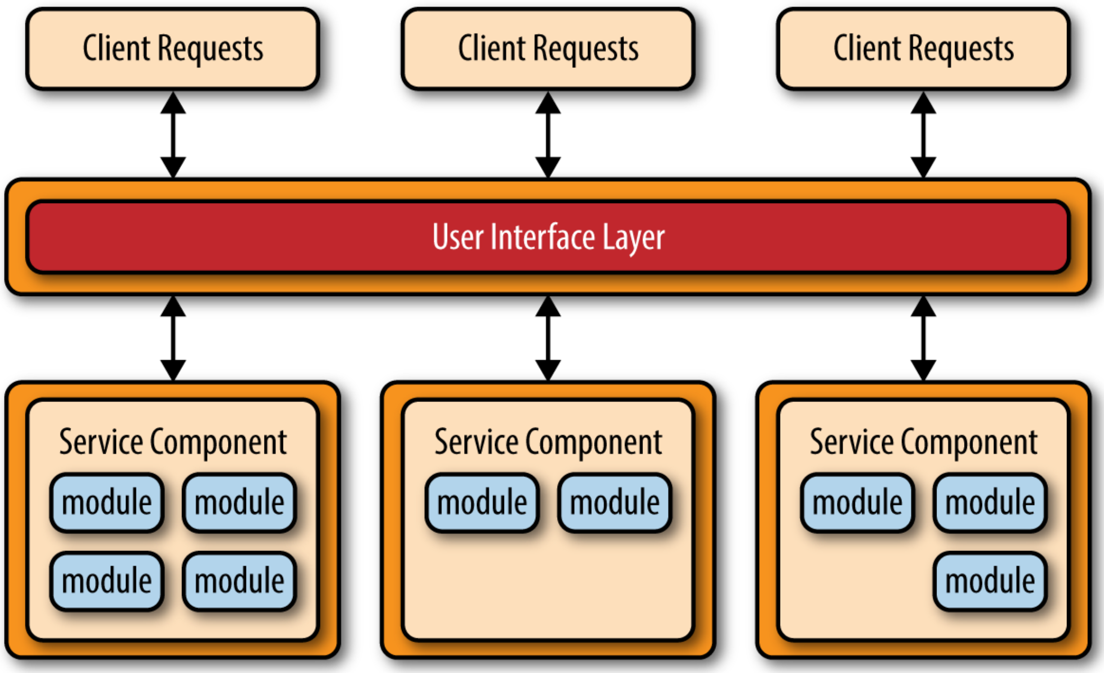


Figure 3. Application REST-based topology (O'reilly 2015).

This type of topology uses a web app interface (using HTML, CSS, and Javascript) for users to interact. This web application interface will also in charge of sending requests and receiving responses from endpoint nodes. Same as the previous topology, each endpoint node should also self-functioning and independent from each other. However, each endpoint node contains at least two modules and each node should perform a broader functionality than nodes from the API REST-based topology. The web app interface should be deployed somewhere and accessible by web browsers. This type of topology is suitable for non-complexity business applications.

Using a(n) message/integration broker

The third topology is using a(n) message/integration broker in between the app and endpoint nodes as in Figure 4 (below).

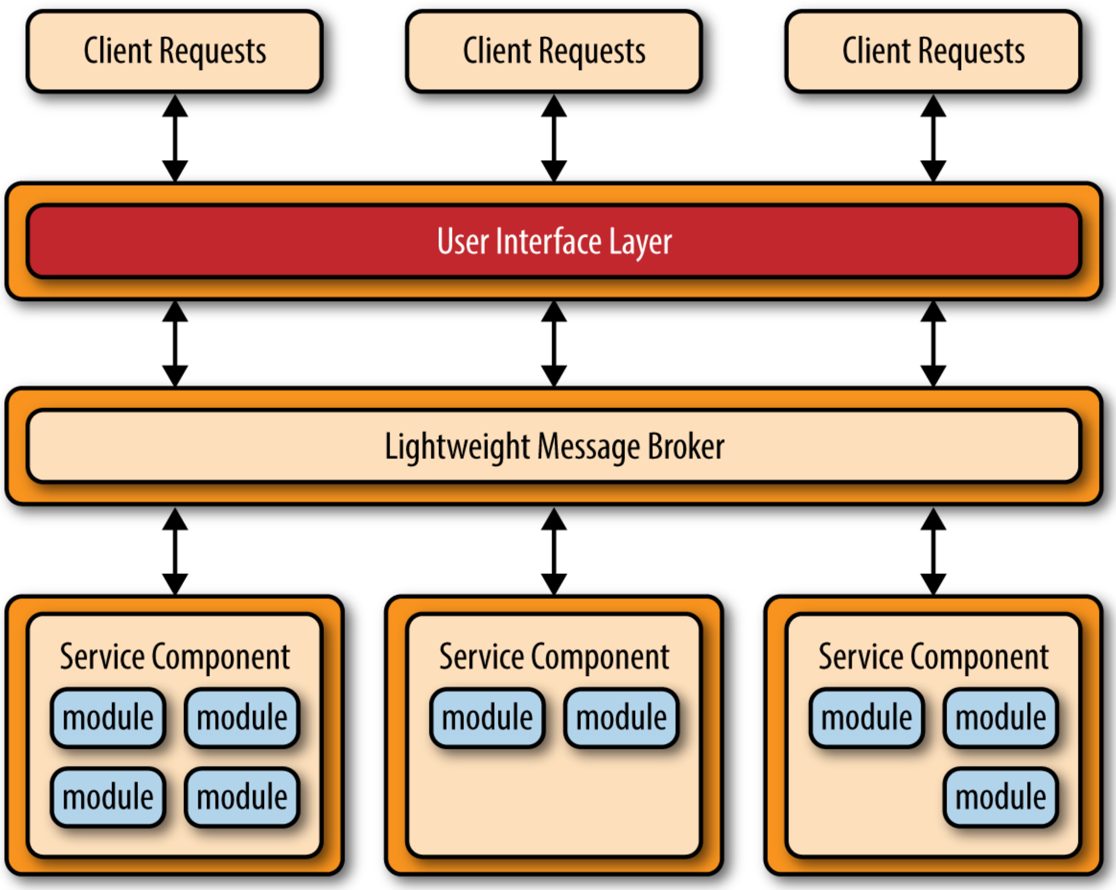


Figure 4. Centralized messaging topology (O'reilly 2015).

This structure is nearly identical to the app REST-based structure except this will use a message/integration broker in between the app interface and the endpoint nodes. This broker plays a role as a faster means of transportation request and does not orchestrate the traffic. This topology does not need to use a RESTful API since the broker does the transportation of the requests and messages. This type of topology is suitable for the big and complicated app since it has multiple advantages such as error handling and asynchronous requests. However, due to all traffic goes through one broker, out of quota can be a problem.

2.1.3 Advantages and disadvantages of microservices architecture

Microservices architecture brings many benefits due to each endpoint defines one functionality and is independent of other endpoints. If one endpoint within the application fails, the whole application will not crash. Besides, the flexibility on technology to use on each endpoint allows users to test different technology stacks on different endpoints as needed without concern about dependencies, and reverting is much easier.

Microservices architecture allows a better understanding of each service while approaching the applications. Furthermore, each endpoint has a decent amount of code with a specific scope that allows faster and continuous deployment [2]. Furthermore, due to microservices architecture, this allows users to scale up and down the most needed services instead of the whole application which will save money.

Besides the advantages, microservices architecture also has several disadvantages. First of all, data transferring between endpoints is complicated. Each endpoint defines a functionality separately – this requires each endpoint to be built thoroughly to be able to handle communication between endpoints, there might be additional code updating in some cases. This would make the endpoints bulkier after a period of time which can cause a delay in request processing.

Besides, microservices is hard to test and debugging due to individual endpoint has to be confirmed before testing starts and each endpoint has its own log [2]. Furthermore, the deployment process will be more complicated than deploying a monolithic application inside a container. Last but not least, microservices would require users to manage multiple databases and resources at the same time.

2.2 Containers

Each microservice application is associated with many libraries, dependencies, and configuration files and this complexity will need to be addressed. Deployments are more complicated and it is hard to specify beforehand. Containers can help to simplify these issues and is a great tool to use with the microservice application. A container is a tool that packs code and all of its needed packages into one package so the package can be deployed or move from one computing environment to another fast [3]. Containers help the application to run smoothly when moving from one environment to another. A container is a package that consists of the application itself, libraries, dependencies, configuration files, and binary files. Docker is a tool that simplifies the process of pack, ship, and run the application as an independent container, which can run anywhere.

2.2.1 Virtual machines and containers

Besides containers, a virtual machine is also a popular choice for deploying applications. Let's have a look at a server architecture using a virtual machine for hosting applications. The right architecture from Figure 5 below describes a server that uses virtual machines for running applications. The system includes hardware, host OS, hypervisor (what enables the machine to spin up VMs), and a VM for each hosted application (which contains guest OS, libraries, and the app itself). For hosting another application, we would need to create another VM which also contains guest OS, libraries, and the app itself. Using VMs consumes resources quickly and makes the application unnecessarily heavier.

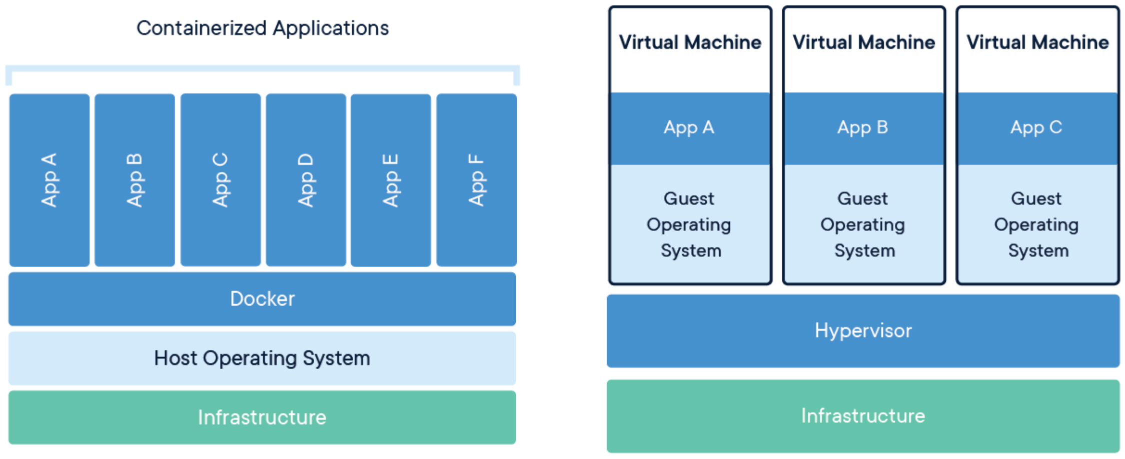


Figure 5. Containers and Virtual Machines (Docker Inc. 2020).

The system of containerized applications includes a host OS, runtime engine, and containerized applications (which contains libraries and application). Without guest OS needed, the containerized application system is much more lightweight than using virtual machines.

With virtual machines, the application package that is transferring and moving around is the virtual machine itself which includes OS and the application. A server that runs 2 virtual machines will contain a hypervisor and two independent operating systems on top of it.

Thanks to this trait, one server can contain more containers than VMs. Furthermore, a virtual machine will take time to load the operating system before running the application while the container can run the application almost immediately. Besides, containers also allow users to have broader modularity. Instead of having all packed applications within a container, the packed applications can be divided into smaller units (for example the application back-end, the application front-end, database, etc.) and this characteristic is essential for implementing microservices architecture application. This would keep the container application fairly simple to maintain and manage without the need to rebuild the entire application.

2.2.2 Docker

Docker Engine is the industry’s de facto container runtime that runs on various Linux (CentOS, Debian, Fedora, Oracle Linux, RHEL, SUSE, and Ubuntu) and Windows Server OS. Docker creates a simple universal packaging approach that packs up all application dependencies inside a container which is then run on Docker Engine. Docker Engine enables containerized applications to run anywhere consistently in any environment, solving the need of adding and setting dependencies for developers and operations teams [4].

Docker is an abstraction built on top of low-level container technologies such as libvirt or LXC. It provides the command-line interface as well as an HTTP API interface to create, public container images and to run containers as in Figure 6. Docker provides a way to pack an application and all of its system dependencies into a standardized unit. A standardized packaging is a Docker image which his layered immutable approach to creating an image. Docker is not only a tool, but it is also an ecosystem of tools and services such as Docker hub – a central repository used to host shared Docker images. Docker has grown to be widely used as a deploy mechanism for applications. Because the guarantee of an image will always run the same regarding the environment, this gives developers parity across development tests and production.

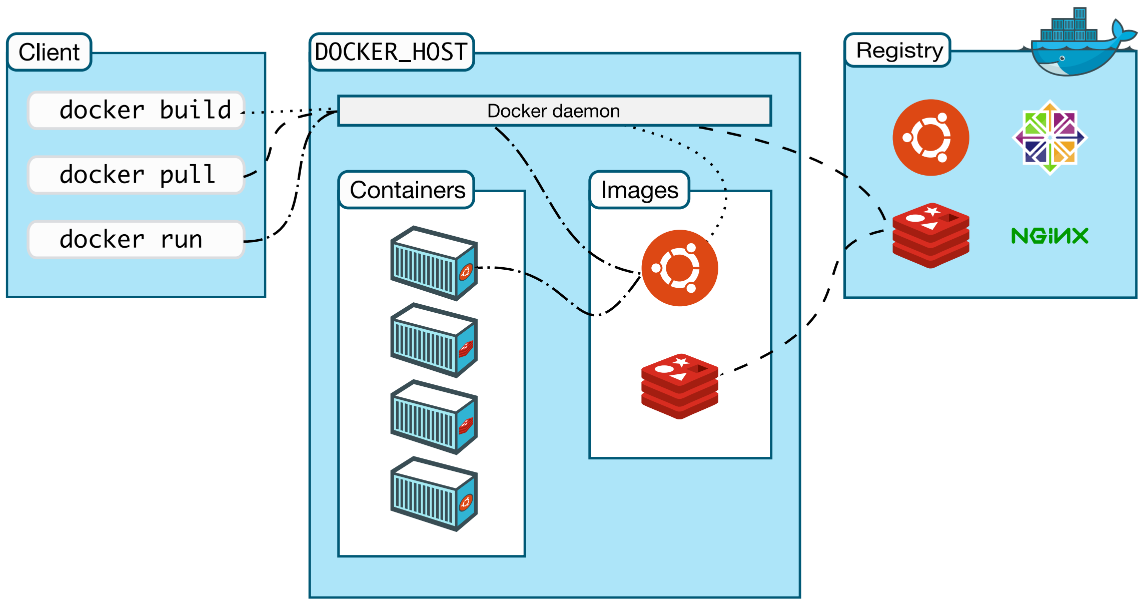
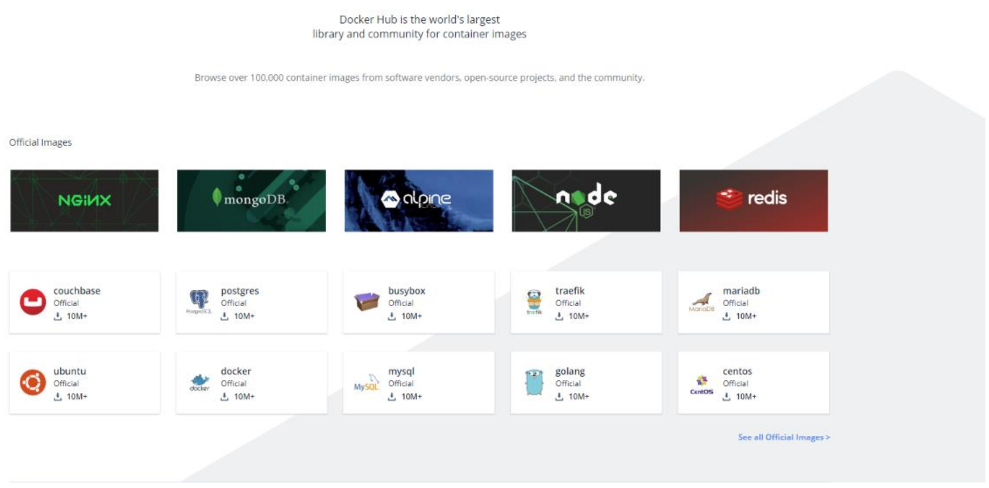


Figure 6. Docker architecture (Docker Inc. 2019).

Docker has a number of components, although it might not seem like it when you run Docker on a Linux machine. Docker is actually a client-server application. The Docker binary does not interact with containers or images directly. The Docker binary simply accepts command line calls and sending instructions to Docker daemon via HTTP or via TCP socket. If both components are on the same host machine, these communication mechanisms are also used.

The Docker Host is the server-side component of the Docker engine. Docker Host houses the Docker daemon which is the process that actually interacts with Docker images and containers. The Docker Host also houses the Docker containers and Docker images. Docker Host exposes an HTTP API that can be used by other programs to manage containers and images.

The final component required to work with Docker is a Docker registry. A Docker registry is a remote service that essentially houses published Docker images. The official Docker registry is called Docker Hub and it is managed directly by Docker. Docker Hub houses thousands of pre-built Docker images contributed by third-party vendors, individual developers, and Docker themselves as in Figure 7 [5].


Docker Hub is the world's largest library and community for container images

Browse over 100,000 container images from software vendors, open-source projects, and the community.

Figure 7. Docker Hub official images (hub.docker.com 2020).

Most common applications, programming languages runtime, and databases have official Docker Hub images available. Running a Mysql database on Docker Host as easy as running command `docker run mysql`. In this case, a client will make an API call to the Docker HTTP API endpoint for creating a container. This will trigger the Docker daemon to pull the latest Docker image for Mysql from Docker Hub and creates a running container from it on a Docker Host. All container and image operations are managed this way.

Another component in the Docker ecosystem is the Docker machine. Docker machine is a command-line tool that can be used to manage Docker Host. Docker machine allows non-Linux users to run Docker clients locally on their localhost and issue Docker command to a remote virtual machine be running at a local hypervisor or a cloud provider. An example of what Docker machine can do is the Docker machine `create` command which will create a fresh new Docker Host named dev that can be controlled using your local Docker client. To use the new Docker Host named dev, the Docker machine can be used to set the environment variable to point to the new machine, and then a command can be issued.

Another useful component in Docker ecosystem is Docker Swarm which is a clustering tool which allows user to manage a group of Docker Hosts as if it was a single Docker Host. Docker Swarm publishes an API almost identical to the normal Docker API which means most tooling is compatible. We can use Docker client to run containers with Docker Swarm and easy to scale out to multiple Hosts.

Sometimes users want to containerize an application with multiple components like databases or other services. Using Docker Compose, users can specify a group of containers and wire them together.

2.2.3 Container orchestration and Kubernetes

With Docker, users can run a single instance of the application with a simple Docker `run` command. In this case, to run a node.js-based application, users only need to run `docker run nodejs` command, but that is just one instance of an application on one Docker Host. What happens when the number of users increases and that instance is no longer able to handle the load? By deploying an additional instance of the application by running the Docker `run` command multiple times, users have to keep a close watch on the load and performance of applications and deploy additional instances if needed; besides, users also need to keep a close watch on the health of these applications and if a container was to fail, users should be able to detect that and run the Docker `run` command again to deploy another instance of that application. What if the Docker Host crashes and is inaccessible? The containers hosted on that Docker Host become inaccessible too. How should these issues be solved?

Container orchestration is a solution that consists of a set of tools and scripts that can help host containers in a production environment. Typically a container orchestration solution consists of multiple Docker Hosts that can host containers, this way even if one fails, the application is still accessible through the others. The container orchestration solution easily allows you to deploy hundreds or thousands of instances of your application with a single command used for Docker Swarm. Some orchestration solutions can help you automatically scale up the number of instances when users increase and scale down the number of instances when the demand decreases. Some solutions can even help you in automatically adding additional hosts to support the user load. Besides, the container orchestration solutions also provide support for advanced networking between these containers across different hosts as well as load balancing user requests across different hosts. They also provide support for sharing storage between the hosts, configuration management, and security within the cluster.

There are multiple container orchestration solutions available today, Kubernetes arguably is the most popular of them all. Kubernetes is a bit difficult to set up and get started but provides a lot of options to customize deployments and has support for many different vendors. Kubernetes is now supported on all public cloud service providers like GCP, Azure, and AWS. And the Kubernetes project is one of the top-ranked projects on Github. By using Kubernetes CLI, users can run thousands of instances of the same application with a single command [6]. Kubernetes can be configured to scale up and down the number of instances and infrastructure itself automatically based on user load. Kubernetes can upgrade these instances of the application in a rolling upgrade fashion one at a time with a single command. If something goes wrong, it can help you roll back these images with a single command [6]. Kubernetes can help you test new features of an application by only upgrading a percentage of these instances through the A/B testing method. The Kubernetes open architecture provides support for many different network and storage vendors. Kubernetes supports a variety of authentication and authorization mechanisms. All major cloud service providers have native support for Kubernetes.

Kubernetes uses the Docker Host to host applications in the form of Docker containers. It need not be Docker all the time. Kubernetes supports alternatives to Dockers.

A Kubernetes architecture consists of a set of nodes as in Figure 8 [6]. A node is a worker machine where containers will be launched by Kubernetes. If a node on which the application is running fails, the application will go down so more than one node is required. A cluster is a set of nodes grouped together, this way even if one node fails, the application is still accessible from other nodes. The master is a node with the Kubernetes control plane components installed. The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes. When Kubernetes is installed on a system, the following components are installed: an API server, an etcd server, a kubelet service, a container runtime engine like Docker, a number of controllers, and schedulers. The API server acts as the front-end for Kubernetes, the users' management devices and command-line interfaces all talk to the API server to interact with the Kubernetes cluster. Next is the etcd key-value store, the etcd is a distributed reliable key-value store used by Kubernetes to store all data used to manage the clusters. Etcd is responsible for implementing a log with the cluster to ensure there are no conflicts between the masters. The scheduler is responsible for distributing work or containers across multiple nodes, it looks for newly created containers and assigns them to nodes. The controller is the brain behind orchestration, they are responsible for noticing and responding when nodes, containers, or endpoints go down. The controller makes the decision to bring up new containers in such cases the container runtime is the underlying software that is used to run containers. Kubelet is the agent that runs on each node in the cluster, the agent is responsible for making sure that a container is running on the node as expected.

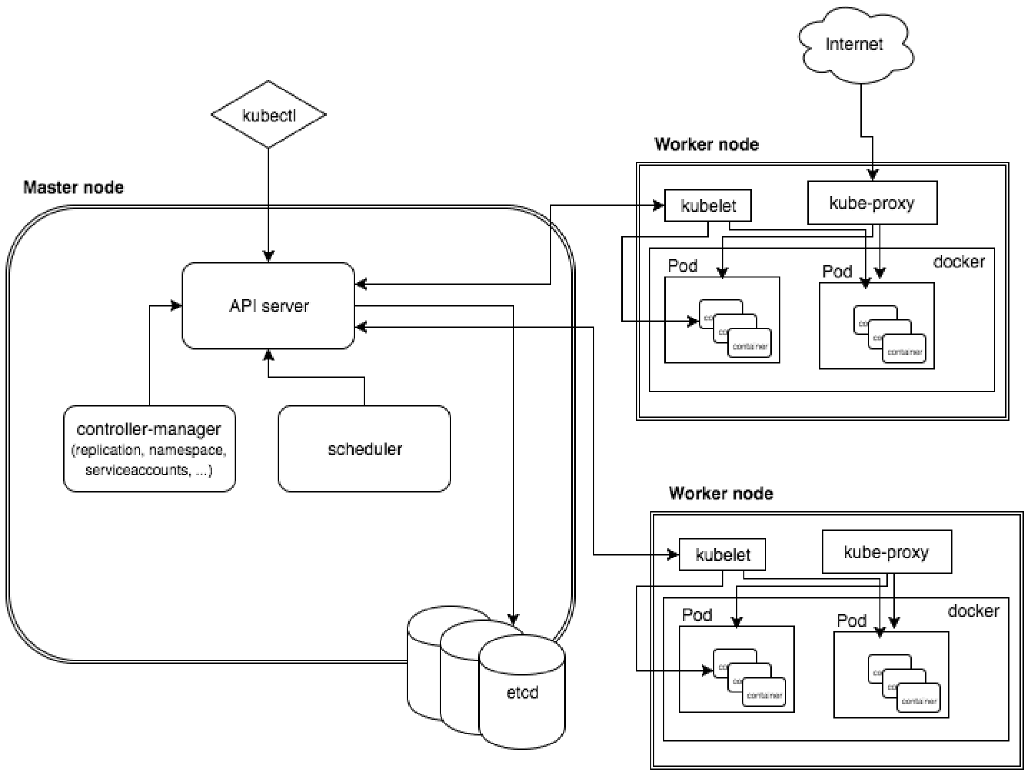


Figure 8. High-level Kubernetes architecture diagram showing a cluster with a master and two worker nodes (Aqua Security 2018).

3 RESTFUL API AND HTTP

3.1 API

API is short for application programming interface that defines “a set of rules that allow programs to talk to each other” [7]. API is the communication interface between the front-end with back-end logic. Figure 9 visualized API’s role in applications which is taking requests to the back-end server. Once the server responds, the interface will send that response back to the front-end terminal (web browser in this case).

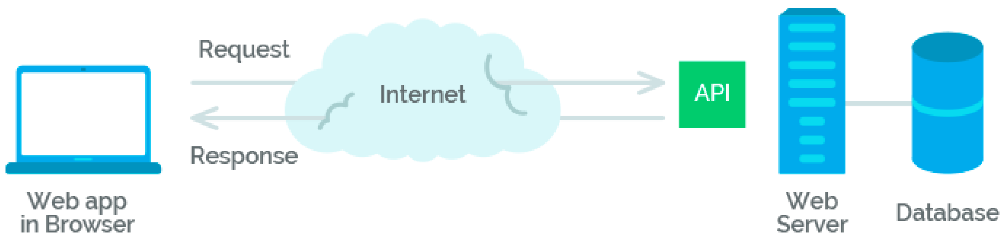


Figure 9. Web service topology with API (MLSDev 2020).

One of the benefits of using API is that the logic of the back-end is hidden from the outside world [9]. This would help companies and corporations provide services to everyone without exposing internal information. No matter which front-end terminal is used, as long as the request is the same, the same result will be sent back from the API.

3.2 RESTful API

REST is short for Representational State Transfer and is more like an architectural way to build applications rather than a protocol or a standard [9]. RESTful API is the API that implements the REST architecture pattern for creating web services. Many software companies use this software architecture for building applications including

Facebook, Google Maps, and eBay. RESTful API is an API that follows the REST principles.

Five principles for REST software architecture:

- Client and server always communicate through an interface and are separated from each other. Either client-side or server-side can be changed and updated as long as the interface stays untouched. Client´-side should be the side that sends requests.
- Requests are sent separately and not related to each other hence no rely on session to keep the connection on.
- Requests and resources are cacheable on both client and server-side which can help to speed up the performance.
- Applications can be layered to keep the logic hidden from the client side.
- The interface stays untouched and is always the same.

3.3 HTTP methods and CRUD

Most applications we use these days follow the client-server architecture. The application itself is the client or front-end part, under the hood it needs to talk to a server or the back-end to get or save the data. This communication happens using the HTTP protocol – the same protocol that powers web communication. There are exposed endpoints that clients and call the services by sending HTTP requests. RESTful API uses simple HTTP protocol principles to provide support to create, read, update and delete data. These operations are referred to all together as CRUD Operations [9].

HTTP Methods:

- GET: Retrieve data from a specified resource
- POST: Submit data to be processed to a specified resource
- PUT: Update a specified resource
- DELETE: Delete a specified resource
- HEAD: Same as get but does not return a body
- OPTIONS: Returns the supported HTTP methods

- PATCH: Update partial resources

Assuming that there is a need for a photo-sharing application with a RESTful API back-end. Users should be able to add new photos to their list as well as share a specific photo with others. Additionally, users should be able to see photos shared by other users. To meet these requirements, the API would have to be able to make different actions on the user photo list. Different HTTP methods can be used here. For example, the GET method can be used to request the interface to fetch and return all the photos within a photo list. The POST method can be used to create a new list or add a photo into a photo list. DELETE method can be used to delete a specific photo or a specific photo list.

With this photo-sharing application, most of the requests surround CREATE, READ, UPDATE and DELETE (CRUD in short). CRUD defines the cycle of records management within a database.

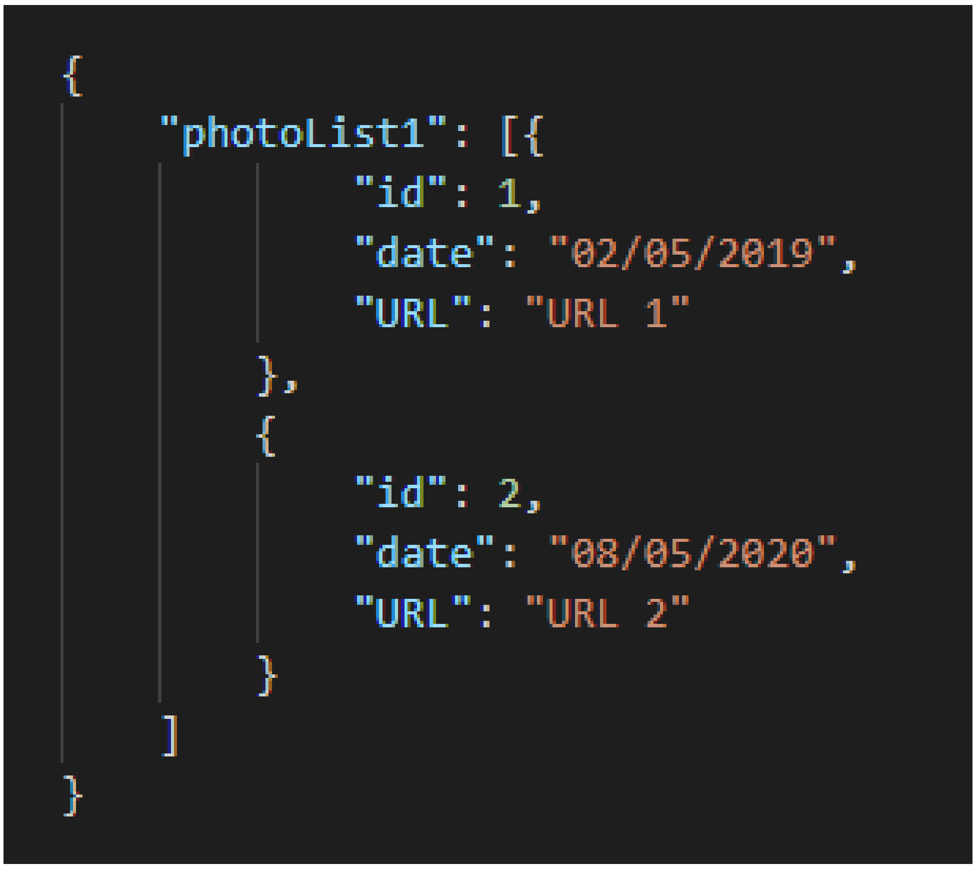
While sending the data over HTTP to the interface for back-end processing, the data needs to be in a pre-agreed form with the back-end. There are two widely used forms: JSON and XML. JSON will be the targeted format within this part due to its readability and popularity over XML.

JSON stands for JavaScript Object Notation. JSON is a plaintext format that can represent a complex structure of data. JSON can keep data type strings, arrays, numbers, or objects. The data string needs to be JSON-ified before sending it to the interface.

Basic JSON syntax:

- Arrays items will be wrapped within square brackets
- Curly brackets can be used to denote objects
- Items exist in pairs
- Strings are wrapped within quotation marks

Example of JSON syntax for a photo list:



```
{
    "photoList1": [{
            "id": 1,
            "date": "02/05/2019",
            "URL": "URL 1"
        },
        {
            "id": 2,
            "date": "08/05/2020",
            "URL": "URL 2"
        }
    ]
}
```

Figure 10. Example JSON format.

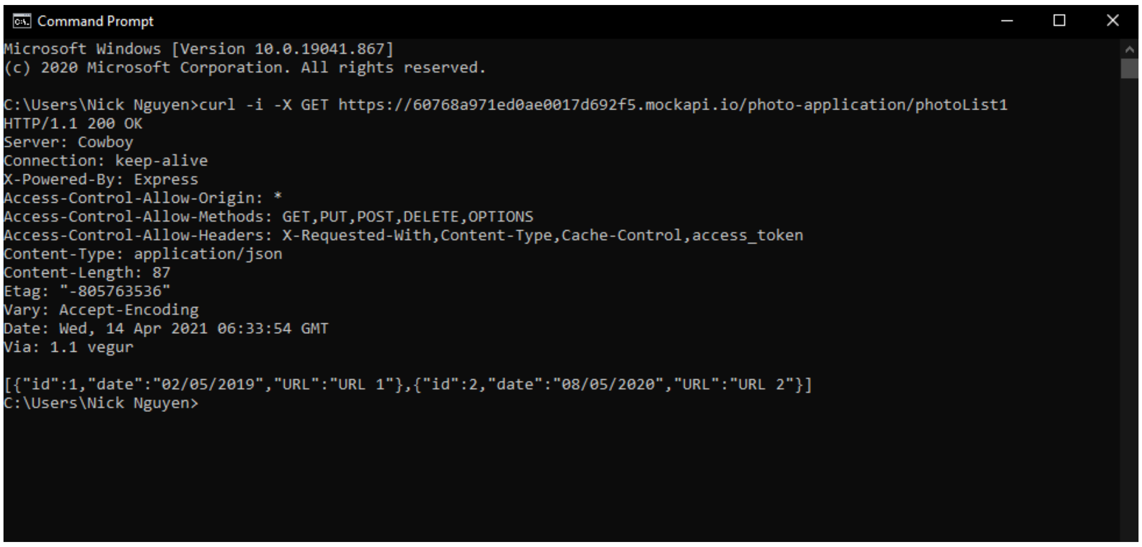
From Figure 10, there are two photos within photoList1, first photo has id 1 and a datestamp of “02/05/2019”. The URL of the first photo is stored as “URL 1”. The second photo has id 2 and a datestamp of “08/05/2020”. The URL of the second photo is stored as “URL 2”.

3.4 Using curl and Postman to test endpoints

Testing plays a major part in software development to ensure the software works as intended. Curl is a command-line tool that can transfer data using different types of network protocols through URLs. Curl can be used to send requests to an endpoint and receive the response.

On a Windows computer, `curl -i – X GET https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1` command can be used on Command Prompt to send

GET request to the endpoint https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1. The response will be as Figure 11 below.

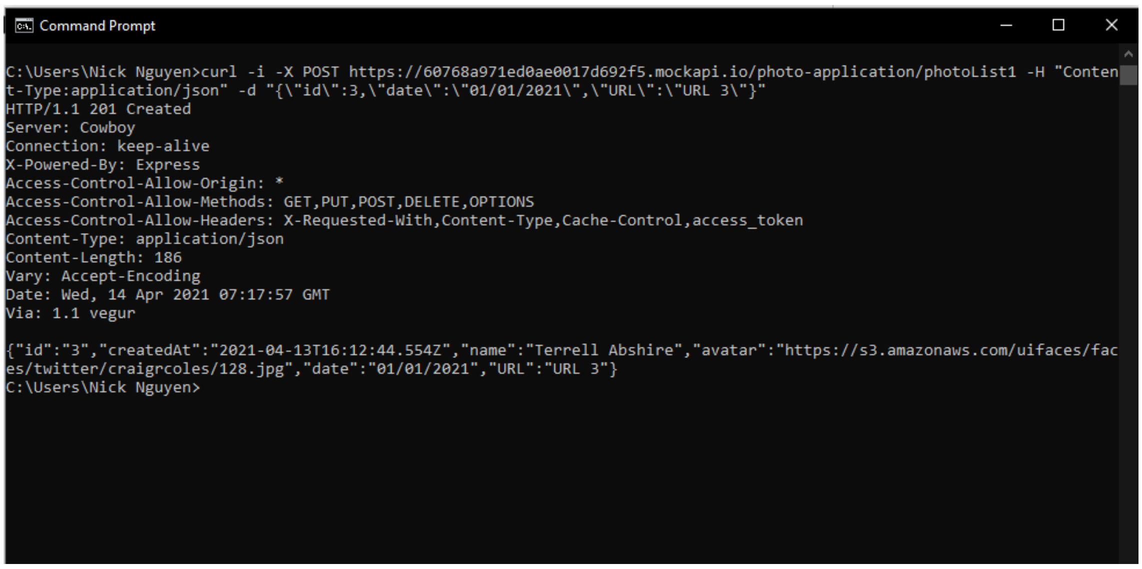


```
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Nick Nguyen>curl -i -X GET https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1
HTTP/1.1 200 OK
Server: Cowboy
Connection: keep-alive
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,PUT,POST,DELETE,OPTIONS
Access-Control-Allow-Headers: X-Requested-With,Content-Type,Cache-Control,access_token
Content-Type: application/json
Content-Length: 87
Etag: "-805763536"
Vary: Accept-Encoding
Date: Wed, 14 Apr 2021 06:33:54 GMT
Via: 1.1 vegur

[{"id":1,"date":"02/05/2019","URL":"URL 1"},{"id":2,"date":"08/05/2020","URL":"URL 2"}]
C:\Users\Nick Nguyen>
```

Figure 11. Sending GET request using curl.

Having a look at the response, the first few lines are the response header. The HTTP status is 200 OK and the Content-Length is 87 bytes. Content-Type is application/json and at the end of the response is the JSON format of the response body.

For adding new records into object livePhoto1, we can send a POST request to the endpoint https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1 as Figure 12 below.

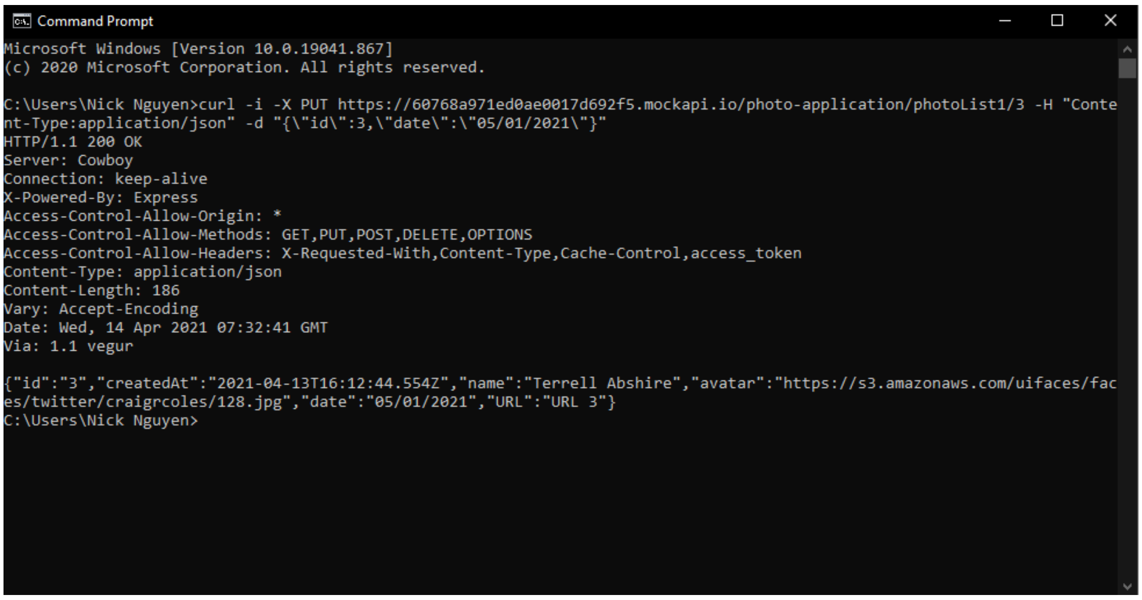


```
C:\Users\Nick Nguyen>curl -i -X POST https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1 -H "Conten
t-Type:application/json" -d "{\"id\":3,\"date\":\"01/01/2021\",\"URL\":\"URL 3\"}"
HTTP/1.1 201 Created
Server: Cowboy
Connection: keep-alive
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,PUT,POST,DELETE,OPTIONS
Access-Control-Allow-Headers: X-Requested-With,Content-Type,Cache-Control,access_token
Content-Type: application/json
Content-Length: 186
Vary: Accept-Encoding
Date: Wed, 14 Apr 2021 07:17:57 GMT
Via: 1.1 vegur

{"id":"3","createdAt":"2021-04-13T16:12:44.554Z","name":"Terrell Abshire","avatar":"https://s3.amazonaws.com/uifaces/fac
es/twitter/craigrcoles/128.jpg","date":"01/01/2021","URL":"URL 3"}
C:\Users\Nick Nguyen>
```

Figure 12. Sending POST request using curl.

Same as GET request, the first few lines are the response header. The HTTP status is 201 and the Content-Length is 186 bytes. The last line returns the response body which is the new object under the photoList1 array.

For updating an object within the photoList1 array, HTTP POST methods can be used as Figure 13 below.



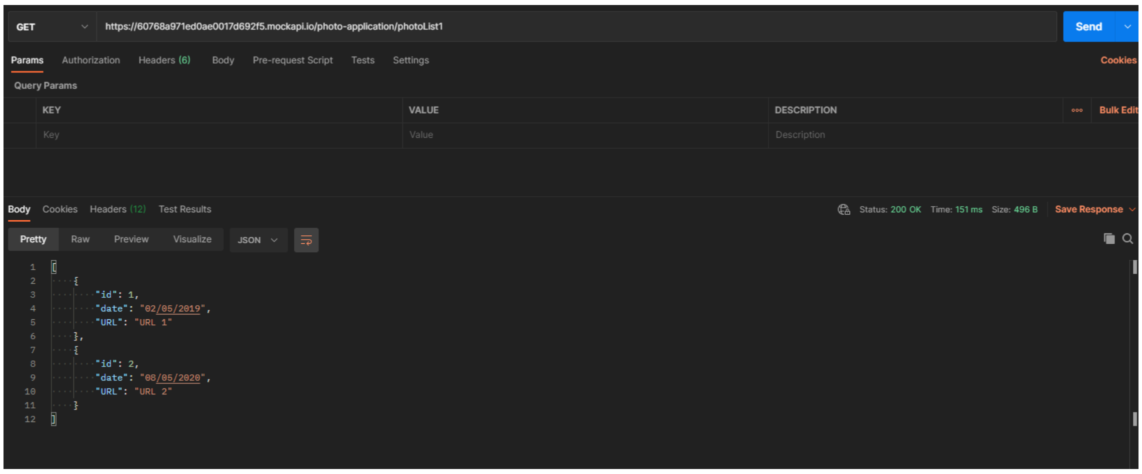
```
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Nick Nguyen>curl -i -X PUT https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1/3 -H "Conte
nt-Type:application/json" -d "{\"id\":3,\"date\":\"05/01/2021\"}"
HTTP/1.1 200 OK
Server: Cowboy
Connection: keep-alive
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,PUT,POST,DELETE,OPTIONS
Access-Control-Allow-Headers: X-Requested-With,Content-Type,Cache-Control,access_token
Content-Type: application/json
Content-Length: 186
Vary: Accept-Encoding
Date: Wed, 14 Apr 2021 07:32:41 GMT
Via: 1.1 vegur

{"id":"3","createdAt":"2021-04-13T16:12:44.554Z","name":"Terrell Abshire","avatar":"https://s3.amazonaws.com/uifaces/fac
es/twitter/craigrcoles/128.jpg","date":"05/01/2021","URL":"URL 3"}
C:\Users\Nick Nguyen>
```

Figure 13. Sending PUT request using curl.

The command `curl -i -X PUT https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1/3 -H "Content-Type:application/json" -d "{\"id\":3,\"date\":\"05/01/2021\"}"` will update date value from “01/01/2021” to “05/01/201”. The HTTP status is 200 OK.

Using curl with Command Prompt is not convenient for everyone, especially for users who have not worked with command lines before. That is where Postman comes into the picture. Postman is a convenient tool with a user-friendly graphical user interface for API testing. Different HTTP methods can be sent with Postman and the response from endpoints will also be visible in Postman. Users can save and organize tests into folders and collections.

Figure 14 below shows how the same GET request can be done using Postman.



https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1

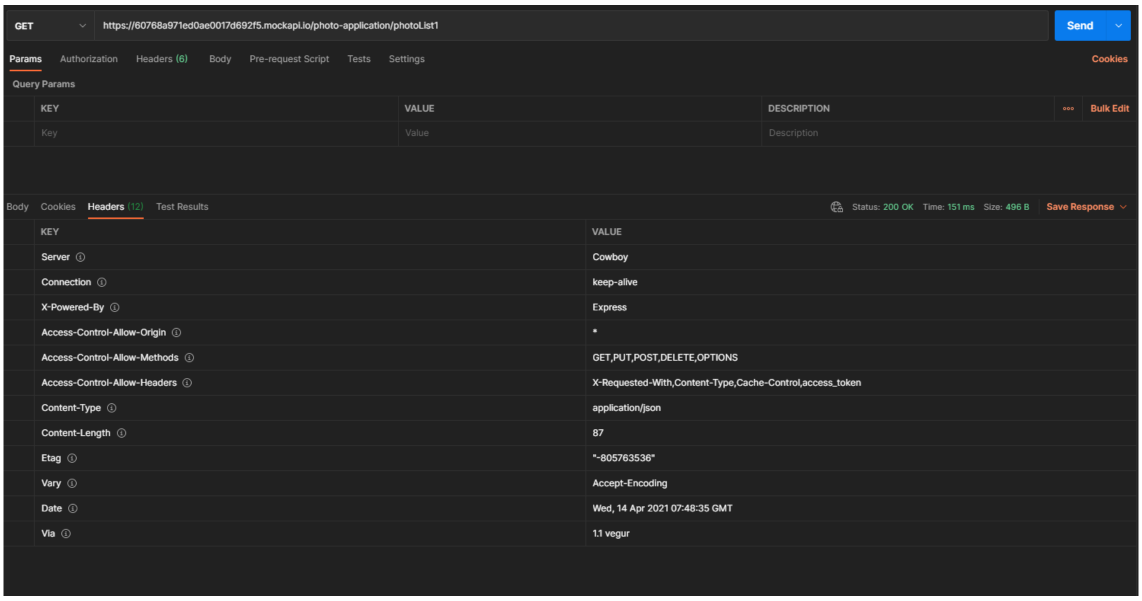
KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 151 ms Size: 496 B

```
[
    {
        "id": 1,
        "date": "02/05/2019",
        "URL": "URL 1"
    },
    {
        "id": 2,
        "date": "08/05/2020",
        "URL": "URL 2"
    }
]
```

Figure 14. Sending get request using Postman.

For seeing the response header, users can switch to the headers tab as Figure 15 shows.



https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1

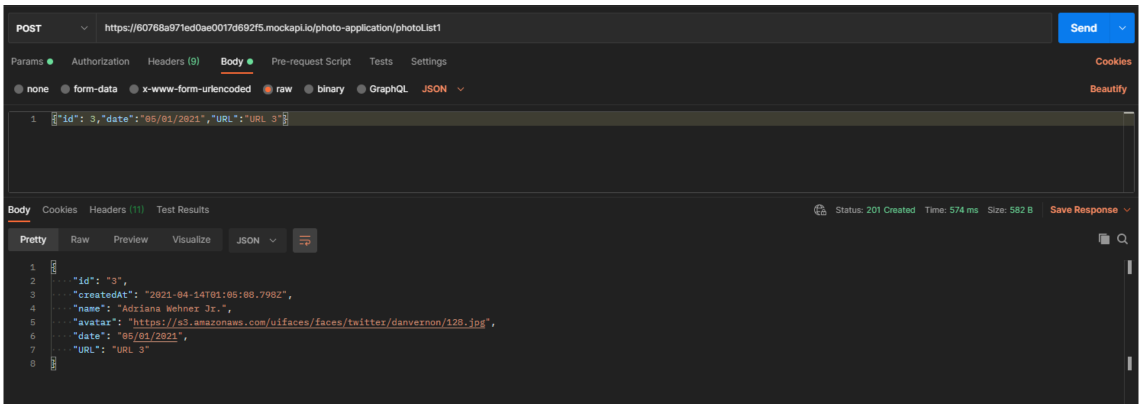
KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 151 ms Size: 496 B

KEY	VALUE
Server	Cowboy
Connection	keep-alive
X-Powered-By	Express
Access-Control-Allow-Origin	*
Access-Control-Allow-Methods	GET,PUT,POST,DELETE,OPTIONS
Access-Control-Allow-Headers	X-Requested-With,Content-Type,Cache-Control,access_token
Content-Type	application/json
Content-Length	87
Etag	"-805763536"
Vary	Accept-Encoding
Date	Wed, 14 Apr 2021 07:48:35 GMT
Via	1.1 vegur

Figure 15. Response header in Postman.

Figure 16 shows the same information will be posted to the endpoint by using Postman instead of curl.



POST https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1

Body: raw, JSON

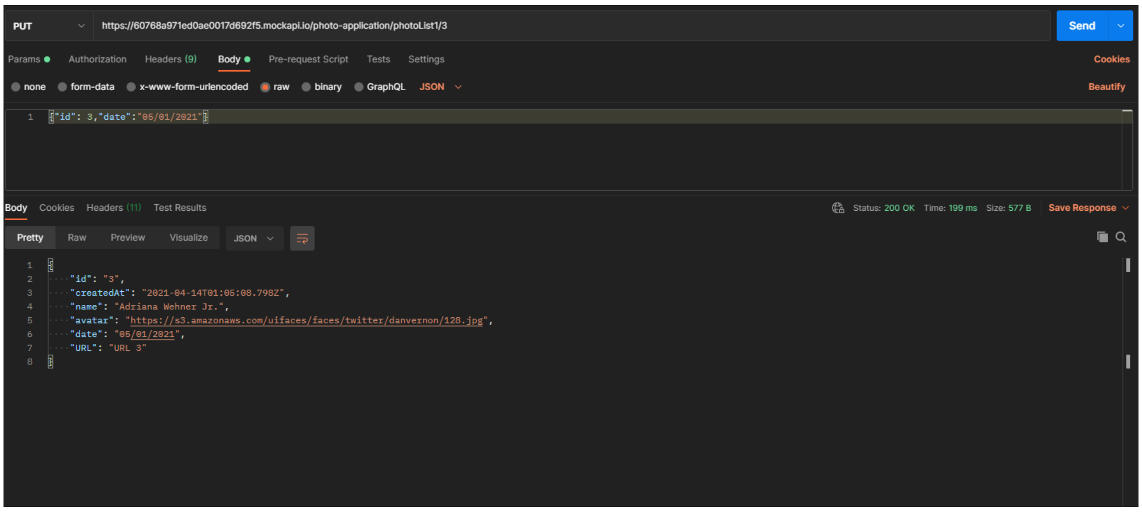
```
{"id": 3,"date":"05/01/2021","URL":"URL 3"}
```

Status: 201 Created Time: 574 ms Size: 582 B

```
{
    "id": "3",
    "createdAt": "2021-04-14T01:05:08.798Z",
    "name": "Adriana Wehner Jr.",
    "avatar": "https://s3.amazonaws.com/uifaces/faces/twitter/danvernon/128.jpg",
    "date": "05/01/2021",
    "URL": "URL 3"
}
```

Figure 16. Sending post request using Postman.

Figure 17 shows the same information will be put to the endpoint by using Postman instead of curl.



PUT https://60768a971ed0ae0017d692f5.mockapi.io/photo-application/photoList1/3

Body: raw, JSON

```
{"id": 3,"date":"05/01/2021"}
```

Status: 200 OK Time: 199 ms Size: 577 B

```
{
    "id": "3",
    "createdAt": "2021-04-14T01:05:08.798Z",
    "name": "Adriana Wehner Jr.",
    "avatar": "https://s3.amazonaws.com/uifaces/faces/twitter/danvernon/128.jpg",
    "date": "05/01/2021",
    "URL": "URL 3"
}
```

Figure 17. Sending put request using Postman.

4 CASE: ASIAKASTIETO API

Starting with project background, an existing cloud application makes a call to Asiakastieto API to fetch a list of companies and those companies' information such as revenue, number of employees, credit ranking, etc. After having the response from Asiakastieto API, this cloud application should send the response to an external database of records system where this information is visible and associated with the company entity.

The project situation is that the cloud application is written in PHP 5.0 deployed on a Debian server. Due to business growth, the application needs to handle more requests hence moving from Apache to Nginx and complicating support for PHP. The Nginx server is lighter and doesn't require many resources to run. Furthermore, Nginx can handle a large number of requests without leaving much memory footprint and good scalability for future development. Additionally, for consistency between existing cloud applications, this application should be re-written in Python.

The expecting result after the thesis work is the application should be re-written in Python and deployed to AWS Lambda as a serverless microservice. The new application should be easy to add more functionalities when needed. The process of developing additional functionalities should not affect the application as a whole. If a new functionality fails to run, other existing functionalities should be running normally.

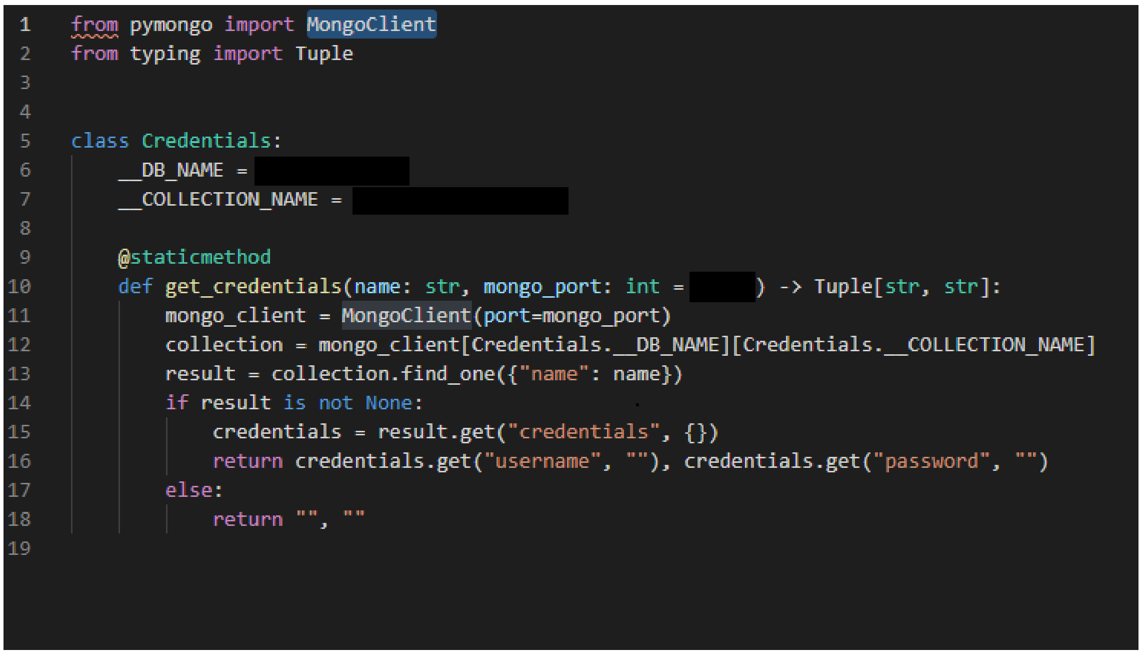
The new application will be re-write and organized following microservices architecture style. Each main functionality or node will be re-written into one folder. The structure of the application including subfolders and many python files in the root folders. The files in the root folder will invoke the functionalities. This will create a layer to hide application logic to avoid cyber attacks and will help with the agility in process of developing new functions.

4.1 Credentials folder

The first folder is the credentials folder. This folder contains 2 files: __init__.py and credential.py.

The __init__.py file is for letting python know that the directories should be treated and can be imported as a package. This would prevent directories with popular names accidentally from hiding needed modules that can occur on the module search path.

The credentials.py file contains a static method that will connect to a Mongo database to get the API credentials as Figure 18.



```python
from pymongo import MongoClient
from typing import Tuple


class Credentials:
    __DB_NAME = [redacted]
    __COLLECTION_NAME = [redacted]

    @staticmethod
    def get_credentials(name: str, mongo_port: int = [redacted]) -> Tuple[str, str]:
        mongo_client = MongoClient(port=mongo_port)
        collection = mongo_client[Credentials.__DB_NAME][Credentials.__COLLECTION_NAME]
        result = collection.find_one({"name": name})
        if result is not None:
            credentials = result.get("credentials", {})
            return credentials.get("username", ""), credentials.get("password", "")
        else:
            return "", ""

```

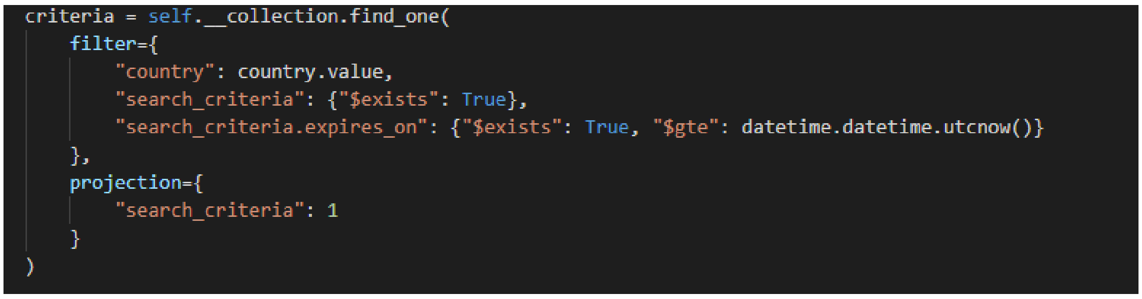
Figure 18. Function get_credentials.

4.2 Creditsafe folder

The second folder is the creditsafe folder. This folder contains 3 main files: country.py, creditsafe_cache.py, and creditsafe_client.py.

In country.py, class Country contains a list of countries alongside its alpha-2 country code list. The alpha-2 country code is of Enum type and returns itself.

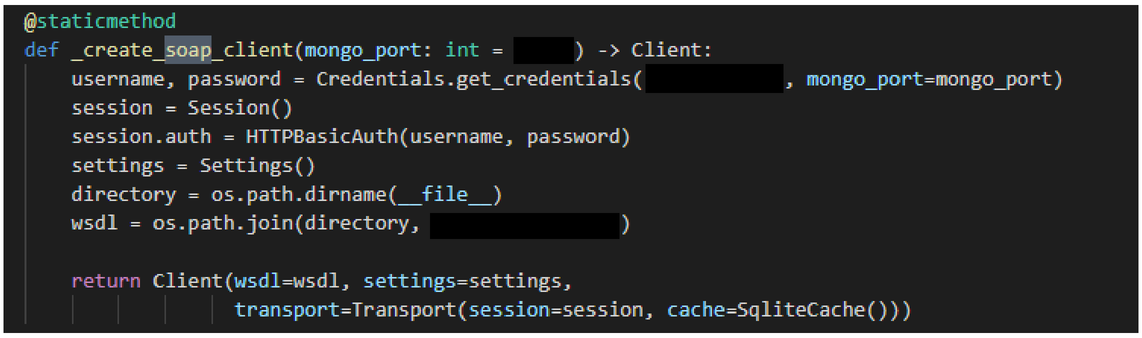
In creditesafe_cache.py, there are 3 static methods: _remove_empty_values, save_seach_criteria and get_search_criteria. These 3 functions will help the cloud application to manage its cache memory so it could respond faster if a similar request already exists in cache memory as Figure 19.



```python
criteria = self.__collection.find_one(
    filter={
        "country": country.value,
        "search_criteria": {"$exists": True},
        "search_criteria.expires_on": {"$exists": True, "$gte": datetime.datetime.utcnow()}
    },
    projection={
        "search_criteria": 1
    }
)
```

Figure 19. Search criteria.

In file creditsafe_client.py, the header, methods, and functions for request and response are defined. This cloud application uses Simple Object Access Protocol (SOAP) for exchanging information. SOAP is a protocol for exchanging communication between machines as Figure 20.



```python
@staticmethod
def _create_soap_client(mongo_port: int = [redacted]) -> Client:
    username, password = Credentials.get_credentials([redacted], mongo_port=mongo_port)
    session = Session()
    session.auth = HTTPBasicAuth(username, password)
    settings = Settings()
    directory = os.path.dirname(__file__)
    wsdl = os.path.join(directory, [redacted])

    return Client(wsdl=wsdl, settings=settings,
                  transport=Transport(session=session, cache=SqliteCache()))
```

Figure 20. Function _create_soap_client.

4.3 Hexillion folder

The third folder is the hexillion folder. This folder mainly contains functions related to the whois API provided by Hexillion.

4.3.1 What is whois API?

Whois API is an endpoint to access the whois server which will receive a domain or IP address as input. Then whois server will return “consistent, automation-friendly results” in XML or JSON format.

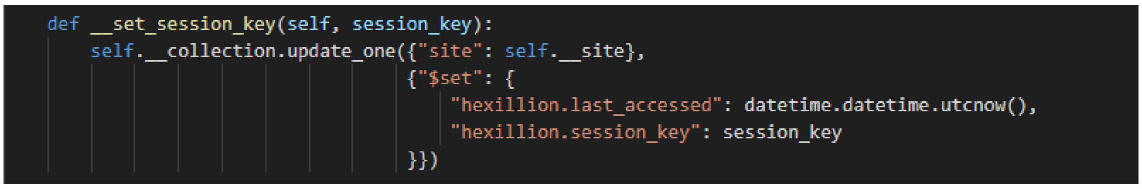
Key features of whois API includes:

- Pulling records from the root server in real-time so the results would be as new as possible. There will be cache memory for improving speed but it can be changed as needed.
- Adding an extra layer to normalize data and return an automation-friendly result.
- Keeping users from encountering query limit issues.
- Works with a wide range of network specifications and configurations.
- Queries are confidential and HTTPS are available.

4.3.2 What does this folder do?

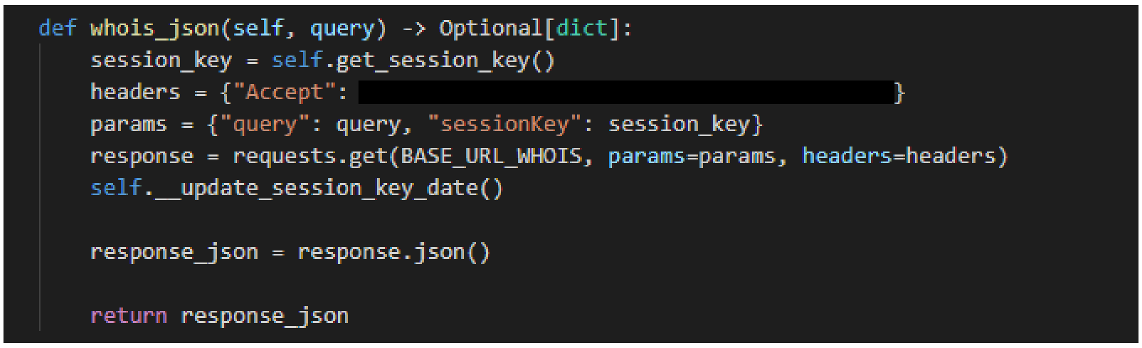
The whole folder is will import the Hexillion library that is needed in this cloud application. Here in the folder, all the issues below are handled:

- Authentication
- Session as Figure 21
- Exceptions
- Parsers as Figure 22
- Returning results



```
def __set_session_key(self, session_key):
    self.__collection.update_one({"site": self.__site},
                                 {"$set": {
                                     "hexillion.last_accessed": datetime.datetime.utcnow(),
                                     "hexillion.session_key": session_key
                                 }})
```

Figure 21. Function _set_session_key.



```
def whois_json(self, query) -> Optional[dict]:
    session_key = self.get_session_key()
    headers = {"Accept": [redacted]}
    params = {"query": query, "sessionKey": session_key}
    response = requests.get(BASE_URL_WHOIS, params=params, headers=headers)
    self.__update_session_key_date()

    response_json = response.json()

    return response_json
```

Figure 22. Function whois_json.

4.4 Root folders

On the root folder level, there are files like asiakastieto.py, authenticate.py, creditsafe_api.py that included in the main function to be run when run is invoked.

On the asiakastieto.py, we have 3 functions:

- The first function is “etree_to_dict”, its responsibility is making dictionary from XML element tree and stripping the namespace out of key names
- The second function is “fetch_company”, its responsibility is fetching company information from Asiakastieto Oy
- The final function is the “main” function that will handle the main logic of the cloud application.

5 CONCLUSION

This thesis has introduced microservices architecture and topologies alongside RESTFUL API, containers, and container orchestration. A part of a cloud application with a microservices structure was built as a result which showcases the specific background and conditions where microservices are beneficial. Furthermore, testing the API using curl or Postman was looked into.

When adding more functionalities in the future, the specific functionality can be created and tested separately in the development environment without the need to modify other functional modules. If a functionality returns errors, the API should return an error message and an error code. However, other functionality modules should still work normally. This will reduce the risk of having downtime for the whole application in the production environment.

The cloud application also has defining features of the microservices architecture. This cloud application is easy to make changes to and introduces new functionalities without affecting a huge part of the code. Additionally, business functionalities will impact the whole landscape of the application. The architecture of this application is decentralized so each of the functionalities will have control over the data that is related to that functionality. This trait will make the application flexible and eliminate the impact of schema changes. The microservices architecture also fits well within the concept of DevOps which will help developers to deploy and make the application running faster. The application in this thesis was written in Python. However, it can communicate with other services through standard channels, hence other services can be written in any coding languages or any technology stacks can be used. For security purposes, the logic behind the application interface is hidden hence increasing its security.

When the response is sent back to the cloud application, this response then will need to be sent to a database of records system in which this information is visible and associated with an entity. This would be the path to develop this project forward. Furthermore, error handling could be improved so that it would be able to return an understandable message to users who do not have a technical background. .

Once the cloud application is able to send account data to the database of records platform, the sales team can see additional account information from Asiakastieto. This will help the sales team to approach potential customers and prepare offers. When adding more functionalities in the future, security and error handling will be taken into accounts during development.

REFERENCES

[1] Richards, M. (2015). Software Architecture Patterns. Retrieved 14 February 2021, from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch04.html

[2] Nemer, J. (2019). Advantages and Disadvantages of Microservices Architecture. Retrieved 21 February 2021, from https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

[3] What is a Container? | Docker. (2020). Retrieved 14 February 2021, from https://www.docker.com/resources/what-container

[4] Docker Documentation. (2019). Retrieved 14 February 2021, from https://docs.docker.com/

[5] Docker Hub. (2020). Retrieved 14 February 2021, from https://hub.docker.com/

[6] Kubernetes Architecture | Aqua. (2018). Retrieved 14 February 2021, from https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-architecture/

[7] Liew, Z. (2018). Understanding And Using REST APIs — Smashing Magazine. Retrieved 21 February 2021, from https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/

[8] Redka, V. (2020). A Beginner's Tutorial for Understanding RESTful API | MLSDev. Retrieved 14 February 2021, from https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api

[9] Chan, J., Chung, R., & Huang, J. (2019). Python API development fundamentals (pp. 2-11). Birmingham: Packt Publishing Ltd.