Documentation Assignment 1 – Airport Security System

VU Cloud Computing, Faculty of Computer Science, University of Vienna

Written by: Jovana Krtolica
Matrikel Number: a11946909

In this documentation, you can read more about the solution I have provided for the first assignment regarding Airport Security System. The documentation contains detailed information in regards the three microservices for Collector, Human Detection and CPanel, explanation regarding the Docker files and docker-compose file as well as it addresses the topics like scalability, bottlenecks, service discovery, communication flow, challenges and problems which occurred during the process of implementation. **The Assignment1 is located in the Git repository under a1146909->Services.**

1. Technology Stack

First thing I would like to discuss is regarding the technology stack which I have chosen to use for the implementation of the services. The technology stack which I have used is Java version 11 as a programming language and Spring Boot as a framework for Java. The choice for this technology stack was due to previous knowledge in Java Spring Boot and because I felt most comfortable with it. The three services are implemented as three separate applications and the project names for the services are the following: collector-microservice for the Collector service, cpanel-service for the Cpanel service and human-detection for the Human Detection. Additionally, I have used Maven, where I have added the required dependencies for the services.

2. Explanation of the structure of the services

Each of the three services is created in a layered structure. The layered structure consists of models, services and controllers. In the model folders, I have created the required domain models which I then used as request body in certain HTTP requests. In the service folders, I have created the functions and methods which contain the logic behind the functionalities which were required. Finally in the controller folders, the controllers with the HTTP requests for certain request methods are to be found. Because I was using Spring Boot, I used some of the predefined annotations available like @Controller or @Service to state the function of the part. The application also contains runnable .jar files which when executed can run the application without the use of an IDE. The location of these files can be found under the following path:

- **collector-microservice/target/collector-microservice-0.0.1-SNAPSHOT.jar**
- **cpanel-service/target/cpanel-service-0.0.1-SNAPSHOT.jar**
- **human-detection/target/human-detection-0.0.1-SNAPSHOT.jar**

These jar files can be created through maven with first executing **mvn clean** command followed by **mvn package** command. First the clean command needs to be executed and then the package command, which will generate the executable .jar file. To run **locally** these jar files, the following command needs to be executed when entered in the root folder of the service, for each service jar:

- **java -jar {NAME_OF_THE_REQUIRED_JAR}.jar**

The server port can be found under the following path for each of the services:

- **collector-microservice/src/main/resources/application-properties**
- **cpanel-service/src/main/resources/application-properties**
- **human-detection/src/main/resources/application-properties**

3. Detailed explanation of functionality of the services

As explained in the previous section, each service is a developed as a separate application in order to achieve the microservices architecture. In this section, I will give a detail explanation of each service and the functionalities it provides through its methos.

- ***Human Detection Service*:**

  The Human Detection service is the service which received a frame through a POST request and sends it for object detection to Google Cloud Vision REST API. When a response is returned, based on the number of people detected, the frame request body is being updated with the person count value.
  In my design of this service, I am creating three folders – models, services and controllers. In the models folder I have five classes. The class Frame represents the request body which is received through a POST request and contains the required fields like timestamp, image, section, event, destination, extra-info and later person-count. Because the Google Cloud Vision API accepts another form of a request body, I have created the classes Image, Features, Request and Final Request. In the Request class I add an Image object and a list of Feature objects. To wrap this into a format acceptable for Google Cloud Vision, I add a list of Request objects in the FinalRequest class. This will have the final format of the request body which is accepted by the Google Cloud Vision.
  Continuing to the services folder, the service class HumanDetectionService located there contains only one method called **postFrameToGCV(Frame frame)** which accepts a frame. In this function, I am first creating the request body which will be sent to GCV. Then I create a connection toward the Google Cloud Vision Url which contains my API key. I make a POST request here, add the required request properties and wait for a connection. If the connection is successful and I get a response code 200, I iterate through the response returned from GCV. Here I check line by line, whether I find a line containing a person and if I find such a line, I update the variable person-count with +1. When I finish iterating with the response, I do a check whether the person-count variable is greater than 0. If that is the case, then I start a new connection towards the face-recognition/frame endpoint, where I do a POST request and send the frame towards this service. After this is done, I update the frame request body with the new variable person-count and do a check regarding the destination field. If there is a destination url provided, then I do a POST request towards the destination URL or otherwise I am just returning the updated request body. During the execution of the function, I am returning the updated frame body, response body from GCV and response codes received from each call.

Finally, in the controllers folder I have a HumanDetectionController where I create a request mapping on /human-detection and then create a HTTP POST Request with mapping on "/frame" which takes frame as a request body and calls the function in the service. The mapping for this controller is the following: "/human-detection/frame" and the port number is 31600.

- ***Collector Service:***

  The Collector service represents a service which handles the communication flow in the system. The basic idea of this service is to receive a frame through a POST request and then forward the image to some other service based on my design decisions.
  Based on my design decisions for this service, I implemented only the required methods. Same as in the previous service, this service is divided into three packages – domains, services and controllers. In the domains package, I have created three classes – Frame, KnownPerson and Person, but I only use the Frame class as I create a request body from its object. The Frame class includes all the variables needed for the request body like timestamp, image, section, event, destination and extra-info.
  In the services package, I have the CollectorService service class which does all the implementation for this service. I first initialize the required url's for the image-analysis, section, alerts, face-analysis and human-detection services. In the method createAFrame(Frame frame), the first thing I do is that I send a POST request to the human-detection service with the frame I have received through a request body. This call will call the human-detection service meaning that each time a frame comes to the collector service, the frame will first be forwarded to the human-detection service. After the call is done and it is successful, I check the destination url received in the request body. If the destination url is empty or not existing, then my idea is to call the image-analysis service. I create a POST request toward the image-analysis/frame url where I transfer the frame as a request body. If the request was successful and I receive a response code 200, I read the response body I get from the call, transform it and send it to the section service. Here, I create a POST request to the section/persons url with the response body I have received from image-analysis. The other case scenario is if the destination url contains the url of the face-recognition service. If the frame destination variable contains the url of the face-recognition service, I create a POST request with the frame which I send to face-recognition/frame. When I get a response code 200 that the response was successful, I am traversing through the response body, where I do a check if there are known-person names. If there are, then I do another POST request but in this case to the alert/alerts url with the response I have received from the face-recognition service. If there are no known persons, then I just return the response body received. During the execution of the function, I am returning the frame body, response body from the services and response codes received from each call.
  IMPORTANT: For this service, my design idea was that the destination URL can either be empty/not existing or contain the url of the face-recognition system. So, when testing this service the destination url should be of this choice.
  The final step for this microservice is the controllers package, where the CollectorController class is located. There I create a request mapping on /collector and then create a HTTP POST Request with mapping on "/frame" which takes frame as a request body and calls the function in the

service. The mapping for this controller is the following: “/collector/frame” and the port number is 31700.

- ***CPanel Service:***

CPanel is the third service which needed to be implemented. CPanel works like a mediator, as the goal of the service is to provide the system with some entry points and then to query information from the Section, Alert and Collector services. The port number of this service is 3800.
My design decision for this service is like the design decision for the two previous ones, as the service is structures in three packages for models, services and controllers. However, a difference here is that the services and controllers packages contain three service classes and three controller classes, one for each Alert, Section and Collector requirements. In the models package, I have created the required classes which represent the request bodies which are going to be sent in the methods in the services package. The Frame class contains all the needed variables for the frame request body like timestamp, image, section, event, destination and extra-info. The Person class contains age and gender variables. The SectionPerson class is intended to be a request body for the calls towards the section service and contains the variables timestamp, section, event, image, extra-info and a list of person objects. The KnownPerson and KnownPersonBody classes are classes intended for the alert service. The KnownPerson class has variable name while the KnownPersonBody is the request body which is going to be sent towards the alert service and contains timestamp, section, event, image, extra-info and a list of KnownPerson objects.
The services package contains three service classes called **AlertService, SectionServvice** and **CollectorService** because each service class implements the methods from the required service.
AlertService + Alert Controller: The **AlertService** class contains methods which are representing a mapping to the Alert service API. There are four methods in this service class. The first method is createNewAlert(KnownPersonBody knownPersonBody) which creates a new alert with the required request body. I create a POST request with the request body to the alert/alerts url. If the request is successful and the response code is 200, a response message with the response code is printed, otherwise an unsuccessful message is printed with the response code received. The second method is called getAlertsInAGivenTime(Map<String, String> params) which returns a ResponseBody in a JSON string format back. This function is having a map of arguments, as the user can put either from, to, aggregate or event as query parameters in the call. In this function, I am first building the correct url format based on the map key:value pairs I receive. Then I make a GET request toward the required alert URL. If the response status is a success, I get the response body, iterate through it and return it as a JSON object in a string format. On the other hand, if the method is unsuccessful, I return the response code and a message. The third method is called getAlertsByUuid(String uuid) which returns a ResponseBody in a JSON string format. In this methos, I am making the URL to be alerts/uuid because I get uuid as a PathVariable. Then I create a GET request to that url and if the response status is a success, I iterate through the response and return the response as a JSONObject in a string format. If the response is not a success, I return a message with the received response code. The fourth method is called deleteAlert(String uuid) which deletes an alert based on its uuid received as a PathVariable. Here I create the url and then a DELETE method towards that URL. If the response is successful, I return a success message with the status code otherwise an unsuccessful message with the status code. The AlertController

is mapped on the “/cpanel/alerts” and then each method has its own mapping. The addPerson is mapped on “/cpanel/alerts” and takes a KnownPersonBody as a request body. It is a POST method and called the createNewAlert function from the AlertService. The getAlertsByUuid is mapped on “/cpanel/alerts/{uuid}” and it is a GET request which takes the uuid as a PathVariable. This method calls the getAlertsByUuid function from the AlertService. The getPersonsInATimeFrame is mapped on “/cpanel/alerts” and it is a GET request towards the getAlertsInAGivenTime function in the AlertService. This function received a Map of strings as RequestBody, sends it to the service method which then does the transformation to the URL. The final method in this controller is the deleteAlert which is mapped on “/cpanel/alerts/{uuid}” and it is a DELETE request towards the deleteAlert function in the AlertService, which takes an uuid as a PathVariable.

Section Service + SectionController: The SectionService class has two methods only. The first method is called getPersonsInATimeFrame(Map<String,String> params) which is returning a JSONObject in a string format. This function takes a Map of string key:value pairs to build the required URL based on the fact whether from, to, aggregate or event query parameters would come. This is the first step of the function. Then I create a GET request toward the section service url. If the response is a success, then I receive back the response body, iterate through it and create a JSONObject in a string format, so I can return it. If the response is not a success, then I return an unsuccessful message with the appropriate status code. The second function is called postPerson(SectionPerson sectionPerson) which takes a SectionPerson object as a request body. Here I do a POST request to the required url from the Section service with the request body. If the response is a success, I am returning a success message with the appropriate status code, otherwise I am returning an unsuccessful message also with the appropriate status code. The SectionController is mapped on “/cpanel/section” and has two methods inside. The addPerson is mapped on “/cpanel/section/persons” and takes a SectionPerson as a request body. It makes a POST request towards the postPerson function located in the SectionService. The getPersonsInATimeFrame is mapped on “/cpanel/section/persons” and takes a Map of string key:value pairs as RequestParameters (query params). It makes a GET request towards the getPersonInATimeFrame function located in the SectionService and based on the query params it then builds the required URL.

Collector Service + CollectorControler: The CollectorService has only one method called createFrame(Frame frame). In this method I am using the correct URL od the collector and creating a POST request, with the Frame object as a request body. If the response status is a success, then I return a success message with the received response code, otherwise I return an unsuccessful message with the received response code. The CollectorContoller is mapped on “/cpanel/collector” and has one method inside. The createFrame is mapped on “/cpanel/collector/frame” and takes a Frame as a request body. It makes a POST request towards the createFrame function located in the CollectorService.

4. Port numbers for each service plus examples of requests:

In the following section, I am going to post the calls which I have tested through Postman to make sure that the system is working properly. To test the services, the services must be up and running.

Because the images in base-64 are large strings, please add in the field for images an image so it would not take to much space in the document.
Camera ports - 31001-31003; Image-analysis port - 31200; Face-recognition port - 31300; Section port - 31400; Alert port - 31500; Human-detection port - 31600; Collector port - 31700; Cpanel port - 3800;

Human-Detection calls executed through Postman (port 31600):

- **POST request URL -** ***http://localhost:31600/human-detection/frame***
- **RequestBody (optional extra-info and destination field)**
  ```
  {
      "timestamp": "2021-10-29T19:32:18",
      "section": -1,
      "event": "entry",
      "image": "<base-64-image>"
  }
  ```
- **ResponseBody -** Either a response body as JSONObject and/or response status codes.

Collector calls executed through Postman (port 31700):

- **POST request URL -** ***http://localhost:31700/collector/frame***
- **RequestBody -**
  ```
  {
     "timestamp":"2010-10-14T11:19:18",
     "image":"<base64-image>",
     "section":1,
     "event":"entry",
     "extra-info":""
  }
  ```
- **ResponseBody -** Either a response body as JSONObject and/or response status codes.

CPanel calls executed through Postman (port 3800):

- **Alert API:**
  - **POST request URL -** ***http://localhost:3800/cpanel/alerts***
  - **RequestBody -**
    ```
    {
     "timestamp": "2010-10-14T11:19:27",
     "section": 1,
     "event": "exit",
     "known-persons":[
     {
     "name":"Ms. X"
     }
     ],
     "image": "<base64-image>",
     "extra-info": ""
    }
    ```
  - **ResponseBody -** Either a response body as JSONObject and/or response status **codes**

        - **GET request URL -** ***http://localhost:3800/cpanel/alerts?from={from}&to={to}&aggregate=count&event=entry***
        - **RequestBody -** NO, only Query Parameters.
        - **ResponseBody -** A response body as JSONObject and response status codes.
        - **GET request URL -** ***http://localhost:3800/cpanel/alerts/{uuid}***
        - **RequestBody -** NO, only Path Variable uuid.
        - **ResponseBody -** A response body as JSONObject and response status codes**.**
        - **DELETE request URL -** ***http://localhost:3800/cpanel/alerts/{uuid}***
        - **RequestBody -** NO, only Path Variable uuid.
        - **ResponseBody -** Response status code.
- **Section API:**
    - **POST request URL -** ***http://localhost:3800/cpanel/section/persons***
    - **RequestBody -**

```
{
   "timestamp":"2010-10-14T11:19:21",
   "section":1,
   "event":"exit",
   "persons":[
      {
         "age":"8-12",
         "gender":"male"
      }
   ],
   "image":"<base64-encoded-string>",
   "extra-info":""
}
```

    - **ResponseBody -** Either a response body as JSONObject and/or response status codes.
    - **GET request URL -** ***http://localhost:3800/cpanel/section/persons?from={from}&to={to}&aggregate=count&event={event}***
    - **RequestBody -** NO, only Query Parameters.
    - **ResponseBody -** A response body as JSONObject and response status codes.
- **Collector API:**
    - **POST request URL -** ***http://localhost:3800/cpanel/collector/frame***
    - **RequestBody -**

```
{
    "timestamp": "2021-10-29T19:32:18",
    "section": 1,
    "event": "exit",
    "image": "<base64-image>"
}
```

    - **ResponseBody -** Either a response body as JSONObject and/or response status codes.
- **Camera stream flow call (ports 31100-31103):**

- **POST request URL -** ***http://localhost:31100/stream?toggle=on***
- **RequestBody -**

```
{
 "destination": "http://cpanel:3800/cpanel/collector/frame",
 "max-frames": 4,
 "delay": 0.1,
 "extra-info": ""
}
```

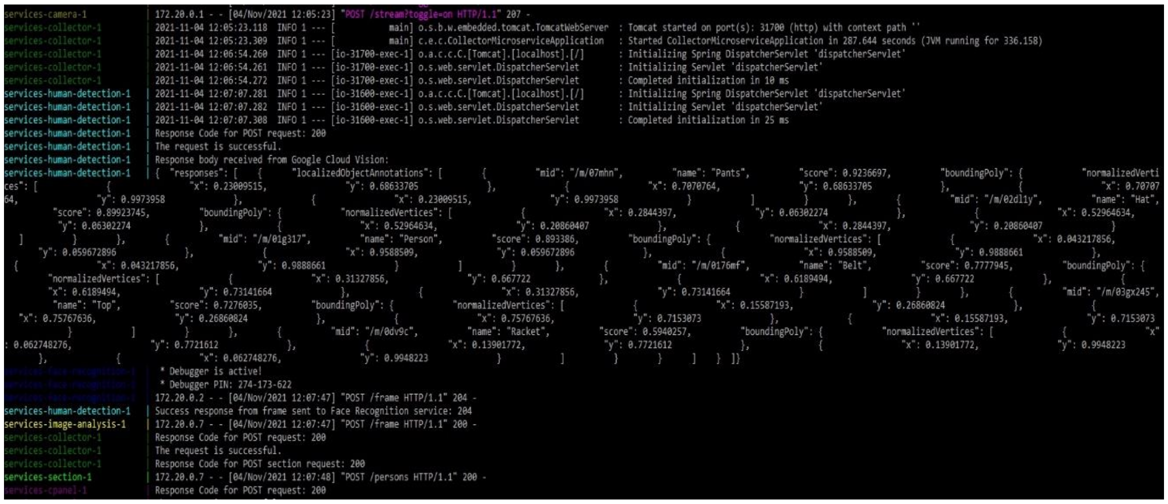
- **QueryParams - toggle=on**
- **ResponseBody -**

```
{
    "code": 200,
     "type": "STATUS",
     "message": "Streamed 4 frame(s) in 22.045551538467407 seconds, with 0 fa
iled requests."
}
```

5. Data/Communication Flow:

For the data and communication flow, I have made some design decisions which I think would fit the solution I have provided through my code. First step of starting the communication flow would be that the camera service is sending images and frames to the cpanel/collector/frame. From there the communication flow tree starts. In my solution, when the frame reaches the cpanel/collector/frame, the cpanel service as an intermediator calls the collector service with a POST request with the frame as a request body. Then the collector starts working its way. When the frame reaches the collector, the collector is sending the frame to the human-detection/frame. After sending the frame to the human-detection service, the collector service checks the destination URL of the frame. If the destination URL is empty or not existing, the collector service will send the frame through a POST request to the image-analysis/frame service. The image-analysis service would then return a certain response back to the collector service and that response from image-analysis service is sent through a POST request to the section service which then returns a certain response code. The section service communicates with the cpanel service as through the cpanel service we can reach or create something to the section service. The second case scenario here would be that the destination URL in the frame sent to the collector contains the URL of the face-recognition service. If this is the case, then the frame received to the collector is sent to the face-recognition/frame service. After this call, the face-recognition sends back a response and if there are known-persons included in the response, then the response is sent to the alert/alerts service. Otherwise, if there are no known-persons received in the response from the face-recognition, the response stays in the face-recognition system. Just the same as the section service, the alert service communicates with the cpanel service through cpanel/alerts, so the cpanel can reach the alert service to create or retrieve something from it. The final communication flow happens when a frame reaches the human-detection service. When a frame reaches the human-detection/frame, then a new request body is created with information from that frame to be sent to the Google Cloud Vision. From the response received from Google Cloud

Vision, there is a check whether there is a person found to update the new person-count variable. If the value for the person-count is greater than zero, then the communication moves further by creating a POST request with the frame towards the face-recognition/service. However, if there is a destination URL provided, then the request with the updated frame is sent to the destination URL. Finally, if the destination URL is provided, then a response code is returned or otherwise the whole updated frame is returned as a response. In the picture below, the cycle flow can be seen when the camera starts streaming frames towards /cpanel/collector/frame :



```
services-camera-1              | 172.20.0.1 - - [04/Nov/2021 12:05:23] "POST /stream?toggle=on HTTP/1.1" 207 -
services-collector-1           | 2021-11-04 12:05:23.118  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 31700 (http) with context path ''
services-collector-1           | 2021-11-04 12:05:23.309  INFO 1 --- [           main] c.e.c.CollectorMicroserviceApplication   : Started CollectorMicroserviceApplication in 287.644 seconds (JVM running for 336.158)
services-collector-1           | 2021-11-04 12:06:54.260  INFO 1 --- [io-31700-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
services-collector-1           | 2021-11-04 12:06:54.261  INFO 1 --- [io-31700-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
services-collector-1           | 2021-11-04 12:06:54.272  INFO 1 --- [io-31700-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 10 ms
services-human-detection-1     | 2021-11-04 12:07:07.281  INFO 1 --- [io-31600-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
services-human-detection-1     | 2021-11-04 12:07:07.282  INFO 1 --- [io-31600-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
services-human-detection-1     | 2021-11-04 12:07:07.308  INFO 1 --- [io-31600-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 25 ms
services-human-detection-1     | Response Code for POST request: 200
services-human-detection-1     | The request is successful.
services-human-detection-1     | Response body received from Google Cloud Vision:
services-human-detection-1     | {  "responses": [    {      "localizedObjectAnnotations": [        {          "mid": "/m/07mhn",          "name": "Pants",          "score": 0.9236697,          "boundingPoly": {            "normalizedVerti
ces": [              {                "x": 0.23009515,                "y": 0.68633705              },              {                "x": 0.7070764,                "y": 0.68633705              },              {                "x": 0.70707
64,                "y": 0.9973958              },              {                "x": 0.23009515,                "y": 0.9973958              }            ]          }        },        {          "mid": "/m/02dl1y",          "name": "Hat",
          "score": 0.89923745,          "boundingPoly": {            "normalizedVertices": [              {                "x": 0.2844397,                "y": 0.06302274              },              {                "x": 0.52964634,
                "y": 0.06302274              },              {                "x": 0.52964634,                "y": 0.20860407              },              {                "x": 0.2844397,                "y": 0.20860407              }
            ]          }        },        {          "mid": "/m/01g317",          "name": "Person",          "score": 0.893386,          "boundingPoly": {            "normalizedVertices": [              {                "x": 0.043217856,
                "y": 0.059672896              },              {                "x": 0.9588509,                "y": 0.059672896              },              {                "x": 0.9588509,                "y": 0.9888661              },
              {                "x": 0.043217856,                "y": 0.9888661              }            ]          }        },        {          "mid": "/m/0176mf",          "name": "Belt",          "score": 0.7777945,          "boundingPoly": {
            "normalizedVertices": [              {                "x": 0.31327856,                "y": 0.667722              },              {                "x": 0.6189494,                "y": 0.667722              },              {
                "x": 0.6189494,                "y": 0.73141664              },              {                "x": 0.31327856,                "y": 0.73141664              }            ]          }        },        {          "mid": "/m/03gx245",
          "name": "Top",          "score": 0.7276035,          "boundingPoly": {            "normalizedVertices": [              {                "x": 0.15587193,                "y": 0.26860824              },              {
                "x": 0.75767636,                "y": 0.26860824              },              {                "x": 0.75767636,                "y": 0.7153073              },              {                "x": 0.15587193,                "y": 0.7153073
              }            ]          }        },        {          "mid": "/m/0dv9c",          "name": "Racket",          "score": 0.5940257,          "boundingPoly": {            "normalizedVertices": [              {                "x"
: 0.062748276,                "y": 0.7721612              },              {                "x": 0.13901772,                "y": 0.7721612              },              {                "x": 0.13901772,                "y": 0.9948223
              },              {                "x": 0.062748276,                "y": 0.9948223              }            ]          }        }      ]    }  ]}
services-face-recognition-1    |  * Debugger is active!
services-face-recognition-1    |  * Debugger PIN: 274-173-622
services-face-recognition-1    | 172.20.0.2 - - [04/Nov/2021 12:07:47] "POST /frame HTTP/1.1" 204 -
services-human-detection-1     | Success response from frame sent to Face Recognition service: 204
services-image-analysis-1      | 172.20.0.7 - - [04/Nov/2021 12:07:47] "POST /frame HTTP/1.1" 200 -
services-collector-1           | Response Code for POST request: 200
services-collector-1           | The request is successful.
services-collector-1           | Response Code for POST section request: 200
services-section-1             | 172.20.0.7 - - [04/Nov/2021 12:07:48] "POST /persons HTTP/1.1" 200 -
services-cpanel-1              | Response Code for POST request: 200
```

FLOW of one cycle through Postman:

- POST request on the ***http://localhost:31100/stream?toggle=on*** where the destination of the sending of the frames would be towards the destination URL with value ***http://cpanel:3800/cpanel/collector/frame*** with the following request body:

```
{
 "destination": "http://cpanel:3800/cpanel/collector/frame",
 "max-frames": 10,
 "delay": 0.1,
 "extra-info": ""
}
```

- This will call the collector service which would then call the human-detection service on the URL ***http://human-detection:31600/frame*** and it will send the first frame of the number of frames provided.
- The human-detection service would then call the Google Cloud Vision API through the following URL
- When a response is returned from the Google Cloud Vision, depending on the URL of the destination variable in the frame, two services could be called.
- First scenario would be if there is no destination field, the image-analysis service would be called on the URL ***http://image-analysis/frame***
- The response from the image-analysis service would then be sent towards the section service on the URL ***http://section/persons*** and would be forwarded back to the cpanel service.

- The second scenario would be if the destination URL contains the face-recognition service URL. If that is the case, the response would be sent toward the ***http://face-recognition/frame*** URL.
- When the response is received form the face-recognition service, a check is done to see if there are known persons then the response is sent towards the ***http://alert/alerts*** URL. Finally, the response from the alert service is returned to the cpanel and another cycle with a new frame starts.

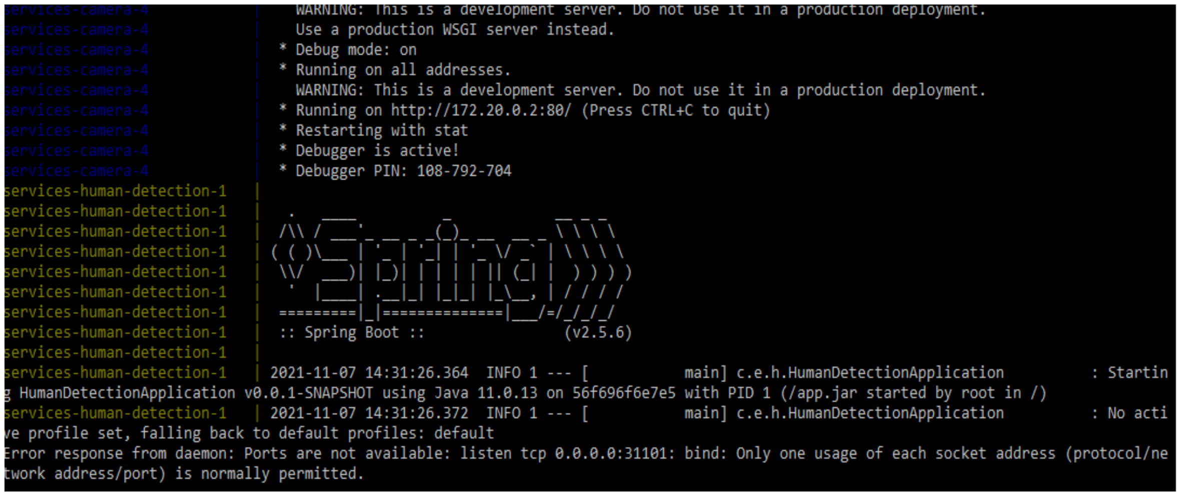
6. Scalability:

The first thing I have noticed during the executing of one cycle starting from camera to send frames to the cpanel/collector/frame is that it takes some time for the collector and the human-detection service to be up. A second observation would be that it took me around 3-5 minutes for a cycle with 10 frames to be executed fully, which means that for testing with 100 frames, the system would take some time to stream the data. However, I have received different results when I added a higher delay between them and then the system took more time to finish iterating each frame circle. When the delay was brought to zero it was quite fast. Regarding most computationally intensive services, I would say that human-detection and controller are the most intensive service as they include most of the processing of the stream and many calls are made here.
With regards to restricting some services the CPU and memory usage, I have observed and reached interesting results. My idea is to restrict two services human-detection and collector services as they are the most consuming. I introduced the following restrictions to both services:

```
limits:
  cpus: '0.001'
  memory: 50M
 reservations:
  cpus: '0.0001'
  memory: 20M
```

When running the containers, the human-detection container started without any issue. However, the collector container started and then exited which means that the consumption I created was very low, as in my design this is the number one service in consumption of CPU power and memory. With using such restrictions, then other services were failing. So, I used other restrictions to the two services, where I restricted the collector to **cpus:"1" and memory: 512M** whereas the human-detection to **cpus:"1" and memory: 512M** also. All the other services are restricted to **cpu:"1" and to memory: 256M.** Then I did a streaming of 20 frames, which was executed in approximately 2 minutes, proving that the stream is way faster.
Then I have created ports range first for the camera and later the image-analysis and face-recognition services each having a range of four ports. Under deploy, I have created 3 replicas of each of each service. Here I have observed some strange behavior when I wanted to scale the camera service. Even though my docker-compose file states that I have version 3, when I execute the command docker-compose --version I see that the version is v2.0.0. I deploy three replicas from the camera service. Additionally, I try to scale the services through executing docker-compose --scale camera=3 command. This means that when I run the containers and in Docker

there appear three instantiations of the same service, for maybe two of the services it would pass but then for one of the services I get the error shown in the picture below.



```
services-camera-4              |     WARNING: This is a development server. Do not use it in a production deployment.
services-camera-4              |     Use a production WSGI server instead.
services-camera-4              |   * Debug mode: on
services-camera-4              |   * Running on all addresses.
services-camera-4              |     WARNING: This is a development server. Do not use it in a production deployment.
services-camera-4              |   * Running on http://172.20.0.2:80/ (Press CTRL+C to quit)
services-camera-4              |   * Restarting with stat
services-camera-4              |   * Debugger is active!
services-camera-4              |   * Debugger PIN: 108-792-704
services-human-detection-1   |
services-human-detection-1   |    .   ____          _            __ _ _
services-human-detection-1   |   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
services-human-detection-1   |  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
services-human-detection-1   |   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
services-human-detection-1   |    '  |____| .__|_| |_|_| |_\__, | / / / /
services-human-detection-1   |   =========|_|==============|___/=/_/_/_/
services-human-detection-1   |   :: Spring Boot ::                (v2.5.6)
services-human-detection-1   |
services-human-detection-1   | 2021-11-07 14:31:26.364  INFO 1 --- [           main] c.e.h.HumanDetectionApplication          : Startin
g HumanDetectionApplication v0.0.1-SNAPSHOT using Java 11.0.13 on 56f696f6e7e5 with PID 1 (/app.jar started by root in /)
services-human-detection-1   | 2021-11-07 14:31:26.372  INFO 1 --- [           main] c.e.h.HumanDetectionApplication          : No acti
ve profile set, falling back to default profiles: default
Error response from daemon: Ports are not available: listen tcp 0.0.0.0:31101: bind: Only one usage of each socket address (protocol/ne
twork address/port) is normally permitted.
```

I get port conflicts and I need to the run the docker-compose file. This is because the containers are associated with random ports at the same time. I have read that this error persists on a Windows but is not existing on a Linux machine. Additionally, I have put constraints on services regarding dependencies. Services depend on other services and would not be available until the services they depend on will be up and running.

7. Service Discovery:

In each of the three implemented services by myself contains a Docker file in its root folder. The Dockerfile for each service contains information regarding the Java 11 jdk used, adding of the curl command to have easier debugging when entering each contains, exposing of the post and the .jar executable file which could be found under the target folder in the root directory of the services. After developing the Dockerfiles for the three services, I have created a docker-compose.yml file where I have added all the services and divided them into two networks as required. In the backend_net network I have all the services except the camera service, while the camera service lies in the camera network. Each service contains information about the image name, the specified port for external and internal use, the network the service is part of as well as the services it depends on. In the cpanel, collector and human-detection services, I have additionally added the build with the context and the location of the dockerfile in the project directory. After I have finished the docker-compose.yml file, I have run the following commands to build the images and run the containers after.

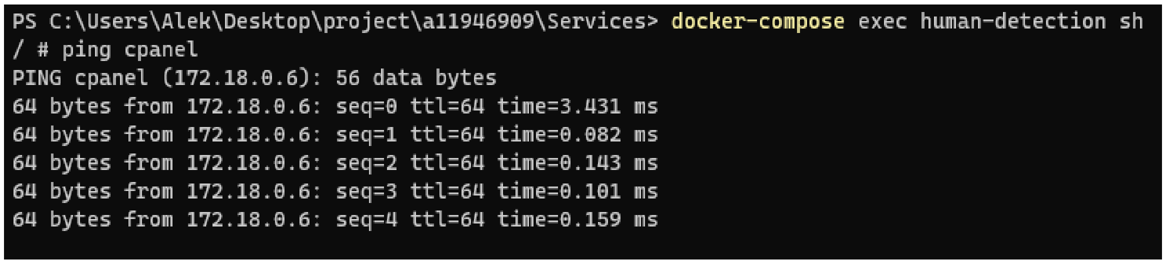
- **docker-compose build --no-cache**
- **docker-compose up**

To be sure that my services are discovered between themselves, I checked first by entering each of the services and checking through commands whether they are discoverable. Because the services are located within a network, Docker offers the load balancer functions and capacities, which manages the flow between the servers and clients. When entering a specific container, let's take an example for the human-detection service, I have executed the following command to enter inside it:

- **docker-compose exec human-detection sh**

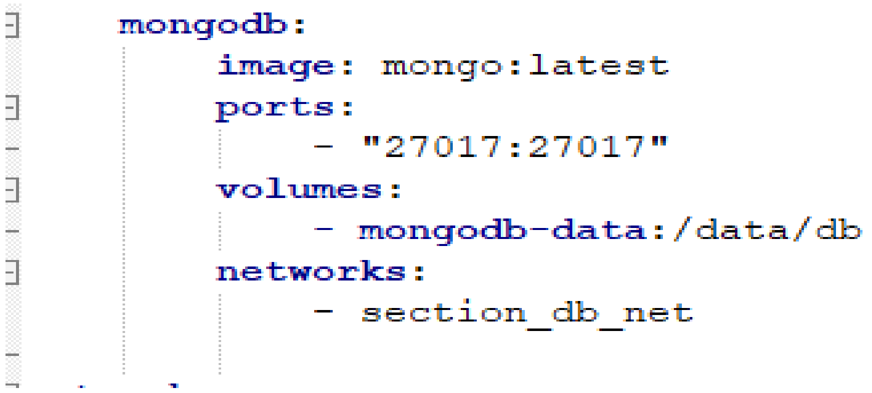
After entering in the human-detection container, the next command to be executed is the ping command to a certain other service. The ping would be successful if we receive an IP address from the service we want to ping as well as some data transfers in bytes.

- **ping {service-name}**



```
PS C:\Users\Alek\Desktop\project\a11946909\Services> docker-compose exec human-detection sh
/ # ping cpanel
PING cpanel (172.18.0.6): 56 data bytes
64 bytes from 172.18.0.6: seq=0 ttl=64 time=3.431 ms
64 bytes from 172.18.0.6: seq=1 ttl=64 time=0.082 ms
64 bytes from 172.18.0.6: seq=2 ttl=64 time=0.143 ms
64 bytes from 172.18.0.6: seq=3 ttl=64 time=0.101 ms
64 bytes from 172.18.0.6: seq=4 ttl=64 time=0.159 ms
```

For the database configuration for the section, I have created a simple service in the docker-compose.yml file for MongoDB. This database represents a simple microservice for one database for all the section services. Here I have set the environment variables for the database, as well as the port on which the database is running. I have created a separate network for the database called section_db_net. The network is added also to the section service and the section service depends on the mongodb service because if this were not the case, problems would arise if the section service were up before the mongodb service. The simple service from the docker-compose file can be seen in the picture below.



```
    mongodb:
        image: mongo:latest
        ports:
            - "27017:27017"
        volumes:
            - mongodb-data:/data/db
        networks:
            - section_db_net
```

8. Bottlenecks:

After observing the system for some time, I have seen some of the bottlenecks which could happen in the system. The first is that due to the constraints in my docker-compose file, certain services need to be up for other services to be up. This may sometimes cause problems as calls would not be executed if the dependent service fails.

Another observation is regarding the API key for the Google Cloud Vision. Since I am hardcoding the key in my code for this exercise, I am putting the system to a bit of security vulnerability. This is not the best practice, as the key should be at least hashed or passed as an argument from the user. Additionally, if the API key changes than the key in the code will need to change, as the calls to the Google Cloud Vision would not work.

9. Problems, challenges and improvement:

During the implementation of the assignment, I stumbled upon a couple of problems and challenges. The first and the biggest challenge I had was with the installation of Docker on a Windows machine, as first my machine did not send any updates and the build version of the OS was older than the one required from Docker. Here, I have lost a couple of days I would say until making my environment work. Regarding the code implementation I did not stumble onto major challenger. However, the challenges happened when I was writing the Dockerfiles for each of the three services. I was using Java Spring Boot as a development stack and most of the tutorials showed different ways of creating the Dockerfiles. When the Dockerfiles were created, then I had issue with the docker-compose.yml files as I always received an internal server error with status code 500. I spent four days figuring out what was the mistake and then I realized I have made two. The first one was that the URLs were wrongly written and the second was that I used another variable for the connection to the human-detection service instead of the one I needed.
Another issue I had was that I was not able from start to build the replicas of a certain service. This issue persists on Windows machines and I should rerun the docker-compose file a couple of times in order to be sure that all the replicated services are up. When used a pure Linux machine, this issue should not persist.