Report on Deevio Assignment

Yiorgos Marinellis

July 4, 2019

Abstract

This text will try to present a proposed solution to the assignment by analysing the procedure that was followed. Throughout the text several assumptions that led to specific decisions will be described. In the end a basic evaluation of some trade-offs will be presented.

1 Problem

Modern manufacturing processes have a high degree of automation and at the same time high quality requirements. Our machine vision solution helps ensuring these quality requirements are met by providing means for automatic recognition of defects. In addition to the recognition, it is essential to store the data about defects. This is mandatory for us to constantly improve our models. Also, our customers need be able to analyze for example common defect types.

The goal of this task is, to write a service that stores results from a classification service in a database and make them available through a REST API. It is meant to run on the edge, that is, a computing device that is deployed in a factory.

Classification results are given as a JSON file with the following structure. Status can be one of processing, complete, output is an array of predictions.

```
{
    "status":"complete",
    "imagePath":"20180907/1536311270718.jpg",
    "imageId":"1536311270718",
    "output":[
        {
            "bbox":[
                1008.8831787109375,
                280.6226501464844,
                1110.0245361328125,
                380.72021484375

            ],
            "probability":0.9725130796432495,
            "label":"nail",
            "result":"good"

        }
    ]
}
```

The REST API must

- Return a prediction for a given imageId
- Return all "weak" classifications (containing a prediction with a probability <0.7)

The JSON file is given to the service via a message bus as a published message on topic new_prediction

2 Objectives

The developed application is intended to be installed on premise using edge devices in a factory. At first on premise installations need to have an easy deploy mechanism. This is the reason why a containerized solution was used. Next, we do not know the computing capabilities of the edge devices. As a result we have to use a compact solution which will be suitable for the potential cpu/memory boundaries. We have to also take into consideration the storage capabilities of the device and the rate of the incoming classification results.

2.1 Web framework

A Flask is considered a python micro-framework for building web applications. Its stack does not include a wide variety of libraries which for us mean that the application will perform on a low memory overhead. In the user has to implement the functionalities he/she needs. As a general attitude Flask is considered the suitable solution for building micro-services with low usage of resources.

2.2 Database selection

The database to use is always a very crucial decision. There are lots of examples of big companies migrating from a database to another. Some developers believe that there is not a perfect database. It is all a matter of trade-off. The path I usually follow in selecting which database to use is ,in the beginning, to select this one that is more comfortable for your data structure and your queries. Of course the selection is not critical and someone needs to continuously measure the performance and the efficiency of the database.

The predefined json data structure of the classification results in conjunction with the REST API implies using a NoSQL database. The biggest advantage is the minimum transformation effort needed to save, retrieve and serve the data. No object mappings, no joins. The latter in combination with the indexes can give a great performance. In theory an application which uses a NoSQL database should not also care about the database schema, which is true, but in order to ensure compatibility with older versions some sort of validation or mapping should be done. In an SQL database you solve this issue with database migrations. Lastly we use not omit a main disadvantage of NoSQL with is updating a complex document in the database. In contrary to SQL database a complex query is needed to update embedded documents and arrays.

2.3 Publish/subscribe protocol

MQTT is a pub/sub protocol over TCP which is suitable for low data and low energy usage. Built on-top of TCP it is suitable for real-time reliable communication on WAN. It delivers binary data and not any XML format. It can be easily scalable by adding another broker or a subscribed client. The way MQTT works is over a long-lived TCP connection with implies that clients are not put on sleep. Finally with an extra data cost MQTT can use TLS/SSL for data security.

Other popular and tested solutions could use ,with an extra overhead, the expressiveness of the XML based XMPP protocol (see. XEP-0080) or the scalability features of Apache Kafka.

Summarizingly, taking also into account the size of the transferred json, the selected MQTT protocol is an efficient solution which combines low resource consumption with real-time communication.

2.4 Data format updates

Changes to the data format are impossible not to happen. MongoDB schemaless approach lets data to be stored in whatever model in the database. But may MongoDB is tolerant to data model changes the application code and the api is not. As a result some kind of data validation is needed. An approach to this could to maintain a loose jsonschema and check for the presence only of some crucial fields like imageId. This soft validation of the data can help as well to maintain a backwards compatibility approach with previous versions. Also, for every field that changes name, a mapping should be stored to ensure that application code runs correctly.

2.5 Easy deploy

The containerized solution was used in order to make the on premise application deploy a matter of minutes. A more complete solution would be to automatically push the build containers to the edge devices -some use a reverse ssh connection- in order to avoid the cost of physical presence. Though such a process requires ,apart from a reliable internet connection, a huge amount of responsibility and permissions from the factory administration team.

2.6 Monitoring

Having on premise installations mandates the use of monitoring system to ensure that components are always app and running. A variety of solutions exist here with Prometheus to the latest trend. A dockerized environment of prometheus, grafana, node-exporter and cadvisor was built in separate branch, `prometheus`.

3 Documentation

3.1 Dependencies

- python-3.5
- Flask-1.0.3
- pymongo-3.8.0
- paho-mqtt-1.4.0
- flake8
- pytest
- coverage

3.2 Setup

From the root directory use the build script to build the application docker image

```
$ ./build <TAG>
```

Mongodb database and Mosquitto MQTT broker services are containerized and initialized through the `docker-compose` tool. Moreover an simple HTTP interface for mongodb was installed for administration purposes. To build and run the multi-container docker application run

```
$ docker-compose up -d
```

You can investigate application logs and output from flake, pytest and coverage using

```
$ docker-compose logs -f app
```

The current set-up firstly warms the mongo database with 150 dummy classification documents, then runs an mqtt client which publish classification result under the topic *new_prediction* and the run the application. For the purposes of this challenge the default embedded flask web server was used which is not suitable for production purposes.
You can access the application on

```
http://127.0.0.1:5000
```

To stop the application execute on the current directory

```
$ docker-compose kill
```

To remove all stopped containers execute

```
$ docker-compose down -v
```

If a more clean deploy is needed with a cold database and not an mqtt publisher please run

```
$ docker-compose -f docker-compose-prod.yml up -d
```

3.3 CLI client

The CLI client was implemented to help on the development and test process of the application. Its main functionalities include a simple implementation of a mongodb and an mqtt client. The mongodb client was used to populate the empty db in order to manually test the implementation of the api calls and draftily evaluate the efficiency of the set-up. The mqtt client emulates the way classification results are forwarded to the application and was implemented in order to verify the connectivity of the publish/subscribe feature of the application.
You can find the implementation of the cli client under `project/cli_utils`.

Draft Evaluation