Poromagia

Final project

Internet of Things (IoT) Project TX00CI65-3010

Metropolia University of Applied Sciences

Bachelor's Degree

Information Technology

Manish Subedi, Ahmed Al-Tuwaijari, Melany Macias, Sophia Schwalb.

23.12.2022

Contents

1 Introduction

The aim of the project is to produce a card sorter for Poromagia. It is a Finnish company that distributes collectible cards and other products related to gaming. The monetary value of the Magic the Gathering collectible cards that the company works with varies from a few cents per card to thousands of euros, and Poromagia's warehouse possesses thousands of euros in cards that need to be sorted by their employees on daily basis.

Therefore, the purpose of the machine is to facilitate automated sorting for Poromagia's collection cards to save significant amounts of time, avoid human labour, reduce financial expenses, and decrease sorting errors.

1.1 Project description

The project is composed of two fundamental parts that complement each other and should work correctly to achieve the correct implementation of the project. The part of the project that has been approached here comprises the development of the software and its implementation in the sorting machine.

The software includes several parts that can be generally described as a computer vision algorithm that recognizes which card it is processing, a web interface that contains sorting categories, and a moving plane that takes the respective card to its correct category.

1.2 Structure of this documentation

The software includes several parts that have been developed independently, but that communicate with each other in different ways that will be described below. In this documentation, we will first address the conception of the project, and the detailed description of how we approached the creation of the technical architecture of the project, as well as the demonstration of the machine that was created by students of mechanical engineering.

In the section about the current state, it is described how the project was implemented and each of the parts that compose it: from the intercommunication between the microcontrollers used, the computer vision algorithm, to the development of the web page, being this the most extensive section.

Finally, the issues faced during the development of the project are addressed, as well as the improvements that can be implemented to improve the final product, since due to the time limit (one student period) and the size of the project, there are aspects where there is room for improvement.

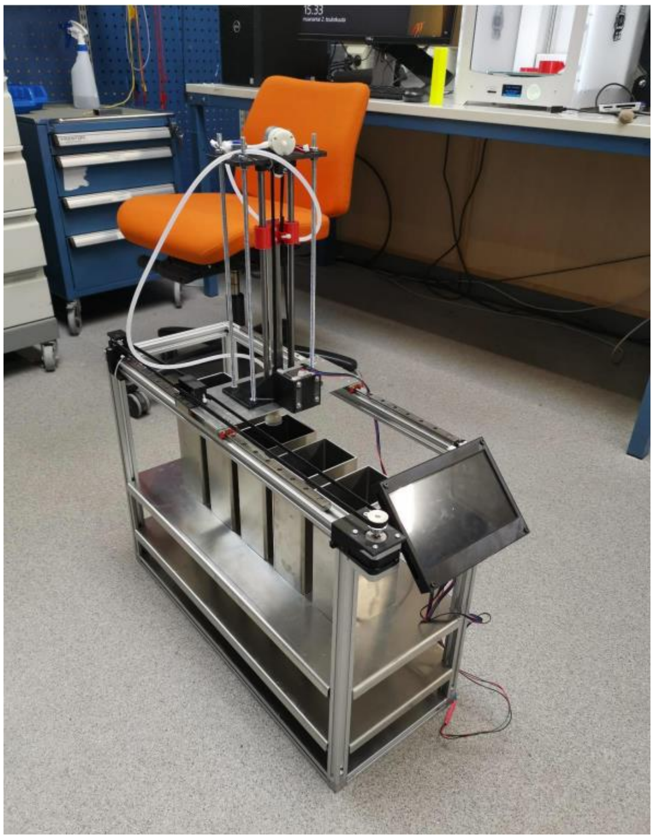
2 Conception

2.1 Physical frame of the machine

As Poromagia’s project contains parts of IT engineering and mechanical engineering, there were 3 mechanical students who worked on the creation of the physical part of the machine, which can be seen below.

As can be seen, the original machine contains five boxes where the cards are stored, a suction cup that is used to suction a specific card, and a vertical arm that raises and lowers the cards. In addition, it contains two motors that are placed in the corners of the left side of the machine. However, it is important to note that the first mechanical students only made the machine frame and nothing else, as they did not have any knowledge of software development or electronics.

Figure 1. Physical frame given by Mechanical students



Subsequently, three other mechanical engineering students attempted to make the arm move vertically, pick up a card by making use of the suction cup to pull the chart up, and subsequently move the arm horizontally to another box slot. However, since they had little knowledge of electronics or programming, they did not implement their part successfully. Therefore, our team not only designed and implemented the software, but also contributed significantly to the physical design of the machine, the correct operation of the motors, and the overall movement in the horizontal and vertical axis.

2.2 Main approach

The main idea of the general operation of the machine is described below.

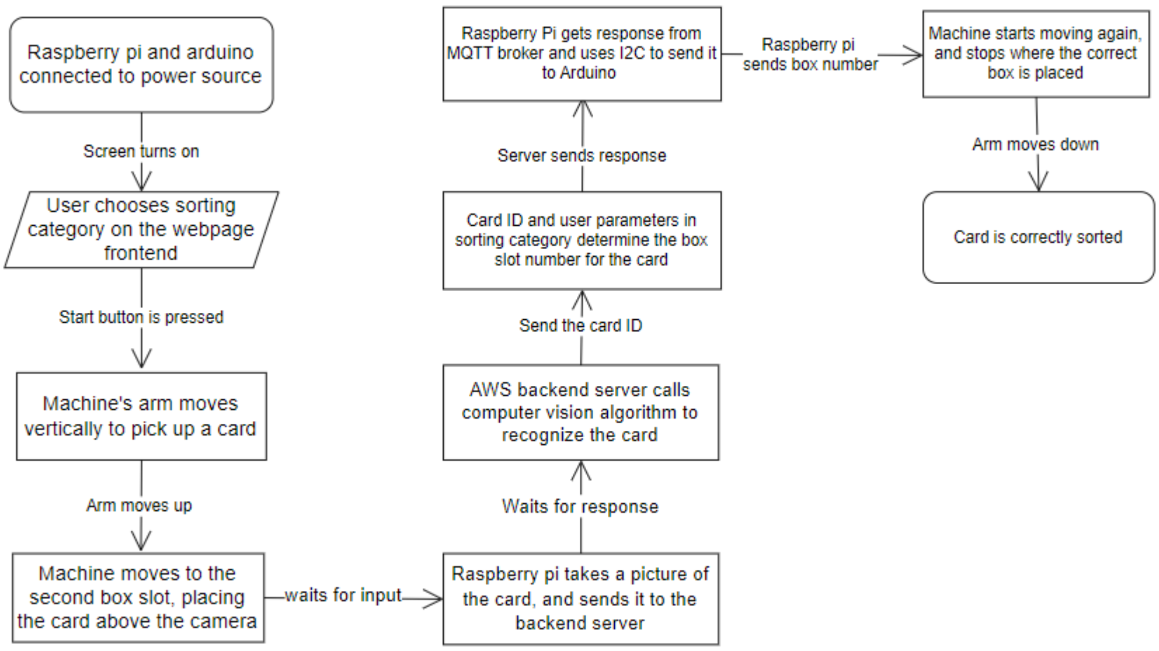

Raspberry pi and arduino connected to power source
Screen turns on
User chooses sorting category on the webpage frontend
Start button is pressed
Machine's arm moves vertically to pick up a card
Arm moves up
Machine moves to the second box slot, placing the card above the camera
waits for input
Raspberry pi takes a picture of the card, and sends it to the backend server
Waits for response
AWS backend server calls computer vision algorithm to recognize the card
Send the card ID
Card ID and user parameters in sorting category determine the box slot number for the card
Server sends response
Raspberry Pi gets response from MQTT broker and uses I2C to send it to Arduino
Raspberry pi sends box number
Machine starts moving again, and stops where the correct box is placed
Arm moves down
Card is correctly sorted

Figure 2. Overall logic diagram

2.2.1 Machine’s start of process

As can be seen in Figure 2, the process starts when the raspberry pi and Arduino are connected to a power outlet. In order to power up the machine, two points of electricity are needed, since each microcontroller has several devices connected to it that need current. The arduino must provide power to an air pump that uses a suction cup, and to the various motors that allow movement. The raspberry pi,

on the other hand, must provide power to the camera that takes the picture of the card, and to the screen that displays the front end of the page.
To use the machine, the user must first use the display connected to the raspberry pi to select a category and enter the necessary parameters, and then click on the "start" button (more detailed information in the "User Manual" document).

Figure 3. Screen that serves frontend

Consequently, the status of the machine is published by the web page, goes through the raspberry pi and finally reaches the arduino, which enables the motors to start moving.

2.2.2 Machine’s card sorting process

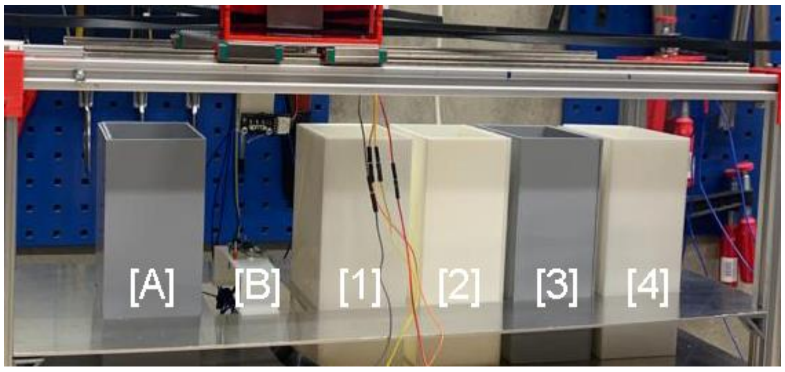


Figure 4. Box slots labeled

As shown in Figure 4, the machine has 6 box slots through which the mechanical arm can move. Box [A] is filled with cards that are ready to be processed, space [B] contains a chamber that is 9.5 centimeters away from the mechanical arm.

Boxes [1, 2 and 3] are where the cards are placed once they have been sorted, these have multiple meanings depending on the category the user has selected. If the user selected "price", then box [1] will be for cards classified as cheap, box [2] for medium priced cards, and box[3] for high priced cards. If the user selected "stock", box [1] will be for cards with very low stock, box [2] for cards with medium stock, and box [3] for cards with high stock. If the user selected "wanted", box [1] will be for cards that are wanted, and box [2] for cards that are not wanted.

Box [4] is fundamental since it will contain the cards that could not be recognized, because some error has happened in the artificial intelligence recognition process, the photo has not been correctly taken, or the sorting server logic is failing.

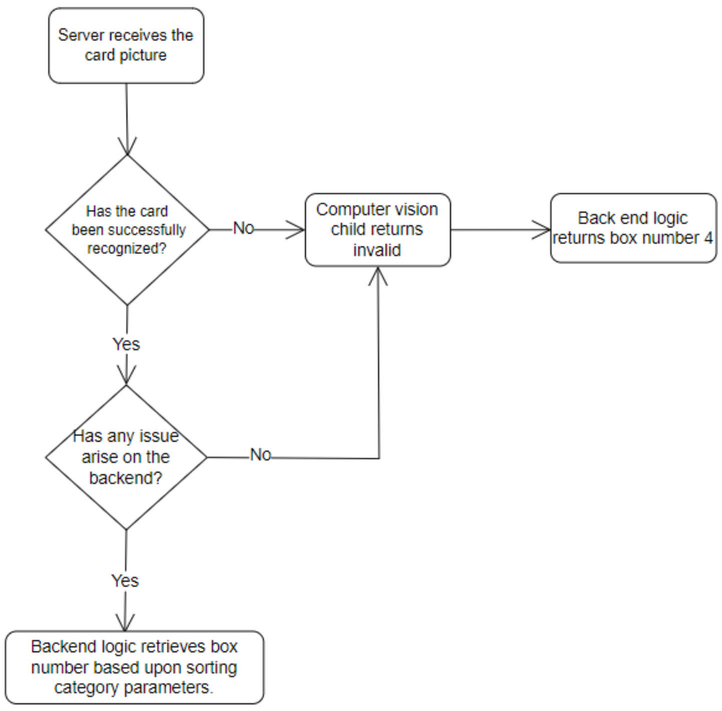

Server receives the card picture
Has the card been successfully recognized?
No
Computer vision child returns invalid
Back end logic returns box number 4
Yes
Has any issue arise on the backend?
No
Yes
Backend logic retrieves box number based upon sorting category parameters.

Figure 5. Box assignment logic diagram

Continuing with the explanation of the process, firstly, the arm goes down on the box [1] to pick up a card. The arm recognizes where it should stop vertically thanks to a small switch that is activated when it has picked up a card.

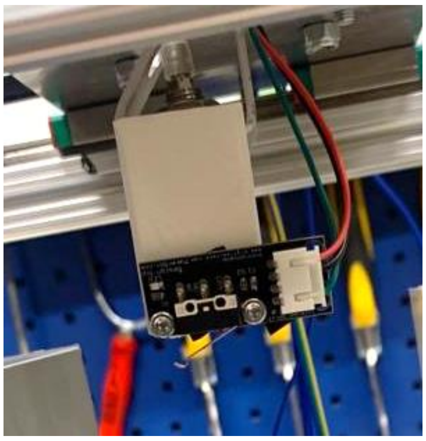


Figure 6. Switch used to detect cards

Once the arm is down and in front of a card, the pump is activated and thanks to the suction cup, the card can be held and then the arduino code allows the arm to go up.

The next step is to move into space [B], shown in Figure 4. This is achieved thanks to the motors used that allow counting the exact number of steps the arm. Then, the arm must move horizontally to get there. Once it stops, the arduino communicates to the raspberry pi via I2C that the card recognition process can start and waits for the result.

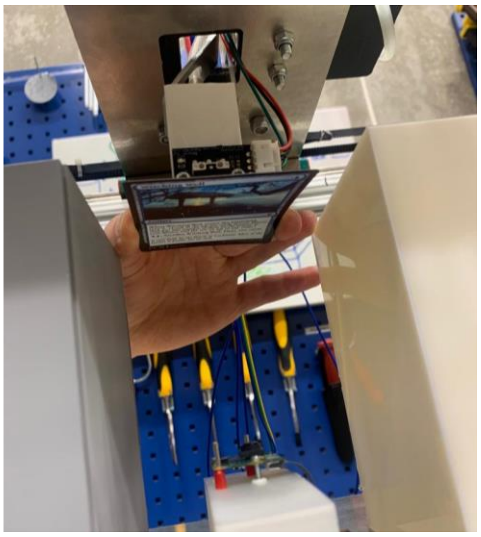


Figure 7. Card placed on top of the camera

A picture of the card is then taken and sent to a cloud server which contains a computer vision model, API endpoints and a database that allows not only the recognition of the card, as it provides its identification number, but also the data that the user has entered on the web page which is necessary to determine the number of the box to which the card should go (more detailed information in section 3.5). Once the backend finished this process, the decision (which is a number between 1-4, corresponding to the numbers in Figure 4) is published.

2.2.3 Machine's end of the sorting cycle

Once the decision has been posted, raspberry receives the response and sends it to the Arduino, which allows the arm to move in the direction of the box number that has been posted. If the process has been successful, the card will go to some box between 1-3, or else the arm will leave the card in box 4.

3 Current State

3.1 Architecture overview

The design of the project architecture can be found below. Multiple programming languages, various communication protocols, cloud servers and databases, as well as frameworks have been used.

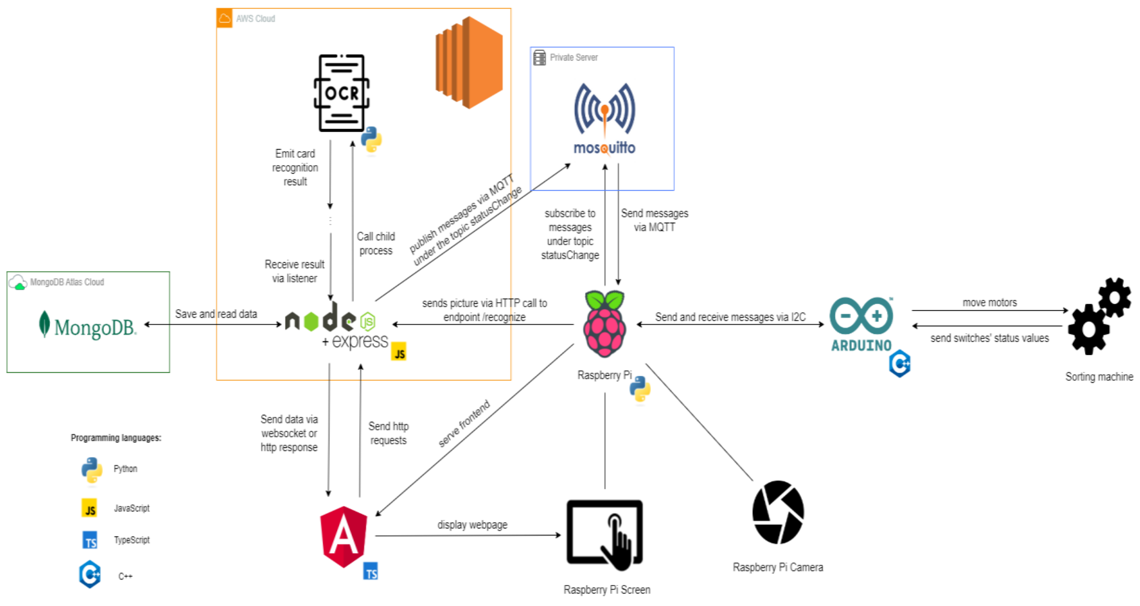


Figure 8. Software architecture diagram [Click to enlarge]

All starts with the Arduino, whose code was developed in C++, since it is the lowest programming part included. This communicates with the machine and allows movement in general. Because the arduino used does not have a wifi module, it communicates with the Raspberry Pi via I2C.

The web page which is fundamental to be able to start using the machine by selecting a category, has its frontend made in Angular using TypeScript. This is shown in the screen shown in figure 3, where the user is allowed to interact with the system.

The frontend communicates constantly with the backend, which is made in Nodejs with javascript and is located in the cloud, on an Amazon Web Services Linux Server. A cloud server is necessary since the front end of the site is on the raspberry, but the backend requires more computational resources, therefore it must be located somewhere else more powerful that can provide answers in a matter of few seconds. In addition, this allows calls to be made to the endpoints from everywhere.

Inside the server, besides the backend where all the endpoints are created, there is the computer vision model developed in Python, which is called from Nodejs every time it is necessary, as a separate process. This is the most resource-consuming part, since it uses a text recognition model called easyOCR that extracts the data from the cards. In addition, it compares the photo taken with many of the cards in the card collection database, so it needs to make requests and have space to compare all possible card matches.

In addition, on the remote server, the connection to the MongoDB database is made, which is also located in the cloud, in Atlas Cloud. The database in the cloud was also necessary, since in order to be able to display data on the website, it had to be accessible from the frontend.

On the other hand, the backend also publishes messages via MQTT, so the project must contain a private broker that allows intercommunication between remote devices. This is necessary because it allows the user from the web page (front end) to send messages about the status of the machine, to stop its process or start it.

Finally, the centre of all communications, the Raspberry Pi, takes care of multiple tasks that are essential to the system. Not only does it serve the front end of the web page, but it also receives messages from the MQTT Broker, communicates with the Arduino, has a camera connected to it, and calls the main backend endpoint sending it the picture taken of the card. Because the microcontroller had to perform all these different tasks, Raspberry Pi was the most suitable choice due to its powerful processor.

3.2 Sorting machine

The machine was in limbo when the team took control of the physical aspect of the project. The conveyor, which carried the cardholder and the suction motor, was not getting enough power from the single DC motor. So, an identical motor was installed on the other end. The motor pairs were able to do the work. The conveyer (like a structure, not exactly a conveyer) could move back and forth. We also split the same lines coming out of the Arduino pins, into two different L298N drivers. One of them had the opposite pin connections as we needed to move those motors in two different directions. The motors were installed on the top right and left edges of the structure. We added two more limit switches. One was to limit the movement of the conveyer along the line. Another one was to know if the cardholder has reached the card yet and the last one would tell if the card-holding arm is stalled up or somewhere in between. It was important to check if the arm is up before moving the conveyer horizontally, or else there would be a risk of damaging the card-holding arm and the suction motor holder.

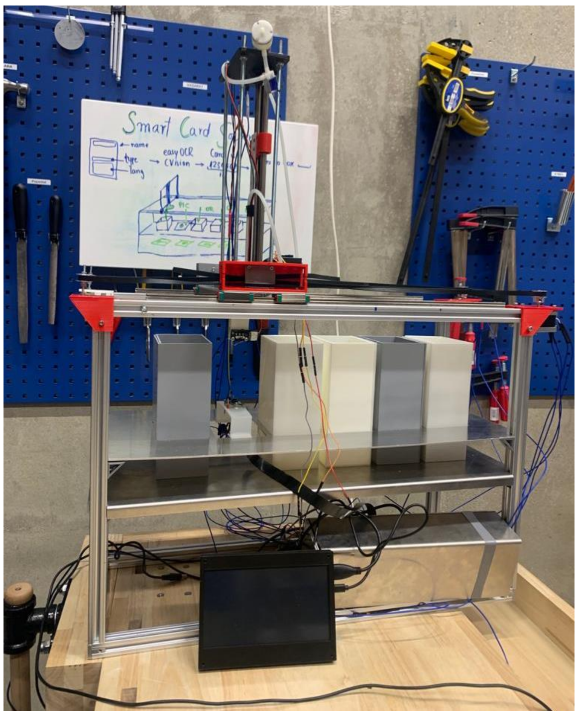


Figure 9. Current physical machine state

In addition, the base material was carefully designed and printed in BigFlash's 3D printed. It needed to be specifically 9.5 centimetres away from the suction cup that holds the card, as the picture requires to be perfectly framed for the computer vision algorithm to work effectively. The camera was screwed to the surface as it had to be always kept stable. Lastly, the team also printed the slot boxes where the cards should be placed, as well as fixed and improved the wiring left by the mechanical engineers' students.

3.3 Arduino

Arduino Uno is a microcontroller board based on the ATmega328P. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator (CSTCE16M0V53-R0), a USB connection, a power jack, an ICSP header and a reset button. It can simply be connected to a computer with a USB cable or powered with an AC-to-DC adapter or battery to get started.

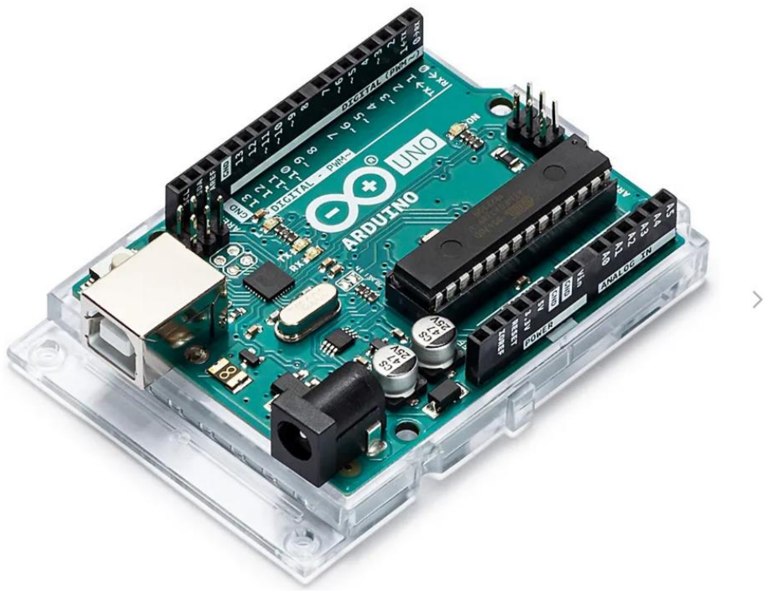
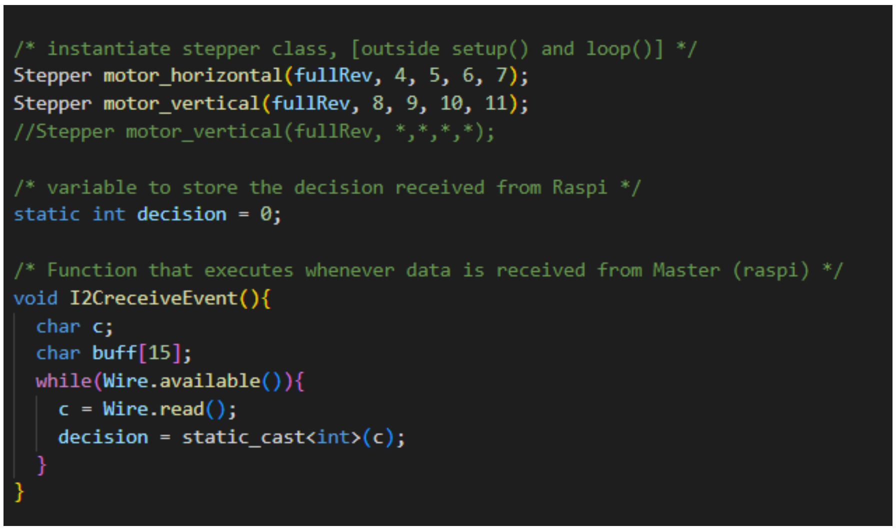


Figure 10. Arduino Uno

It is used as an I2C slave in this system. It controls the step motors and notifies Raspberry pi when the card has been placed on top of the camera. The program flashed into the Arduino can be split into 3 different functions. The first one initializes I2C communication with Raspberry pi, receives decision, and sends command to take picture. Another function is step motors' operation. The motors are installed in horizontal and vertical plane. Last function is to pick up the card with the suction motor and valve installed on the arm.



```
/* instantiate stepper class, [outside setup() and loop()] */
Stepper motor_horizontal(fullRev, 4, 5, 6, 7);
Stepper motor_vertical(fullRev, 8, 9, 10, 11);
//Stepper motor_vertical(fullRev, *,*,*,*);

/* variable to store the decision received from Raspi */
static int decision = 0;

/* Function that executes whenever data is received from Master (raspi) */
void I2CreceiveEvent(){
  char c;
  char buff[15];
  while(Wire.available()){
    c = Wire.read();
    decision = static_cast<int>(c);
  }
}
```

Code 1. Function to execute when data is received from Master.

A slice of the program that waits on I2C data and reads it into a variable is shown above. The optimal speed of the motors is configured as 50. The number of steps between two boxes was counted as approximately 200 (in steps). All the digital IO pins were used up in the Uno. The step motors connect to the L298N which connects with the Arduino using 4 pins each, as the step motors on horizontal plane shared the pins so that totals up to 8 Digital IO pins for the motors. The suction motor required 6V to operate. We found out if Arduino is supplied with 12V, the Vin would supply 12V as well and likewise. The circuit that connects Arduino with suction motor and the valve consisted of mosfets too. The state of the switches was read in a regular interval. In fact, these were the primary determinants of how the process flows. For example, the card handling arm (suction motor holder) would only move horizontally if SW3 is being pressed.

Another such example is that the suction motor will turn off and the valve will be closed if SW2 is pressed.

The Arduino program can be optimized with the implementation of FreeRTOS.

3.4 Raspberry Pi

The system uses as main microcontroller the Raspberry Pi 4 with 4GB of RAM.

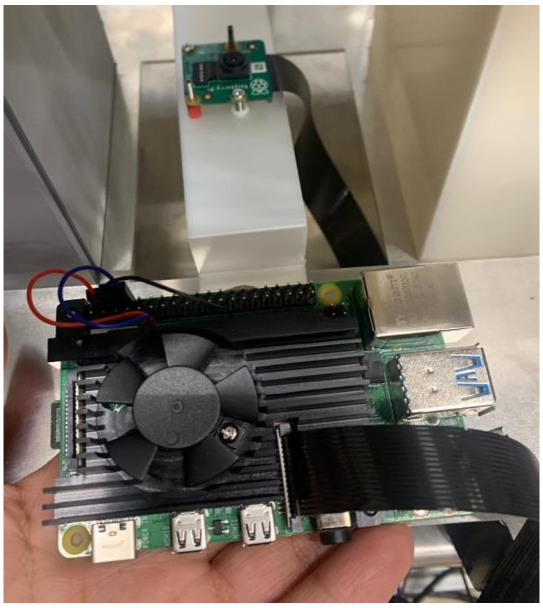


Figure 11. Raspberry pi used in the project

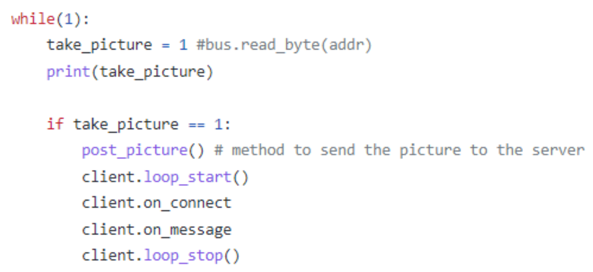
As mentioned in section 3.1, The main need was to find a microcontroller powerful enough to carry out different activities simultaneously. The Raspberry Pi has enough space to store the pictures taken from the charts, and due to its four USB ports, and two micro-HDMI ports, the Raspberry Pi of the project has a 7" screen with resolution up to 1024*600 connected.

Thanks to the Raspberry Pi OS (previously called Raspbian), which is the recommended operating system for normal use on a Raspberry Pi, along with having the project cloned from GitHub and installing all the necessary dependencies to run the project, the screen allows to display the frontend of the web page developed in Angular, which was connected through a HDMI cable easily to the microcontroller.

On the other hand, the camera with which the pictures of the cards are taken is also connected to the Raspberry Pi. The v2 Camera Module has a Sony IMX219 8-megapixel sensor, works perfectly with the Raspberry Pi 4 B, it has been connected to the Pi CSI port by using a one-meter-long cable as shown in the figure 11.

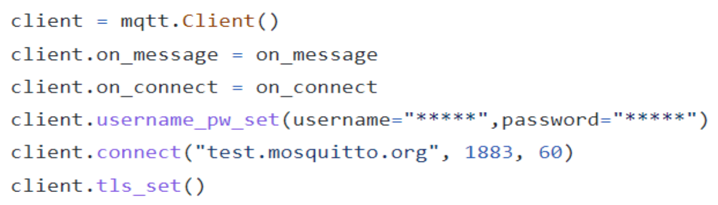
The Raspberry Pi also contains the code that allows the photos taken to be sent to the backend server located in the cloud. It also contains the I2C communication set up as a master, however, that is further described in section 3.7.3.

Code 2. When the position of the motors is suitable, the system will take a picture.



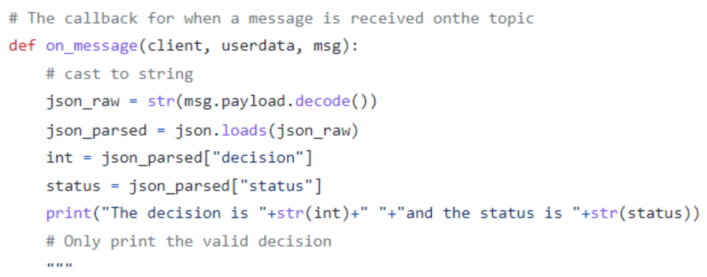
```
while(1):
    take_picture = 1 #bus.read_byte(addr)
    print(take_picture)

    if take_picture == 1:
        post_picture() # method to send the picture to the server
        client.loop_start()
        client.on_connect
        client.on_message
        client.loop_stop()
```

The software also contains the connection to the MQTT broker, as well as the function that subscribes to the topic where the webpage publishes. More information in the section 3.7.1.



```
client = mqtt.Client()
client.on_message = on_message
client.on_connect = on_connect
client.username_pw_set(username="*****",password="*****")
client.connect("test.mosquitto.org", 1883, 60)
client.tls_set()
```

Code 3. Connection to the MQTT broker.



```python
# The callback for when a message is received onthe topic
def on_message(client, userdata, msg):
    # cast to string
    json_raw = str(msg.payload.decode())
    json_parsed = json.loads(json_raw)
    int = json_parsed["decision"]
    status = json_parsed["status"]
    print("The decision is "+str(int)+" "+"and the status is "+str(status))
    # Only print the valid decision
    """
```

Code 4. Function that handles the messages received from the topic.

Finally, due to the number of tasks performed by the Raspberry Pi, the temperature of the microprocessor tends to increase, which can damage the components with prolonged overheating. Therefore, as shown in Figure 11, the Raspberry Pi system has a fan that works by connecting to the GIO pins located on the left side of the entire surface.

3.5 Backend

The backend, written in JavaScript and Python, consists of an HTTP API, a MongoDB database, a MQTT client, a WebSocket server, and a computer vision model for the card recognition.

3.5.1 NodeJS and Express

The backend was implemented with the use of the asynchronous, event-driven JavaScript runtime environment NodeJS. In NodeJS, many connections can be handled concurrently and without locks if no synchronous methods of the NodeJS standard library are used. This is possible by using a so-called Event Loop that offloads operations to the kernel whenever possible. NodeJS enters this event loop after executing the input script and leaves it when there are no more call-backs to perform. [5] Express is a NodeJS framework that offers features and functions for web and mobile applications. [6]

3.5.2 HTTP endpoints

The HTTP API of the Express backend contains middleware for setting the CORS header and for handling errors. In addition, it includes the following routes:

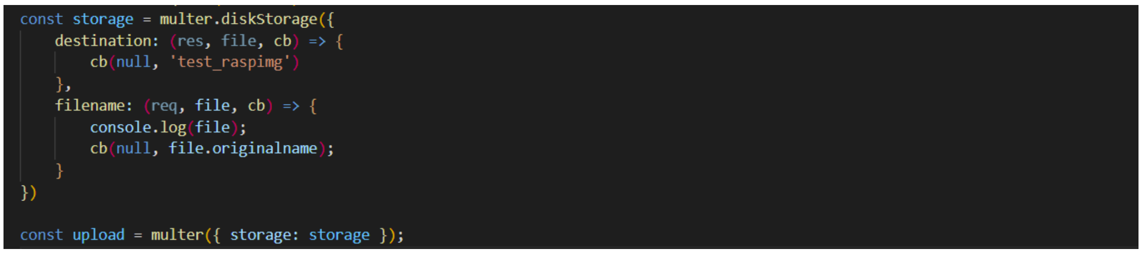
Route	**Me-thod**	**Description**
Routes to control the machine		
/start	POST	Endpoint to start the machine It checks and inserts the sorting values in the database and publishes the start status via MQTT.
/stop	POST	Endpoint to stop the machine It adds the end time to the latest database entry and publishes the stop status via MQTT.
Routes for card recognition		
/recognize	POST	Endpoint to recognize a card It reads the card image from the request body, calls the computer vision model to recognize the card, saves the result in the database and sends it via WebSocket (for the frontend) and via MQTT (for the raspberry pi).
Routes for getting statistics data		
/cardsCount /all	GET	Endpoint to get the number of sorted cards per day It gets all sorted cards in the given time period from the database and counts them per day.
/cardsCount /recognized	GET	Endpoint to get the number of recognized cards per day It gets all cards sorted in box 1-3 in the given time period from the database and counts them per day.
/cardsCount /notRecognized	GET	Endpoint to get the number of not recognized cards per day. It gets all cards sorted in box 4 in the given time period from the database and counts them per day.
/cardsCount /boxes/:id	GET	Endpoint to get the number of cards sorted in the given box per day

Route	**Me-thod**	**Description**
		It gets all cards sorted in the box with the given number in the given time period from the database and counts them per day.
/cardsCount /boxes	GET	Endpoint to get the number of sorted cards per box and day It gets all sorted cards in the given time period from the database, splits them up according to the box they were sorted in and counts them per day for each box.
/recognizeTimes	GET	Endpoint to get the times it took to recognize the cards It calculates the time it took the recognize the card for each card in the given time period; splits the recognize times into 5s intervals and counts the number of cards whose recognize time is in the respective interval.
/cardsCount /categories	GET	Endpoint to get the number of times the machine ran with each category It gets all sorting entries in the requested time period from the database and counts the entries per category
/sortingData /categories	GET	Endpoint to get the times the machine ran It gets the category and the start and end time for all sorting entries in the requested time period and calculates the time between start and end

Table 1. HTTP endpoints of the Express backend

3.5.2.1 Recognize endpoint

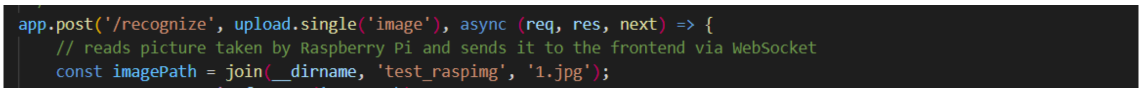
The recognize endpoint is a POST request is one of the main endpoints that make the system work efficiently. The picture sent from the raspberry pi is saved in the remote server in the backend by using the multer module, that gets the card and saves it on a folder.



```
const storage = multer.diskStorage({
    destination: (res, file, cb) => {
        cb(null, 'test_raspimg')
    },
    filename: (req, file, cb) => {
        console.log(file);
        cb(null, file.originalname);
    }
})

const upload = multer({ storage: storage });
```

Code 5. Multer usage in the backend server

After the image is saved, the endpoint simply retrieves the image from the folder to call the computer vision model, save the result to the database, publish the box number via MQTT, and send the image through WebSocket.



```
app.post('/recognize', upload.single('image'), async (req, res, next) => {
    // reads picture taken by Raspberry Pi and sends it to the frontend via WebSocket
    const imagePath = join(__dirname, 'test_raspimg', '1.jpg');
```

Code 6. Recognize endpoint

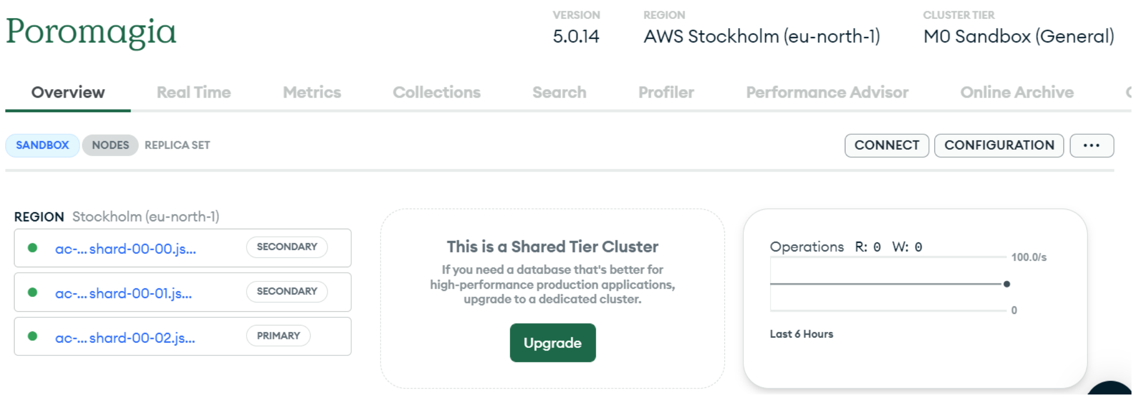
3.5.3 MongoDB

The data is stored in the NoSQL-database MongoDB, which allows storing information in JSON format. The database is made up of different collections containing documents that represent the entries in the database and consist of (sometimes nested) BSON data. BSON is a variant of JSON offering more data types. [1] Because of this basic structure, MongoDB allows adding attributes to single entries without having to change the entire collection.

3.5.3.1 Cloud database

The system makes use of the most advanced cloud database service on the market, MongoDB Atlas Database. An account was created, and the free tier is used, which is more enough for the system demo since the use of the product is intended to be individual. The database needed to be in the cloud, since the server that accesses it is also in the cloud, and the data must be accessed locally from the raspberry pi where the front-end is. The database could have been on the backend server on AWS, however, due to the limited free tier capability of the server, a non-relational database could not be created there, so Atlas database seemed the most suitable option.

Also, it comes with very useful configuration panel to track everything related to database collections.



VERSION 5.0.14
REGION AWS Stockholm (eu-north-1)
CLUSTER TIER M0 Sandbox (General)

REGION Stockholm (eu-north-1)
ac-....shard-00-00.js... SECONDARY
ac-....shard-00-01.js... SECONDARY
ac-....shard-00-02.js... PRIMARY

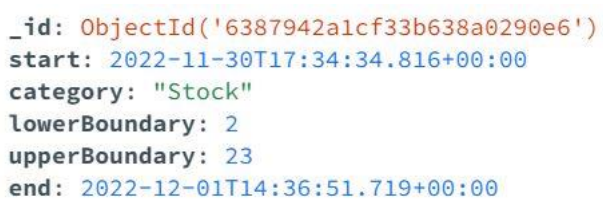
This is a Shared Tier Cluster
If you need a database that's better for high-performance production applications, upgrade to a dedicated cluster.

Operations R: 0 W: 0

Figure 12. Overview of the database in the cloud

3.5.3.2 Collections

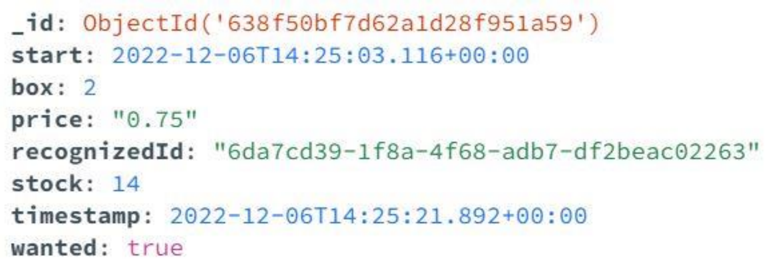
There are two collections used for storing the data. The collection “sortingData” is used for storing the time the machine was started and stopped, as well as the selected category and boundary values.



```
_id: ObjectId('6387942a1cf33b638a0290e6')
start: 2022-11-30T17:34:34.816+00:00
category: "Stock"
lowerBoundary: 2
upperBoundary: 23
end: 2022-12-01T14:36:51.719+00:00
```

Figure 13. Properties of the sortingData collection

The collection “resultData” contains the data of the recognized card including the card id, the price, stock and wanted value, the number of the box the card was sorted into, a timestamp for the moment the card recognition was started and a timestamp when the card was recognized.



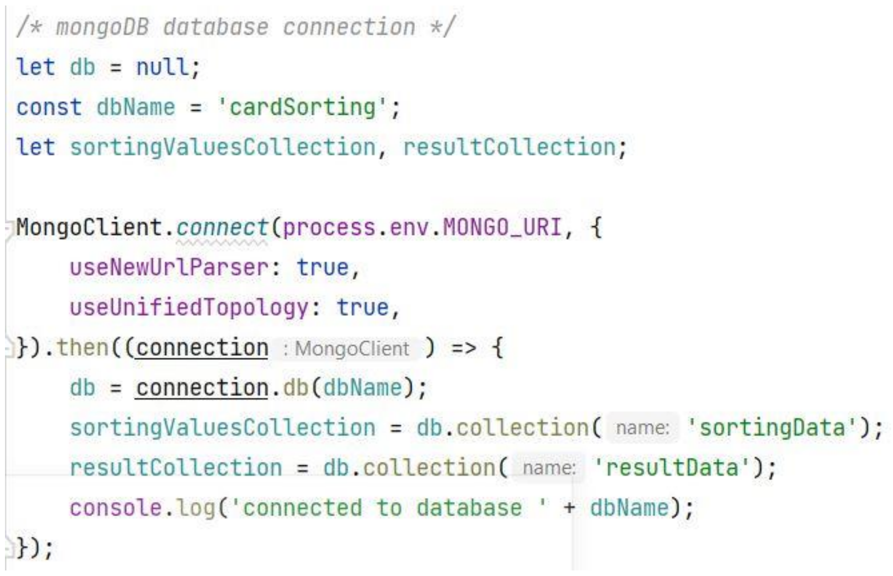
```
_id: ObjectId('638f50bf7d62a1d28f951a59')
start: 2022-12-06T14:25:03.116+00:00
box: 2
price: "0.75"
recognizedId: "6da7cd39-1f8a-4f68-adb7-df2beac02263"
stock: 14
timestamp: 2022-12-06T14:25:21.892+00:00
wanted: true
```

Figure 14. Properties of the resultData collection

The collections are mainly used for tracking the card sorting and for being able to display data and statistics in the frontend, such as the time it took to recognize cards. Users can also use this data to check which was the last selected category, for example. For developers this data can be used to check how recognition times change when changing the computer vision model, for example.

3.5.3.3 Connection to database in backend

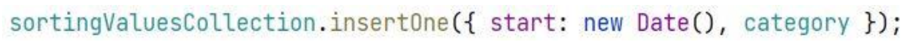
The connection to the “cardSorting” MongoDB database is established in the backend using the mongodb library.



```
/* mongoDB database connection */
let db = null;
const dbName = 'cardSorting';
let sortingValuesCollection, resultCollection;

MongoClient.connect(process.env.MONGO_URI, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
}).then((connection : MongoClient ) => {
    db = connection.db(dbName);
    sortingValuesCollection = db.collection( name: 'sortingData');
    resultCollection = db.collection( name: 'resultData');
    console.log('connected to database ' + dbName);
});
```

Code 7. Establishing a connection to the MongoDB database

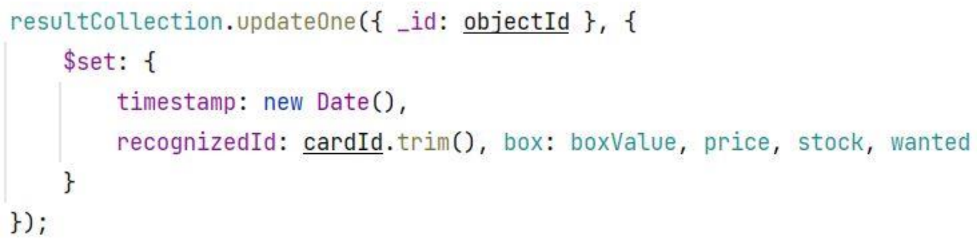
There are multiple functions used to edit and read a collection’s entries. The insertOne function is used to create a new entry in the given collection.



```
sortingValuesCollection.insertOne({ start: new Date(), category });
```

Code 8. Inserting a new entry in the sortingData collection

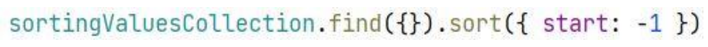
With the updateOne function an existing entry can be changed by adding new properties or changing the value of an existing property.



```
resultCollection.updateOne({ _id: objectId }, {
    $set: {
        timestamp: new Date(),
        recognizedId: cardId.trim(), box: boxValue, price, stock, wanted
    }
});
```

Code 9. Updating the entry with the given objectId in the resultData collection

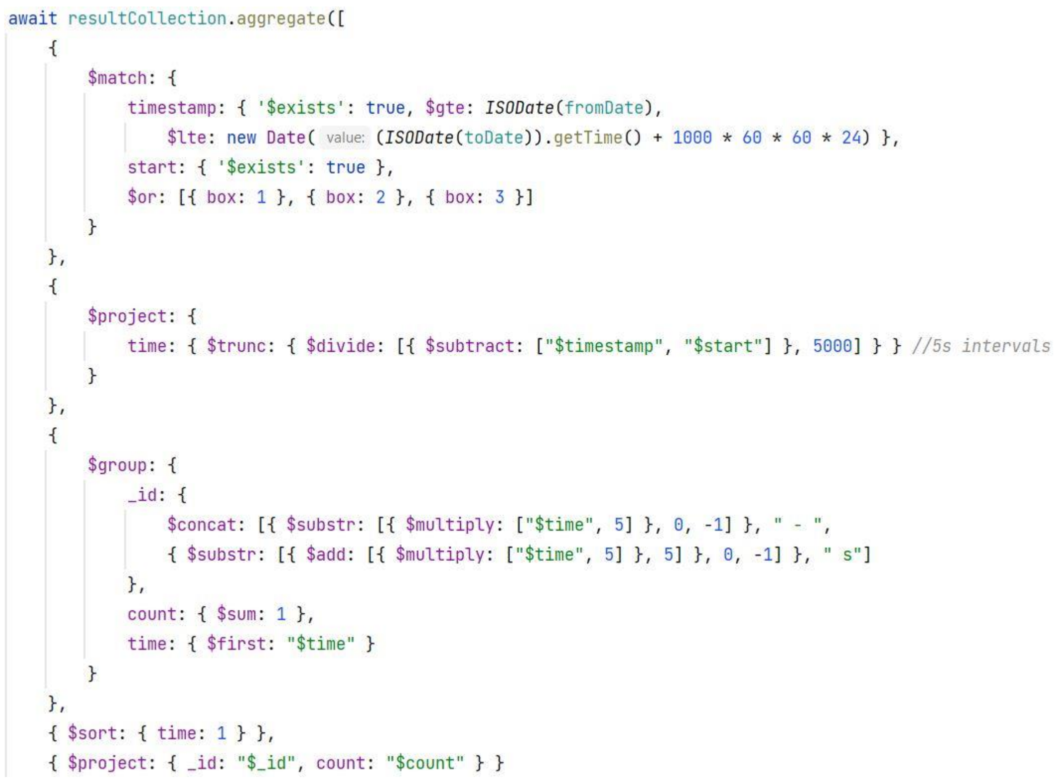
To get one or multiple entries according to given property values the find function is used. The properties are passed to this function as a JSON object, and it can also be used with an empty JSON object to get all entries of a collection.



```
sortingValuesCollection.find({}).sort({ start: -1 })
```

Code 10. Find function to get all entries of the sortingData collection and sort them afterwards

The aggregate function is used for more complex queries. With this function an aggregation pipeline is created to perform multiple operations which are called stages. In this application the stages $match, $sort, $project and $group are mainly used to filter, sort and group database entries or to change the fields in the current query. In these stages different expressions such as $cond (condition), $lte (less than or equal), $sum, $concat, $multiply, etc. can be used to compare values, combine values, or calculate new values. [11]



```
await resultCollection.aggregate([
    {
        $match: {
            timestamp: { '$exists': true, $gte: ISODate(fromDate),
                $lte: new Date( value: (ISODate(toDate)).getTime() + 1000 * 60 * 60 * 24) },
            start: { '$exists': true },
            $or: [{ box: 1 }, { box: 2 }, { box: 3 }]
        }
    },
    {
        $project: {
            time: { $trunc: { $divide: [{ $subtract: ["$timestamp", "$start"] }, 5000] } } //5s intervals
        }
    },
    {
        $group: {
            _id: {
                $concat: [{ $substr: [{ $multiply: ["$time", 5] }, 0, -1] }, " - ",
                { $substr: [{ $add: [{ $multiply: ["$time", 5] }, 5] }, 0, -1] }, " s"]
            },
            count: { $sum: 1 },
            time: { $first: "$time" }
        }
    },
    { $sort: { time: 1 } },
    { $project: { _id: "$_id", count: "$count" } }
```

Code 11. Example of an aggregation pipeline to execute a more complex query

3.5.4 Computer vision model

The card recognition system is developed in Python, and makes use of classes, where each new card to be recognized creates an instance of the two main classes used by the system, TextParser and ImageMatch. through the get_match_and_sort() function.

Besides, the development process of the classes using artificial intelligence has been documented in more detail in Jupyter Notebooks, which can be found in the source code on GitHub.

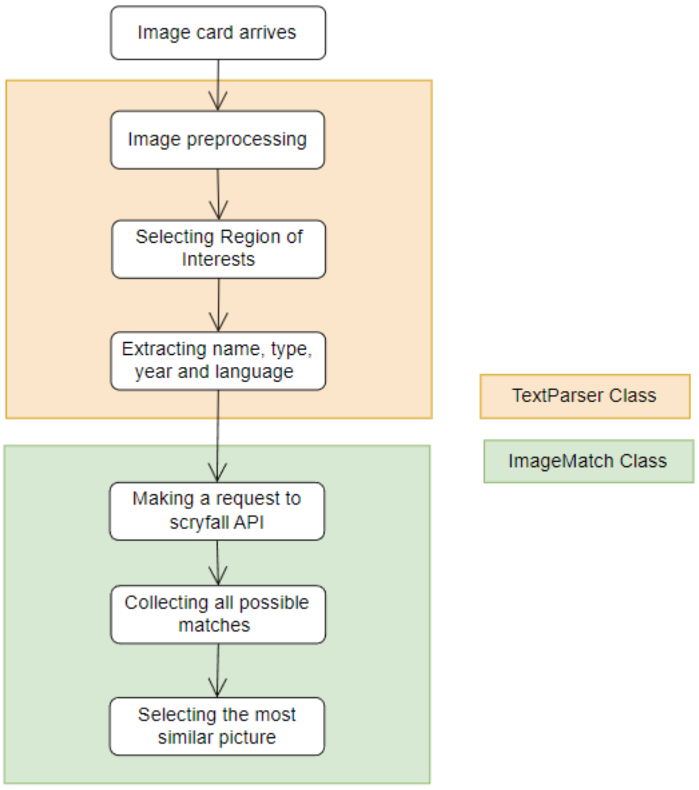


Image card arrives
Image preprocessing
Selecting Region of Interests
Extracting name, type, year and language
Making a request to scryfall API
Collecting all possible matches
Selecting the most similar picture
TextParser Class
ImageMatch Class

Figure 15. Diagram of the Image Recognition classes

3.5.4.1 Magic: The gathering cards

Magic: The Gathering (colloquially known as Magic or MTG) is a tabletop and digital collectable card game created by Richard Garfield. Released in 1993 by Wizards of the Coast (now a subsidiary of Hasbro), Magic was the first trading card game and had approximately thirty-five million players as of December 2018, and over twenty billion Magic cards were produced in the period from 2008 to 2016, during which time it grew in popularity [12].

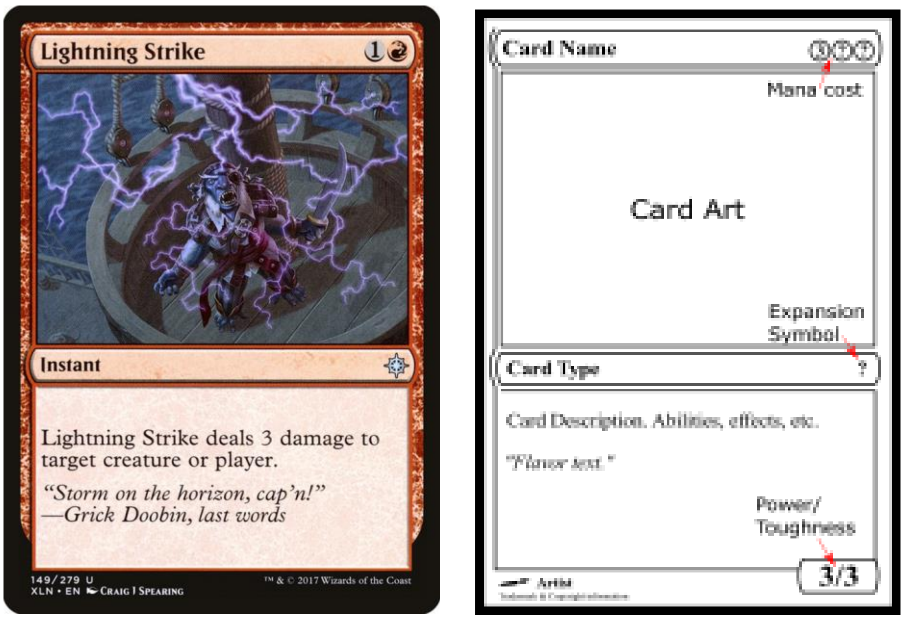

Lightning Strike
Instant
Lightning Strike deals 3 damage to target creature or player.
"Storm on the horizon, cap'n!" —Grick Doobin, last words
149/279 U
XLN • EN Craig J Spearing
™ & © 2017 Wizards of the Coast
Card Name
Mana cost
Card Art
Expansion Symbol
Card Type
Card Description, Abilities, effects, etc.
"Flavor text."
Power/ Toughness
3/3
Artist

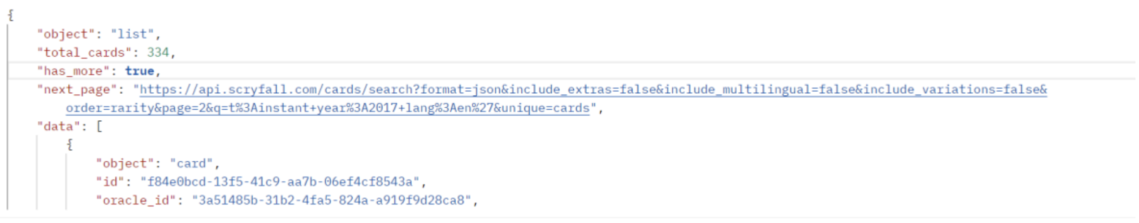
Figure 16. Card example and structure.

In Figure 16, example of the cards that the Poromagia company works with is shown. It can also be seen next to it the structure that the cards usually have, that despite not having a unique identification number that characterizes them, they have several fields that can delimit the number of coincidences with other cards.

The cards, which currently are in the hundreds of thousands number of cards, have a global database and a REST-API that is intended for public use called scryfall. In this, several parameters can be specified to delimit the number of results, since there are many cards that are identified with the same characteristics. For example, the image in the picture, whose name is "Lightning Strike", its type is "Instant", its year is "2017", and its language is English, although these characteristics limit the number of matches of similar cards, if a request is made to the API with the GET method:

https://api.scryfall.com/cards/search?order=rarity&q=t:Instant+year:2017+lang:en

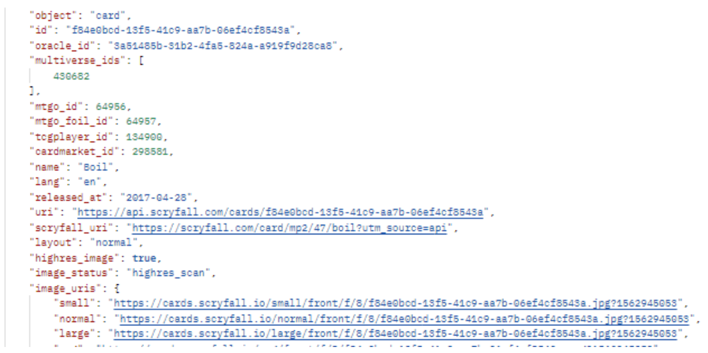
There are 334 cards that contain the same parameters as the ones specified in the API call. This happens because the cards do not have a unique identifier that is printed on the physical surface of the cards.



```
{
    "object": "list",
    "total_cards": 334,
    "has_more": true,
    "next_page": "https://api.scryfall.com/cards/search?format=json&include_extras=false&include_multilingual=false&include_variations=false&
        order=rarity&page=2&q=t%3Ainstant+year%3A2017+lang%3Aen%27&unique=cards",
    "data": [
        {
            "object": "card",
            "id": "f84e0bcd-13f5-41c9-aa7b-06ef4cf8543a",
            "oracle_id": "3a51485b-31b2-4fa5-824a-a919f9d28ca8",
```

Figure 17. Response of the call to the scryfall API.

In the array of the cards that compose the response, there are multiple key-value pairs that are useful. An example is shown below.



```
"object": "card",
"id": "f84e0bcd-13f5-41c9-aa7b-06ef4cf8543a",
"oracle_id": "3a51485b-31b2-4fa5-824a-a919f9d28ca8",
"multiverse_ids": [
    430682
],
"mtgo_id": 64956,
"mtgo_foil_id": 64957,
"tcgplayer_id": 134900,
"cardmarket_id": 298581,
"name": "Boil",
"lang": "en",
"released_at": "2017-04-28",
"uri": "https://api.scryfall.com/cards/f84e0bcd-13f5-41c9-aa7b-06ef4cf8543a",
"scryfall_uri": "https://scryfall.com/card/mp2/47/boil?utm_source=api",
"layout": "normal",
"highres_image": true,
"image_status": "highres_scan",
"image_uris": {
    "small": "https://cards.scryfall.io/small/front/f/8/f84e0bcd-13f5-41c9-aa7b-06ef4cf8543a.jpg?1562945053",
    "normal": "https://cards.scryfall.io/normal/front/f/8/f84e0bcd-13f5-41c9-aa7b-06ef4cf8543a.jpg?1562945053",
    "large": "https://cards.scryfall.io/large/front/f/8/f84e0bcd-13f5-41c9-aa7b-06ef4cf8543a.jpg?1562945053",
```

Figure 18. Element zero of the array of the response to the previous call.

Each element of the array of the response contains an “id” key-value pair, along with a “image_uris” which contains the pictures of the specific element in different image qualities.

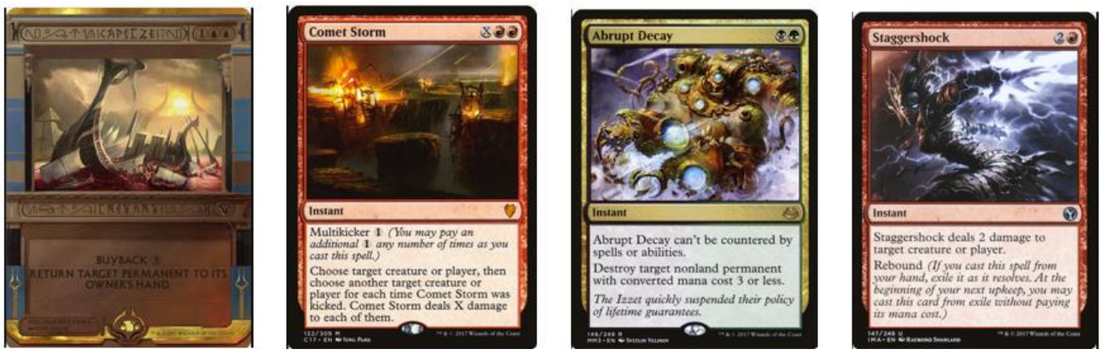
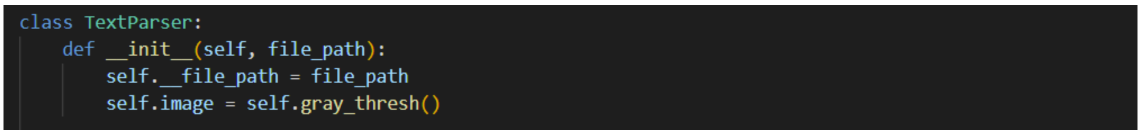

BUYBACK 3
RETURN TARGET PERMANENT TO ITS OWNER'S HAND.
Comet Storm
Instant
Multikicker 1 (You may pay an additional 1 any number of times as you cast this spell.)
Choose target creature or player, then choose another target creature or player for each time Comet Storm was kicked. Comet Storm deals X damage to each of them.
Abrupt Decay
Instant
Abrupt Decay can't be countered by spells or abilities.
Destroy target nonland permanent with converted mana cost 3 or less.
The Izzet quickly suspended their policy of lifetime guarantees.
Staggershock
Instant
Staggershock deals 2 damage to target creature or player.
Rebound (If you cast this spell from your hand, exile it as it resolves. At the beginning of your next upkeep, you may cast this card from exile without paying its mana cost.)

Figure 19. Few card images from ”image_uris” key in the response array.

3.5.4.2 TextParser class

The first part of the card recognition system is called TextParser class, which is the class that oversees extracting the text.



```
class TextParser:
    def __init__(self, file_path):
        self.__file_path = file_path
        self.image = self.gray_thresh()
```

Code 12. TextParser Class initialization

The class only needs to receive a photo to create an instance. Upon receiving the photo, it is converted to black and white since thus instead of each pixel in the array of numbers that make up a photo [r,g,b], each pixel contains either a 255 or a 0, i.e., black or white, which reduces processing time. Besides, the size of the picture is taken into consideration, because the columns and rows are used later to delimit the name, type, language and year.

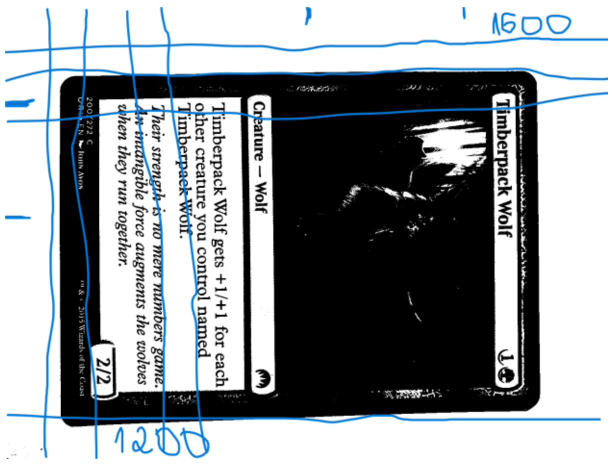


Figure 20. Columns and rows of a card.

From there, the image goes through adaptive thresholding processes, pre-processing that allows to standardize the images with a different contrast or brightness to maximize the readability of the different text fields of the carts.

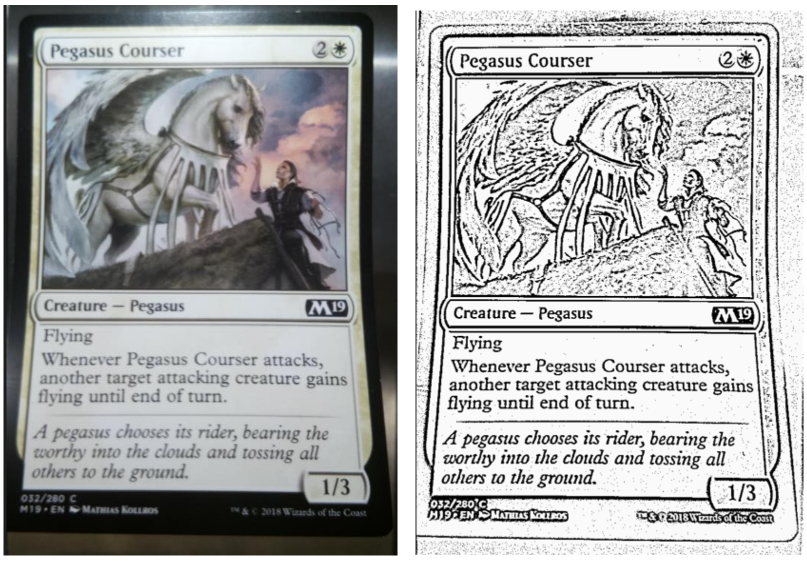
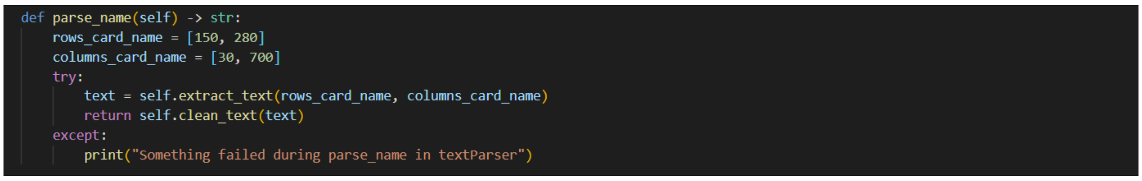
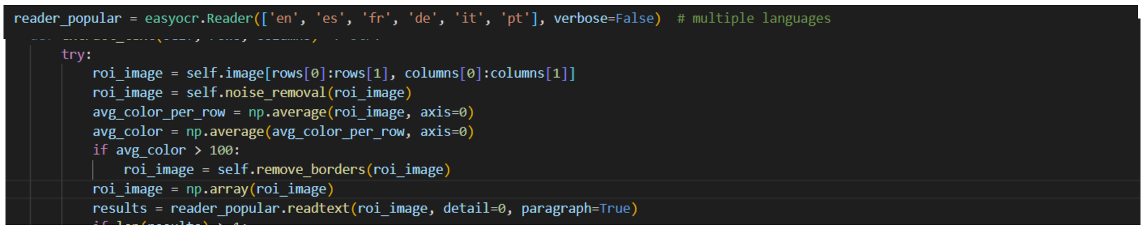

Pegasus Courser
Creature — Pegasus
M19
Flying
Whenever Pegasus Courser attacks, another target attacking creature gains flying until end of turn.
A pegasus chooses its rider, bearing the worthy into the clouds and tossing all others to the ground.
1/3
032/280 C
M19 • EN Mathias Kollros
™ & © 2018 Wizards of the Coast

Figure 21. Card difference after pre-processing methods.

After, ROI (Region of Interest) are taken by making use of the specific [row, column] position where the parameters start and end (as loose as possible to maximize the possibilities to detect the field). Name, type, language, and year are taken following that process.


```
def parse_name(self) -> str:
    rows_card_name = [150, 280]
    columns_card_name = [30, 700]
    try:
        text = self.extract_text(rows_card_name, columns_card_name)
        return self.clean_text(text)
    except:
        print("Something failed during parse_name in textParser")
```
Code 13. Function that specifies the ROI of the name of a card.

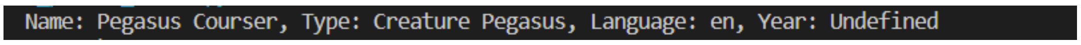
The function extract_text, makes use of the library easy_OCR to extract the text. EasyOCR is a python module for extracting text from image. It is a general Optical Character Recognition that can read both natural scene text and dense text in document. The module is currently supporting 80+ languages [13].



```
reader_popular = easyocr.Reader(['en', 'es', 'fr', 'de', 'it', 'pt'], verbose=False)  # multiple languages
    try:
        roi_image = self.image[rows[0]:rows[1], columns[0]:columns[1]]
        roi_image = self.noise_removal(roi_image)
        avg_color_per_row = np.average(roi_image, axis=0)
        avg_color = np.average(avg_color_per_row, axis=0)
        if avg_color > 100:
            roi_image = self.remove_borders(roi_image)
        roi_image = np.array(roi_image)
        results = reader_popular.readtext(roi_image, detail=0, paragraph=True)
```

Code 14. Extract text function making use of easyOCR.

After the pre-processing, text extraction, text cleaning takes into place, as the text recognition might read some random letters which are not in the picture but are just texture. Below is the result of the text recognition in the figure 21.

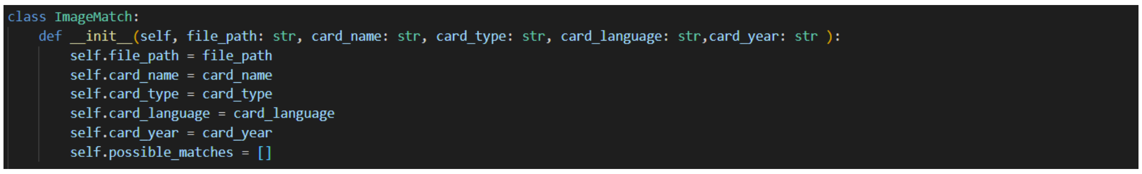


```
Name: Pegasus Courser, Type: Creature Pegasus, Language: en, Year: Undefined
```

Figure 22. Card extraction text results.

3.5.4.3 ImageMatch class

The ImageMatch class, which oversees finding the matching card of the taken photo, receives as parameters the results of the TextParser class, i.e., the path of the photo, the name, type, language, and year of the TextParser class extracted in the previous step.



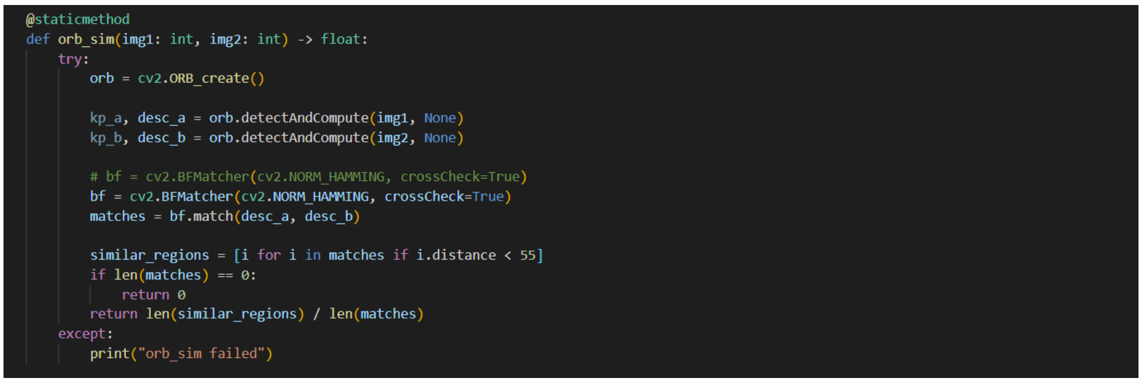
```
class ImageMatch:
    def __init__(self, file_path: str, card_name: str, card_type: str, card_language: str,card_year: str ):
        self.file_path = file_path
        self.card_name = card_name
        self.card_type = card_type
        self.card_language = card_language
        self.card_year = card_year
        self.possible_matches = []
```

Code 15. ImageMatch Class initialization

After the initialization, the class makes use of the scryfall API, described in the section 3.5.4.1. Function called requests_to_server() calls the scryfall resfulAPI by using the input from the TextParser class. The request returns an array of all the possible matches that coincide with the same parameters given in the request, as shown in code 15.

For each element of the resulting array, the “id” and “image_uris” parameters are collected and looped through to compare each image to the original card to be recognized. This is achieved in orb_sim() which makes use of the openCV library.

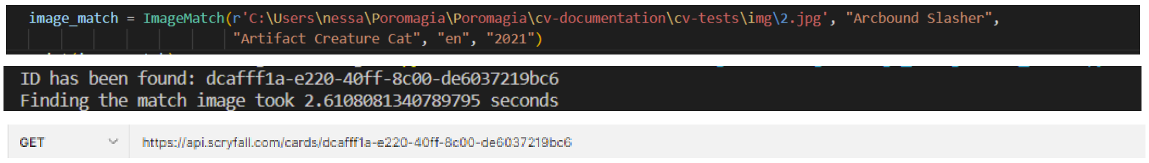
OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, among other possibilities [14].



```python
@staticmethod
def orb_sim(img1: int, img2: int) -> float:
    try:
        orb = cv2.ORB_create()

        kp_a, desc_a = orb.detectAndCompute(img1, None)
        kp_b, desc_b = orb.detectAndCompute(img2, None)

        # bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
        matches = bf.match(desc_a, desc_b)

        similar_regions = [i for i in matches if i.distance < 55]
        if len(matches) == 0:
            return 0
        return len(similar_regions) / len(matches)
    except:
        print("orb_sim failed")
```

Code 16. Method that uses computer vision to gets the similarity percentage of a card compared to the original card

After the array that contains the similarities of all the matching card with the original card is filled, the highest percentage number contains the key-pair of the image selected as the final match, and then its scryfall ID is returned, as it is how the Poromagia database organizes the cards too. If the ID is used to make a GET request to the scryfall API, the recognize d card is returned.



```python
image_match = ImageMatch(r'C:\Users\nessa\Poromagia\Poromagia\cv-documentation\cv-tests\img\2.jpg', "Arcbound Slasher",
                         "Artifact Creature Cat", "en", "2021")
```

```
ID has been found: dcafff1a-e220-40ff-8c00-de6037219bc6
Finding the match image took 2.6108081340789795 seconds
```

```
GET    https://api.scryfall.com/cards/dcafff1a-e220-40ff-8c00-de6037219bc6
```

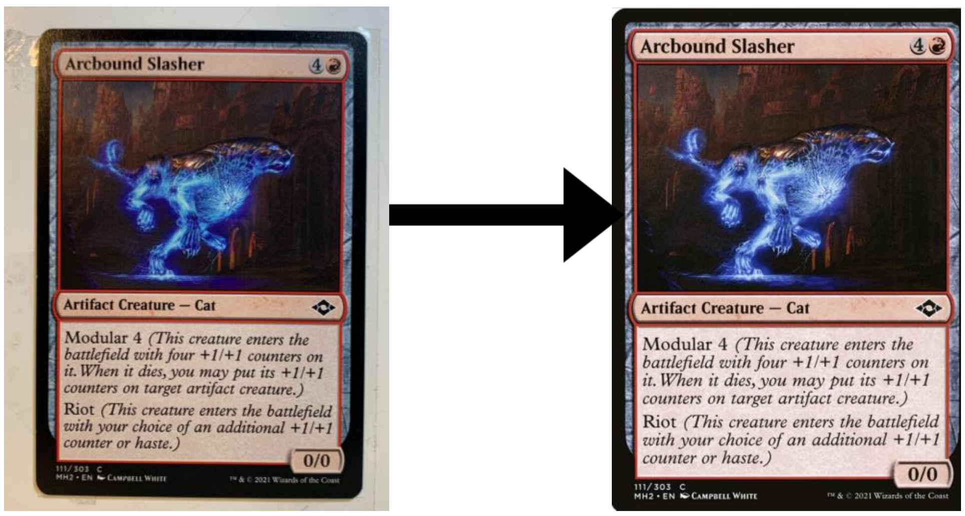


Figure 23. Image recognition process: instance of the ImageMatch with the required parameters, found ID from scryfall API, and gotten original card match.

3.5.5 Deployment

Since the memory on the Raspberry Pi is limited and the backend, especially the card recognition, uses a lot of memory, it is not possible to run the backend and frontend on the Raspberry Pi. Thus, the backend is deployed in the cloud, so that it can be called by the Raspberry Pi without using its memory.

3.5.5.1 AWS and EC2

Amazon Web Services (AWS) is the world’s most comprehensive and broadly adopted cloud platform, offering over 200 fully featured services from data centers globally. Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services (AWS) Cloud [10].

Amazon EC2 provides the following features:

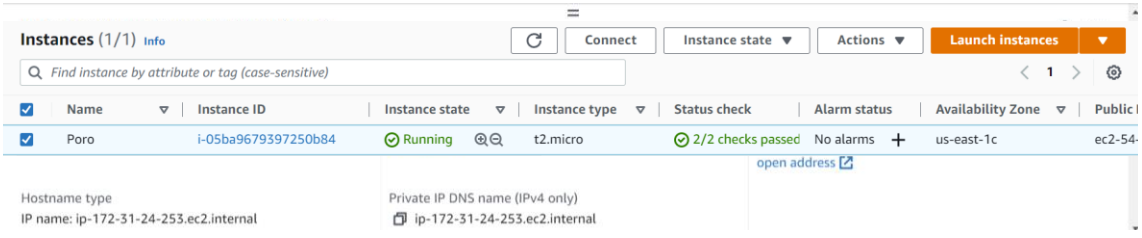
- Virtual computing environments, known as instances.
- Various configurations of CPU, memory, storage, and networking capacity for your instances, known as instance types.

- Secure login information for your instances using key pairs.

- A firewall that enables you to specify the protocols, ports, and source IP ranges that can reach your instances using security groups.

3.5.5.2 Amazon Linux Server

An AWS server has been created to store the backend, whose operating system is Amazon Linux. Amazon Linux 2 is a Linux operating system from Amazon Web Services (AWS). It provides a security-focused, stable, and high-performance execution environment to develop and run cloud applications [15].



	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public I
☑	Poro	i-05ba9679397250b84	Running	t2.micro	2/2 checks passed	No alarms	us-east-1c	ec2-54-

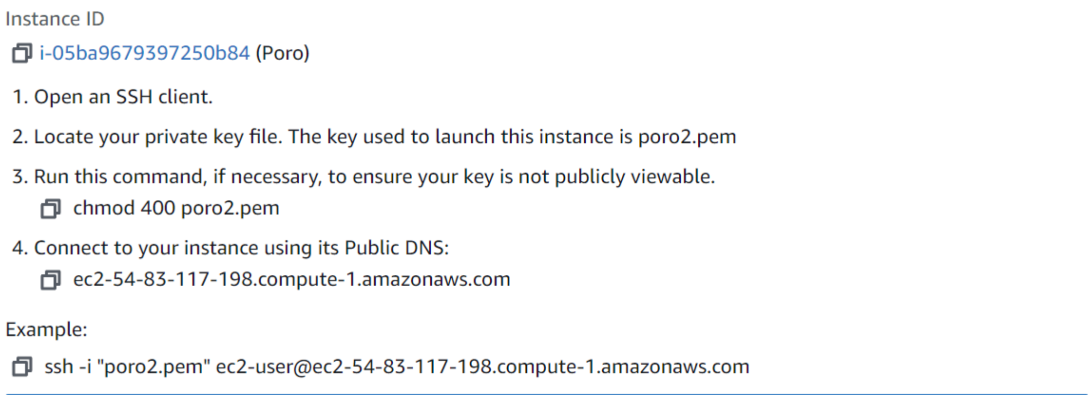
open address

Hostname type
IP name: ip-172-31-24-253.ec2.internal

Private IP DNS name (IPv4 only)
ip-172-31-24-253.ec2.internal

Figure 24. Poromagia’s instance console management.

The backend can be accessed by using putty, and SSH as the network communication protocol, and it can be accessed by using the key password, and the following instructions.



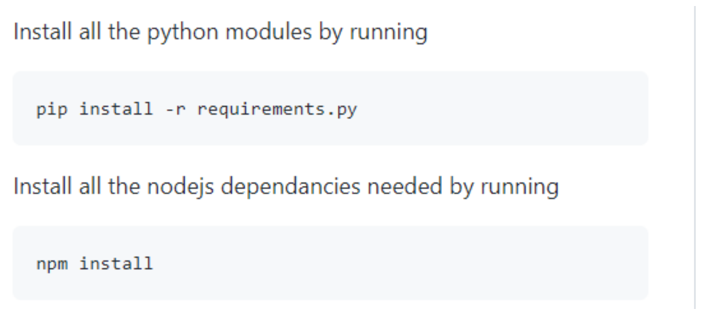
Instance ID
i-05ba9679397250b84 (Poro)

1. Open an SSH client.
2. Locate your private key file. The key used to launch this instance is poro2.pem
3. Run this command, if necessary, to ensure your key is not publicly viewable.
   ```
   chmod 400 poro2.pem
   ```
4. Connect to your instance using its Public DNS:
   ```
   ec2-54-83-117-198.compute-1.amazonaws.com
   ```

Example:
```
ssh -i "poro2.pem" ec2-user@ec2-54-83-117-198.compute-1.amazonaws.com
```

Figure 25. Instance SSH access instructions.

The server has cloned the repostory of Poromagia in GitHub, in the same way described in the section 3.4, and consequently, Node, npm, Python and pip. were installed along with all the required dependancies



Install all the python modules by running

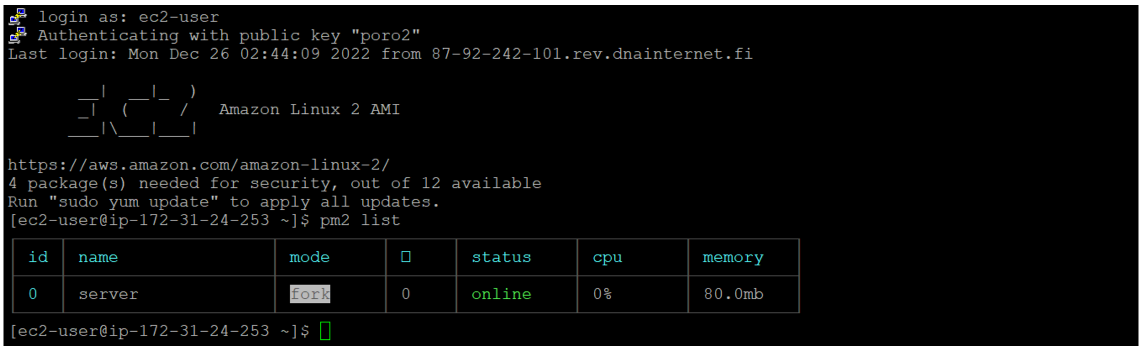
```
pip install -r requirements.py
```

Install all the nodejs dependancies needed by running

```
npm install
```

Figure 26. Installation of all the required dependencies in the backend server.

In order to make the Express server run all the time, the module pm2 was used.



```
login as: ec2-user
Authenticating with public key "poro2"
Last login: Mon Dec 26 02:44:09 2022 from 87-92-242-101.rev.dnainternet.fi

       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
4 package(s) needed for security, out of 12 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-24-253 ~]$ pm2 list
 id | name      | mode | ↺ | status | cpu | memory
 0  | server    | fork | 0 | online | 0%  | 80.0mb
[ec2-user@ip-172-31-24-253 ~]$
```

Figure 27. Server.js always running in the instance.

3.6 Frontend

In this application Angular is used as the development platform to implement the frontend. Therefore, the frontend consists of multiple reusable Angular components and Angular services.

3.6.1 Angular

Angular is built on Typescript and includes a component-based framework to build scalable web applications as well as integrated libraries offering features like routing or client-server communication. [2] Typescript is built on JavaScript, adding additional syntax for types and converting to JavaScript making it possible for it to run everywhere JavaScript runs. [3] The components are split into a HTML template, a SCSS styling file, and a typescript class. The HTML-Template instructs Angular how to render the component, the SCSS file defines the style of the HTML elements, the typescript class creates the functionality and specifies Angular-specific information - like which HTML-template to use - through a @Component() decorator. Angular extends the HTML syntax so that in the HTML-template, dynamic values that are automatically updated if their state changes, can be added from the corresponding component.[2] Additionally, each component contains a spec-file to test the component. All components are part of the app module, which also handles the routing between the components and contains services as well as a global HTML-, SCSS, and Main-Typescript-File. Services are “objects that get instantiated just once during the lifetime of an application. They contain methods that maintain data throughout the life of an application” [4] and make it possible to share data with different components. Usually, they are implemented by using Dependency Injection which is explained below.

3.6.2 Component overview

The Angular frontend complies with the general Angular structure with services as well as components consisting of HTML template, SCSS styling file, typescript class, and testing file. All components and services in this project are part of one module called “app”.

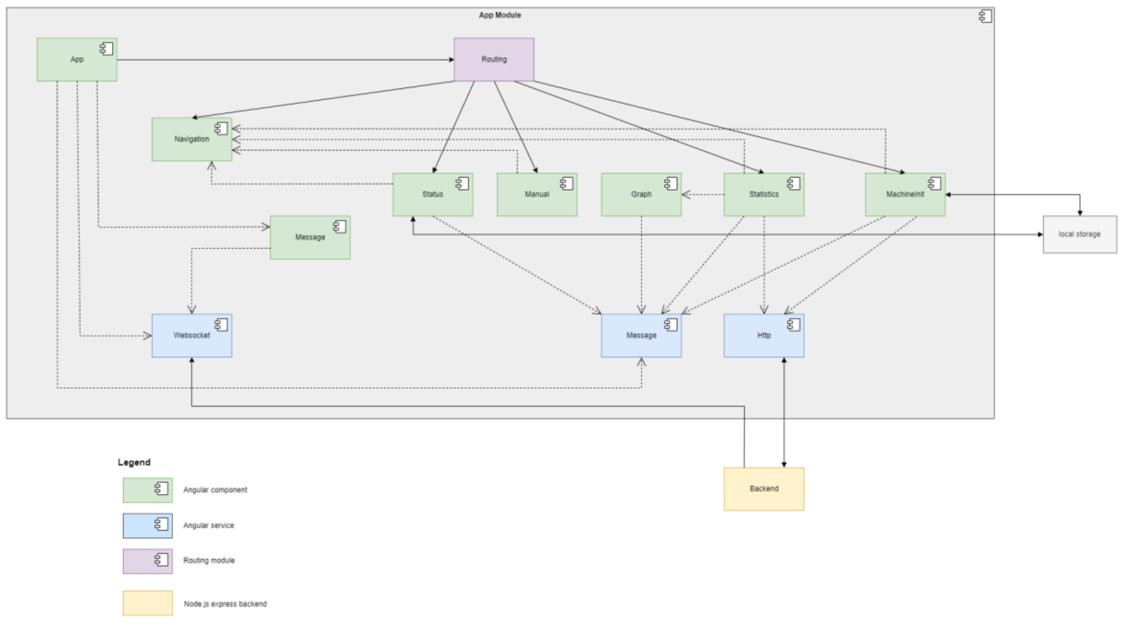


Figure 28. Component diagram of the Angular components and services and the backend components and dependencies. [Click here to enlarge]

3.6.2.1 Components

The **app** component is the main component, which acts as a container for all other components. It only contains the messages component and a router outlet element which gets replaced by the currently displayed component depending on the current URL (managed by the routing module). In the app component, the websocket service is called to set up the WebSocket connection and a listener for all error messages is added to this connection so that those messages are directly forwarded to the messages service and displayed in the message component.

The **message** component is used for displaying pop up messages, such as error messages, warnings, or success messages in the application. It gets the messages from the messages service and displays an element for each message. The styling of the message depends on the message type so that users can easily recognize errors, warnings, and success messages.

The **navigation** component is included in the status, manual, statistics, and machineInit component. It contains an expandable hamburger menu enabling the user to navigate between the different routes. When calling this component, the index of the currently displayed page is added as an input value, so that the page can be highlighted in the menu.

The **status** component can be accessed via the URL http://localhost:4200/status. It displays the current card status including the taken picture, a picture of the recognized card, the Poromagia-values of the recognized card (price, stock and wanted), and the number of the box, in which the card is being sorted. The component interacts with the websocket service to add listeners to different WebSocket messages from the backend for getting the card data and images after a new picture was taken or a new card was recognized. The card status is read (on initialization) from and written to the local storage so that it remains the same even after closing and reopening the webpage.

The **manual** component is displayed when entering the URL http://localhost:4200/manual. It contains a user manual with different expandable segments for the different pages.

The **graph** component is a generic component for displaying charts in the application. The chart is creating according to the given input parameters. The x-axis values, y-axis values and dataset labels are required values. More values just as the chart type, axis titles or tension values can be added. In general, the chart can contain one or multiple datasets.

The **statistics** component can be accessed via the URL http://localhost:4200/statistics. It contains a dropdown to select the requested time period and the data that should be displayed. When those values are set, the component calls the http service to send an http request to the NodeJs backend and get the requested data. After the data is received, it is processed, and an instance of the graph component is created with the parsed data to display the requested chart.

The **machineInit** component is the component displayed when entering the base URL http://localhost:4200/ and users are redirected to this page if they enter an invalid URL such as http://localhost:4200/abc. This component enables the user to start and stop the machine and enter the sorting values which are used to determine in which box a card is sorted. The component contains a start or stop button according to the current machine status. If this button is pressed the http service is called to send the start or stop request to the backend. The current machine status and the sorting values are stored in the local storage and read when the component is initialized to make sure the correct status is always displayed even after closing and reopening the page.

3.6.2.2 Services

The **http** service handles the HTTP communication between the Angular components and the server. It contains multiple methods to send HTTP requests to the backend and return the response to the respective component. It contains one method to handle all calls to statistics endpoints, which calls one of the methods to get statistics data according to the requested data. This method is used to be able to generically call all statistics endpoints with the same method and therefore making it easy to add and remove charts in the statistic component.
The **websocket** service handles the WebSocket communication between the Angular components and the server. It contains methods to set up the connection and to disconnect, as well as methods to add and remove listeners.
The **message** service adds messages to a message list for a given period, so that they are displayed in the UI. It also saves the type of the message, so that this type can be used for styling the message in the message component.

3.7 Communication technologies

3.7.1 MQTT

The communication between the Server and the Raspberry Pi is established using MQTT. MQTT is a standard messaging protocol for the Internet of Things, designed as a publish/subscribe messaging transport. MQTT clients are very small and require minimal resources, so they can be used on small microcontrollers. [7]

3.7.1.1 Broker

Broker acts as a manager for the MQTT messages as everything from publishers to subscribers goes through the broker. The Arduino Uno neither has an ethernet nor a wi-fi module that could establish a TCP/IP network. So, the MQTT client was configured on Raspberry, and communicated with Arduino over I2C lines.

In this project, MQTT facilitates the communication between the Server and the Raspberry pi. A MQTT client on the Raspberry subscribes to messages on the topic where the decision is published once the algorithm recognizes a card. The decision is an integer, the box number where the identified card belongs to. Then, the Raspberry sends the decision (integer) to the Arduino via I2C line.

The mosquitto MQTT broker is configured for the project that runs under the address hetkinen.ddns.net on port 2048. It is password protected. Port forwarding has been enabled on a home router that forwards all the traffic on port 2048 to the server that hosts the broker. A dynamic update client is installed on the server that updates the public IP address so that the domain name is always accessible. The information is published as:

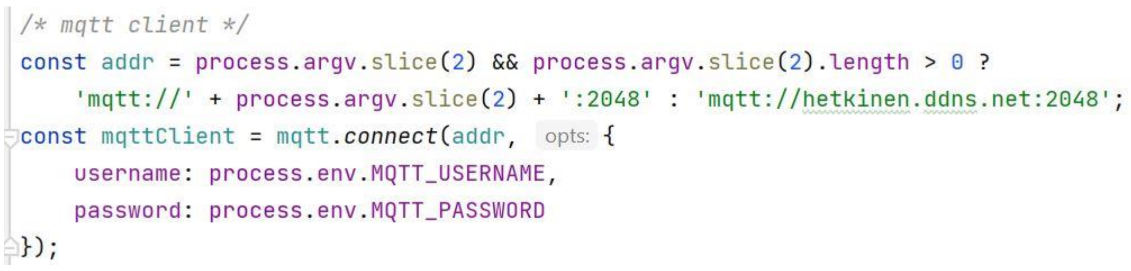
```
{
        “state”: “0”,
        “decision”: “4”
}
```

The JSON message informs about the state of the machine, basically the Arduino. It goes into a deep sleep state if the value is 0. And decision is the output from the identification algorithm. The state is either 0 for off, or 1 for on, while the decision ranges from 0 to 5.

The implementation of the broker was critical to the successful development of the software, because although the decision of which box slot the card should go in can be returned through a POST request, the status of the machine (which the user handles through the web page when the user presses the "start" button) has no other way to be notified but to be published from the server for Raspberry to read.

3.7.1.2 Backend implementation

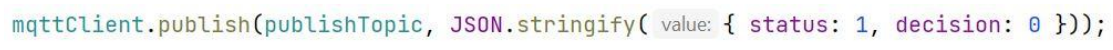
In the NodeJS backend a connection to the MQTT broker is established using a username and password.



```
/* mqtt client */
const addr = process.argv.slice(2) && process.argv.slice(2).length > 0 ?
    'mqtt://' + process.argv.slice(2) + ':2048' : 'mqtt://hetkinen.ddns.net:2048';
const mqttClient = mqtt.connect(addr,  opts: {
    username: process.env.MQTT_USERNAME,
    password: process.env.MQTT_PASSWORD
});
```

Code 17. Connecting to the MQTT broker in the backend

MQTT messages are published in the backend under the topic “statusChange” and read by the Raspberry Pi. MQTT messages contain the status of the machine and number of the box the current card should be sorted into.



```
mqttClient.publish(publishTopic, JSON.stringify( value: { status: 1, decision: 0 }));
```

Code 18. Publishing a message via MQTT

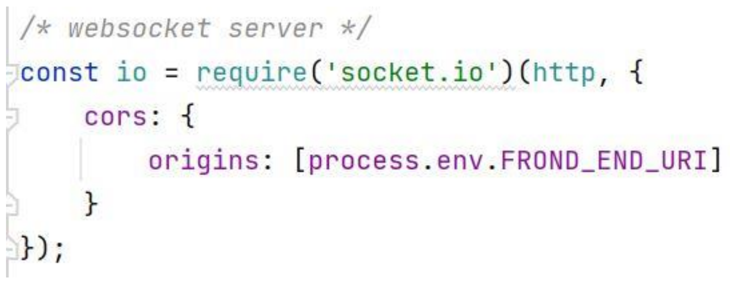
If there is no card to recognize the decision value is set to 0.

3.7.2 WebSocket

Between backend and frontend, a connection using WebSocket is established to send the card data and error messages from back- to frontend. WebSocket is a bidirectional, stateful protocol, meaning the connection between server and client is kept alive until it is closed. Thereby it is suitable for applications that process and display data at the client end that is continuously being sent by the backend server. Furthermore, a WebSocket is used for error messages in this application to be able to display the messages directly whenever an error, regardless of the current http request or response. Thus, multiple error messages can be displayed while calling an http endpoint and the data processing in the endpoint can continue and send a result back at the end.

3.7.2.1 WebSocket server

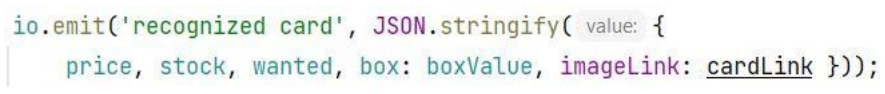
For establishing a WebSocket connection socket.io is used. In the backend a WebSocket server is created using the created Express server.



```
/* websocket server */
const io = require('socket.io')(http, {
    cors: {
        origins: [process.env.FROND_END_URI]
    }
});
```

Code 19. Initialization of the WebSocket server

The WebSocket messages are emitted as a string of a JSON object under a given event. In the following code showing an example of emitting a message, “recognized card” is set as the event and the message contains the card values and an image link.



```
io.emit('recognized card', JSON.stringify( value: {
    price, stock, wanted, box: boxValue, imageLink: cardLink }));
```

Code 20. Emitting a message via WebSocket

3.7.2.2 WebSocket client

In the frontend a connection to this webserver is established. To read the WebSocket messages event listeners for the respective events are added to the WebSocket. In the websocket service there is a generic function to add a listener to an event.

```
addListener(type: string, listenerFunction: (data: string) => void) {
  if (this.socket) {
    this.socket.on(type, listenerFunction);
  }
}
```

Code 21. Method in the websocket service to add an WebSocket event listener

This method can be called in the Angular components to add methods that are executed when a message with the given event is received.

```
this.websocketService.addListener( type: 'recognized card', this.cardRecognizedListener);
```

Code 22. Example of calling the addListener method to add an WebSocket event *listener in an Angular component*

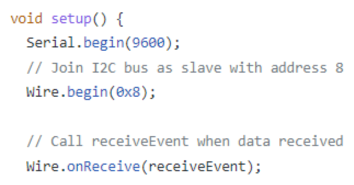
3.7.3 I2C

Raspberry pi communicates with Arduino via I2C. Raspberry pi takes images, sends images to the Computer Vision Algorithm, and gets the decision via MQTT. The decision is sent to the Arduino.

Raspberry Pi is configured as I2C Master and Arduino as a Slave.

```
from smbus import SMBus

addr = 0x8 #bus address
bus = SMBus(1) # indicates /dev/i2c-1
```

Code 23. Configuration of the Raspberry as the Master

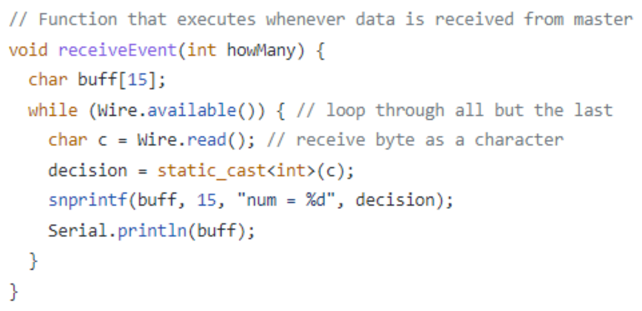


```
void setup() {
  Serial.begin(9600);
  // Join I2C bus as slave with address 8
  Wire.begin(0x8);

  // Call receiveEvent when data received
  Wire.onReceive(receiveEvent);
```

Code 24. Configuration of the Arduino as the Slave

Raspberry subscribes to a topic for status of the sorting machine (ON/OFF). It will then put the Arduino in the respective state. [Either START or STOP] Arduino will be put into deep sleep state on STOP state.

Raspberry also gets the decision (as an integer) from the same topic. The integer determines the position of one of the 4 slots where the card is to be dropped. The first three will be identified cards and the last slot would hold the unidentified cards.The Raspberry then writes the integer to I2C bus (to Arduino) Arduino performs necessary action, puts the card to the respective box.



```
// Function that executes whenever data is received from master
void receiveEvent(int howMany) {
  char buff[15];
  while (Wire.available()) { // loop through all but the last
    char c = Wire.read(); // receive byte as a character
    decision = static_cast<int>(c);
    snprintf(buff, 15, "num = %d", decision);
    Serial.println(buff);
  }
}
```

Code 25. Executing data in Arduino every time Master sends something.

The Arduino, on the other hand, sends a command when the suction system takes a card and waits on top of the camera so the Raspberry can proceed to take a picture and start the process.

3.8 Sequence diagrams

To illustrate the interaction between the different components of the application, the following paragraphs exemplarily describe two of the main processes: A user starting the card sorting machine by using the webpage and the card recognition process. For better clarity the diagrams are simplified and contain only the success cases, so error cases and the transmission and display of error messages are not included.

3.8.1 Starting the machine

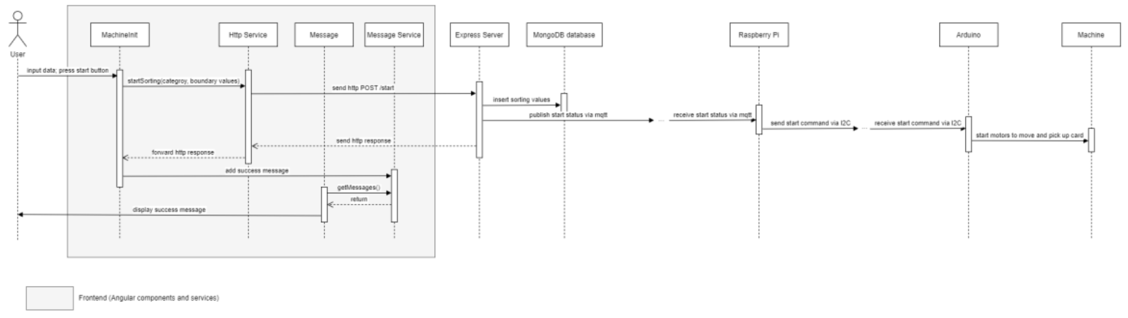
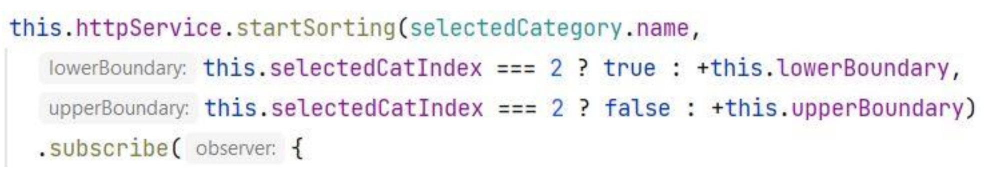

User
MachineInit
Http Service
Message
Message Service
Express Server
MongoDB database
Raspberry Pi
Arduino
Machine
input data; press start button
startSorting(categroy, boundary values)
send http POST /start
insert sorting values
publish start status via mqtt
receive start status via mqtt
send start command via I2C
receive start command via I2C
start motors to move and pick up card
send http response
forward http response
add success message
getMessages()
return
display success message
Frontend (Angular components and services)

Figure 29. Sequence diagram for starting the machine. [Click to enlarge]

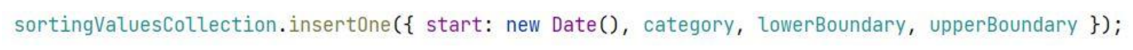
This sequence diagram displays the process of a user starting the card sorting machine by using the webpage. The process when stopping the machine is similar to the described process with the difference that no sorting values are included in the messages and method calls.
Starting point is the Machine Control page (http://localhost:4200) where a user chooses a category, enters the boundary values in the input fields (if the category isn't "Wanted") and presses the start button. When the button is pressed, the MachineInit component calls the startSorting function in the http service with the selected values.



```
this.httpService.startSorting(selectedCategory.name,
    lowerBoundary: this.selectedCatIndex === 2 ? true : +this.lowerBoundary,
    upperBoundary: this.selectedCatIndex === 2 ? false : +this.upperBoundary)
  .subscribe( observer: {
```

Code 26. Calling the startSorting function in the MachineInit component

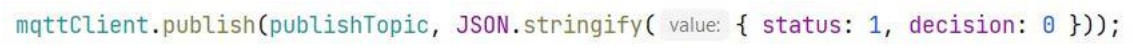
The http service sends an HTTP POST request to the /start endpoint of the Express server. In the endpoint the selected sorting values (category and boundary values) and a timestamp is stored in the MongoDB database.



```
sortingValuesCollection.insertOne({ start: new Date(), category, lowerBoundary, upperBoundary });
```

Code 27. Storing the sorting values with a timestamp in the database

The status 1 (“start”) is published via MQTT under the topic “statusChange”. Since there is no new recognized card the decision value in the MQTT message is set to 0.



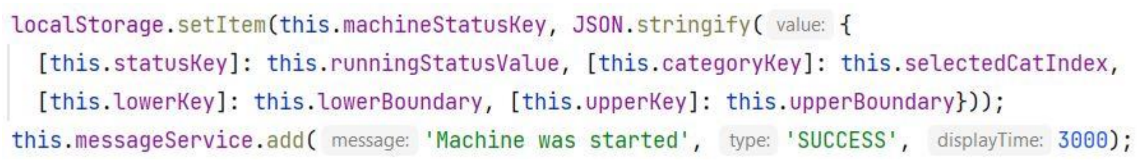
```
mqttClient.publish(publishTopic, JSON.stringify( value: { status: 1, decision: 0 }));
```

Code 28. Publishing the status value via MQTT

The MQTT message is received by the Raspberry Pi which has subscribed to the topic “statusChange”. It forwards this start command via I2C to the Arduino.

When the Arduino receives the start signal, it starts moving the motors of the machine to pick up a new card and place it above the camera.

During this process of forwarding the start command from the Express backend to the machine, the endpoint in the backend returns an HTTP response which is forwarded by the http service to the machineInit component. When this component receives the HTTP response with a success status code the sorting values are stored in the local storage and a success message is added to the message list in the message service.



```
localStorage.setItem(this.machineStatusKey, JSON.stringify( value: {
  [this.statusKey]: this.runningStatusValue, [this.categoryKey]: this.selectedCatIndex,
  [this.lowerKey]: this.lowerBoundary, [this.upperKey]: this.upperBoundary}));
this.messageService.add( message: 'Machine was started', type: 'SUCCESS', displayTime: 3000);
```

Code 29. Storing the sorting values in local storage and adding a success message

The message component, that gets the messages from the message service with the method getMessages(), detects this change in the message list and displays the newly added success message, so that the user can see it on the webpage.

3.8.2 Card recognition process

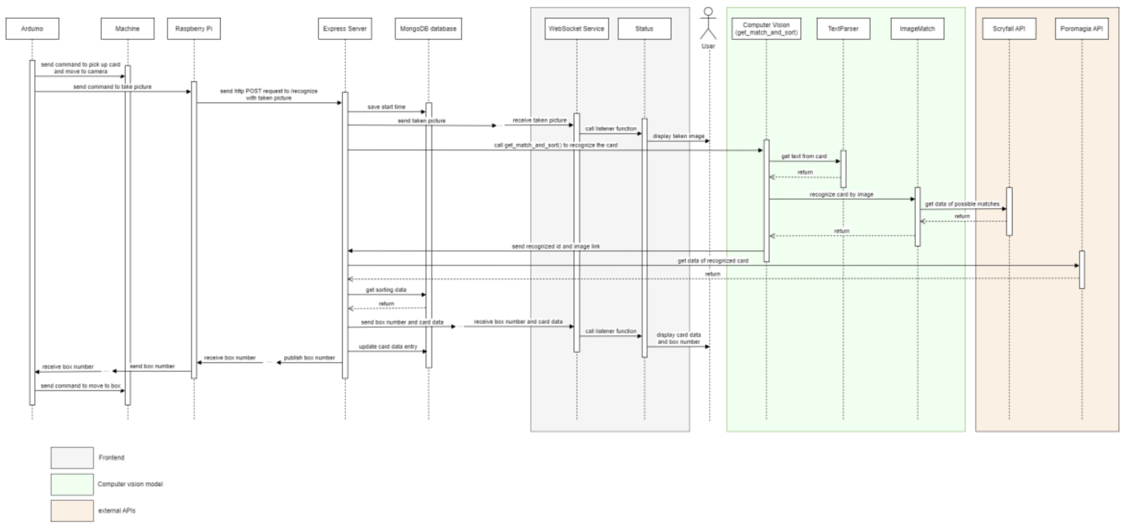


Figure 30. Sequence diagram for the card recognition process. [Click to enlarge]

This sequence diagram describes the process of taking a picture, recognizing the card, and sorting it accordingly. Starting point of the diagram is the Arduino sending the command to pick up a card and move it in front of the camera lens to the machine. This command is executed after the machine was started (described in the previous sequence diagram) or after the machine finished sorting a card.

After the machine has placed the card above the camera the Arduino sends the command to take a picture to the Raspberry Pi, and the Raspberry Pi takes the picture and sends it via HTTP request to the /recognize route of the Express server.

```
url = "http://54.83.117.198:3000/recognize"
files =  {'image': open('picture.jpg', 'rb')}
r = requests.post(url, files=files)
```

Code 30. Sending the taken picture via HTTP request to the Express server

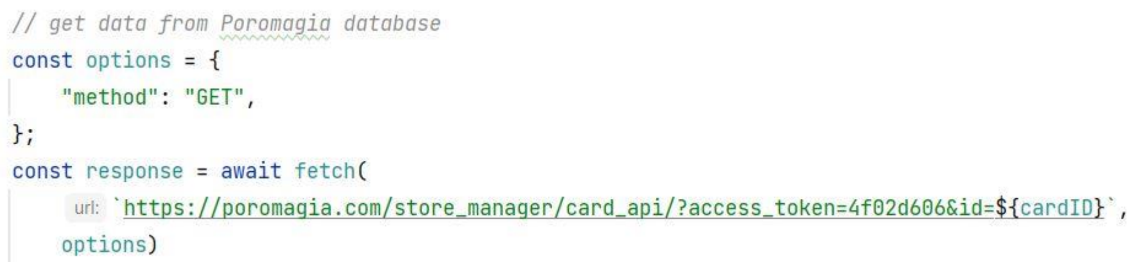
Before processing and recognizing the picture, the Express server stores the start time of the recognition process in the MongoDB database and sends the taken picture via WebSocket to the frontend. In the frontend the websocket service receives the message and calls the listener function of this event, which was added by the status component. In this function the HTML image element in the status component is set and therefore displayed on the webpage (http://localhost:4200/status).
At the same time the Express server calls the get_match_and_sort function of the computer vision model. Since the model is written in python, a child process is used.

```
// call the computer vision model to recognize the card
const childPython = spawn(process.env.PYTHON_VERSION, ['get_match_and_sort.py', cardImage]);
```

Code 31. Initializing the child process to call the computer vision model

In the get_match_and_sort function the TextParser class is called to get the texts on the card in the taken image. After the result is returned by the TextParser the ImageMatch class is called to recognize the card depending on the image. To get the images of the possible matches according to the card data the scryfall API is called. Then, computer vision is used to get the recognized card out of the matches from the image and the result is returned.

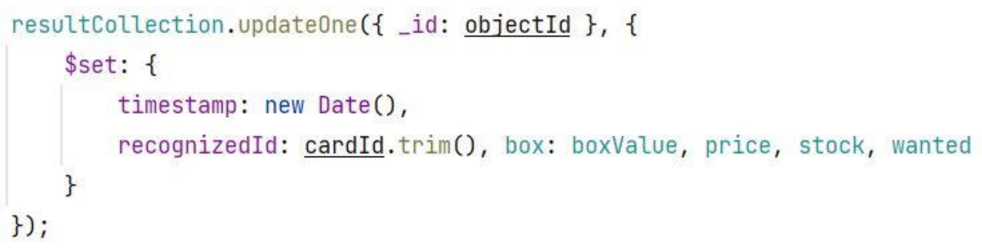
This process is described in more detail in chapter 3.5.4. The get_match_and_sort function returns the id of the recognized card and the link to the image of this card back to the Express server. When receiving this data, the Express server calls the Poromagia API with the id of the recognized card to get Poromagia’s data for the given card.



```
// get data from Poromagia database
const options = {
    "method": "GET",
};
const response = await fetch(
    url: `https://poromagia.com/store_manager/card_api/?access_token=4f02d606&id=${cardID}`,
    options)
```

Code 32. Calling the Poromagia API to get the card data

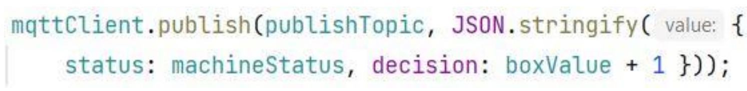
To get the sorting data the user entered before starting the machine the Express server reads the data of the last sorting entry in the MongoDB database. With the returned data from the Poromagia API and the database query the number of the box the card should be sorted in is determined. This number is sent via WebSocket to the frontend, where the websocket service receives the result and calls the listener function which was added by the status component to save the returned card values and display them and the image of the recognized card on the webpage.
At the same time the Express server updates the database entry of the current card by adding the data of the recognized card (price, stock, wanted and box values) and a timestamp for the end of the card recognition.



```
resultCollection.updateOne({ _id: objectId }, {
    $set: {
        timestamp: new Date(),
        recognizedId: cardId.trim(), box: boxValue, price, stock, wanted
    }
});
```

Code 33. Update of the current card’s database entry

Finally, the Express server publishes the number of the box in which the card should be sorted in via MQTT. The box value is increased by 1 in this message, because the boxes start at position 2 (the box with the cards to sort is on position 0 and the camera on position 1), so the box with the number 1 has the position 2, the box with number 2 has position 3, etc.



```
mqttClient.publish(publishTopic, JSON.stringify( value: {
    status: machineStatus, decision: boxValue + 1 }));
```

Code 34. Publishing of the recognized card's box number

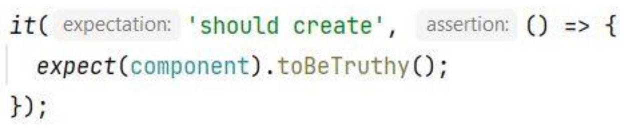
The Raspberry Pi receives this value via MQTT and forwards it via I2C to the Arduino.

When the Arduino receives the number of the box the current card should be sorted into, it starts the motors to move the card to the respective box and release it there. After this the process starts again from the beginning with picking up a new card.

3.9 Testing

3.9.1 Frontend tests

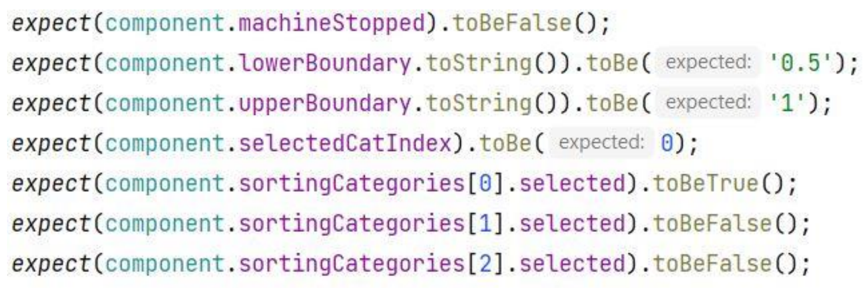
In the frontend the testing framework Jasmine is used for testing the application. In Jasmine tests can be written by using the global function it which defines a spec. The first parameter of this function is a string describing the test and the second parameter is a function containing the test code.



```
it( expectation: 'should create',  assertion: () => {
  expect(component).toBeTruthy();
});
```

Code 35. Example of a basic spec using the it-function

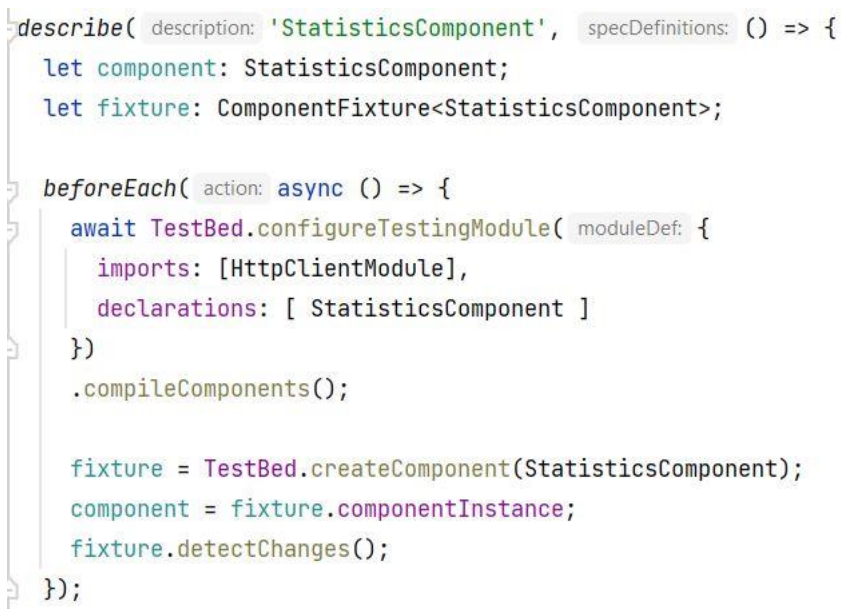
The expect function in combination with a matcher is used to compare an actual value to an expected value. In Jasmine there are many different matchers hat can be used for different use cases. In this application the matchers “toBe(…)”, “toBeTrue()” and “toBeFalse()” are mainly used.



```
expect(component.machineStopped).toBeFalse();
expect(component.lowerBoundary.toString()).toBe( expected: '0.5');
expect(component.upperBoundary.toString()).toBe( expected: '1');
expect(component.selectedCatIndex).toBe( expected: 0);
expect(component.sortingCategories[0].selected).toBeTrue();
expect(component.sortingCategories[1].selected).toBeFalse();
expect(component.sortingCategories[2].selected).toBeFalse();
```

Code 36. Examples of using the expect function

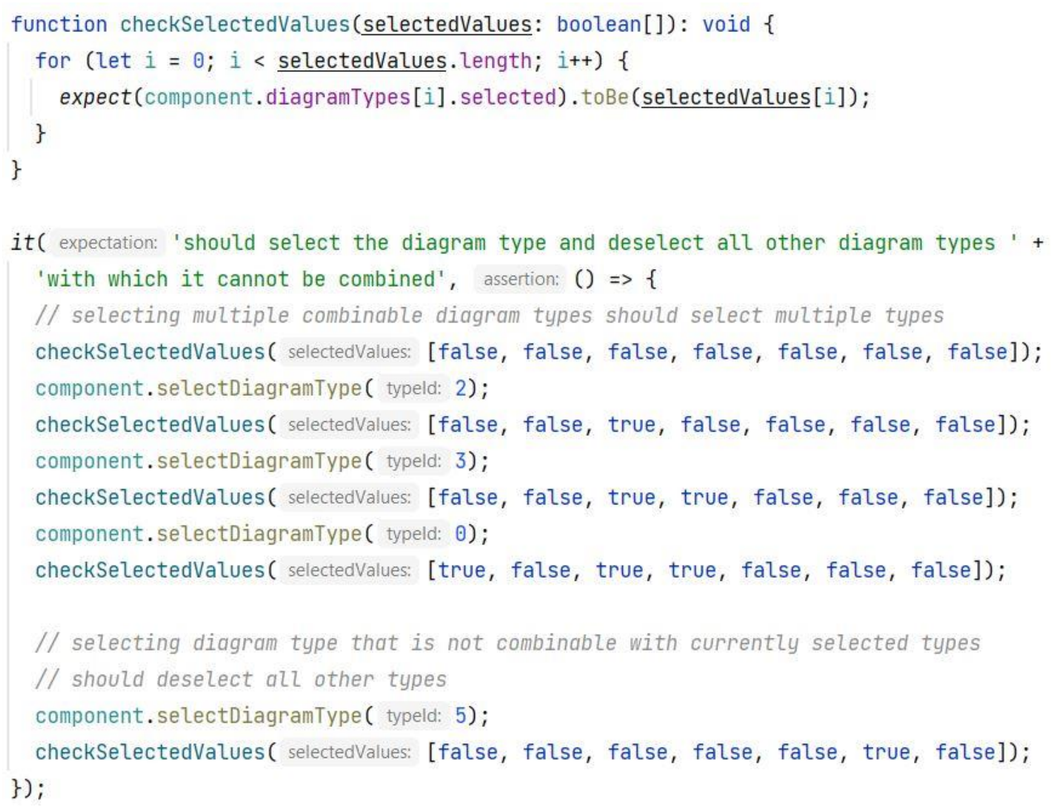
In general, multiple tests can be grouped using the describe function. In this function the global function beforeEach can be used to execute code before each spec. In this application it is mainly used for setting up the tests and creating the component. In Jasmine there are also other methods for executing code after each, before all or after all specs, which are not used in this application at the time of documentation.



```
describe( description: 'StatisticsComponent',  specDefinitions: () => {
  let component: StatisticsComponent;
  let fixture: ComponentFixture<StatisticsComponent>;

  beforeEach( action: async () => {
    await TestBed.configureTestingModule( moduleDef: {
      imports: [HttpClientModule],
      declarations: [ StatisticsComponent ]
    })
    .compileComponents();

    fixture = TestBed.createComponent(StatisticsComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
```

Code 37. Describe-function containing a beforeEach method to create the component before the execution of each spec

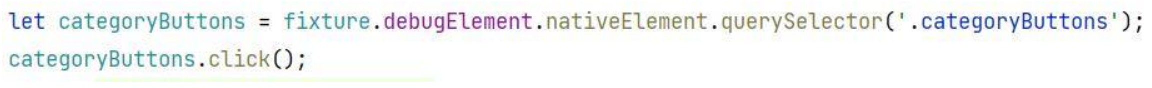
It is also possible to declare other functions in the describe function, for example helper functions for code that is executed in multiple tests.



```
function checkSelectedValues(selectedValues: boolean[]): void {
  for (let i = 0; i < selectedValues.length; i++) {
    expect(component.diagramTypes[i].selected).toBe(selectedValues[i]);
  }
}

it( expectation: 'should select the diagram type and deselect all other diagram types ' +
  'with which it cannot be combined', assertion: () => {
  // selecting multiple combinable diagram types should select multiple types
  checkSelectedValues( selectedValues: [false, false, false, false, false, false, false]);
  component.selectDiagramType( typeId: 2);
  checkSelectedValues( selectedValues: [false, false, true, false, false, false, false]);
  component.selectDiagramType( typeId: 3);
  checkSelectedValues( selectedValues: [false, false, true, true, false, false, false]);
  component.selectDiagramType( typeId: 0);
  checkSelectedValues( selectedValues: [true, false, true, true, false, false, false]);

  // selecting diagram type that is not combinable with currently selected types
  // should deselect all other types
  component.selectDiagramType( typeId: 5);
  checkSelectedValues( selectedValues: [false, false, false, false, false, true, false]);
});
```

Code 38. Example of a helper function and its usage in a spec

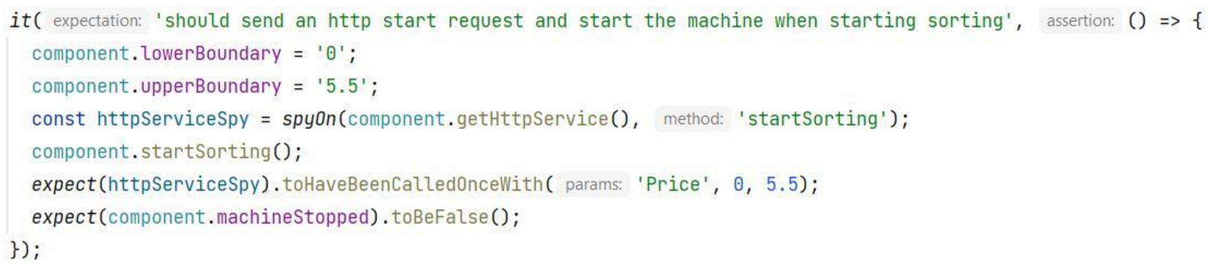
To get http elements the debug element of the component fixture can be used. With the query selector a class or id can be set to get the corresponding element. There are different methods that can be executed in this element to trigger an event, such as clicking the element.



```
let categoryButtons = fixture.debugElement.nativeElement.querySelector('.categoryButtons');
categoryButtons.click();
```

Code 39. Getting an HTML element and mocking clicking it

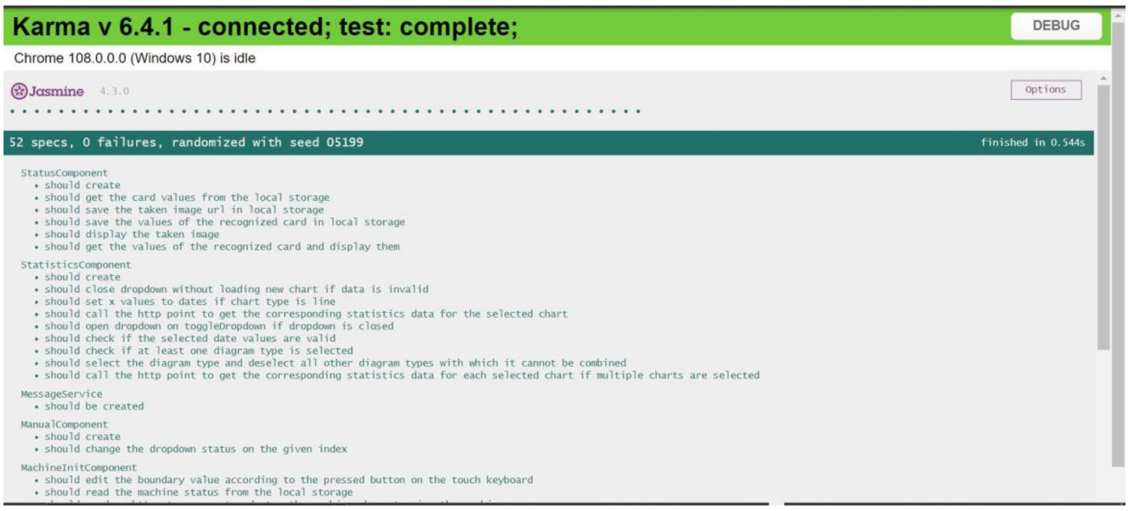
Spies are used to stub functions. It is possible to define the code that should be executed when the stubbed function is called and to test, how often and with which arguments it was called. In this application spies are mainly used for stubbing functions of the Angular services.



```
it( expectation: 'should send an http start request and start the machine when starting sorting', assertion: () => {
  component.lowerBoundary = '0';
  component.upperBoundary = '5.5';
  const httpServiceSpy = spyOn(component.getHttpService(), method: 'startSorting');
  component.startSorting();
  expect(httpServiceSpy).toHaveBeenCalledOnceWith( params: 'Price', 0, 5.5);
  expect(component.machineStopped).toBeFalse();
});
```

Code 40. Spy on the http service to check if the startSorting method was called with the correct arguments

In this application [7] each Angular component contains a testing file with the ending .spec.ts. Tests can be executed for one spec for one describe or all together. When executing the tests, a browser tab opens to display the result of the tests.



```
Karma v 6.4.1 - connected; test: complete;                                DEBUG
Chrome 108.0.0.0 (Windows 10) is idle
Jasmine 4.3.0                                                             Options
• • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • • •
52 specs, 0 failures, randomized with seed 05199                 finished in 0.544s

StatusComponent
  • should create
  • should get the card values from the local storage
  • should save the taken image url in local storage
  • should save the values of the recognized card in local storage
  • should display the taken image
  • should get the values of the recognized card and display them
StatisticsComponent
  • should create
  • should close dropdown without loading new chart if data is invalid
  • should set x values to dates if chart type is line
  • should call the http point to get the corresponding statistics data for the selected chart
  • should open dropdown on toggleDropdown if dropdown is closed
  • should check if the selected date values are valid
  • should check if at least one diagram type is selected
  • should select the diagram type and deselect all other diagram types with which it cannot be combined
  • should call the http point to get the corresponding statistics data for each selected chart if multiple charts are selected
MessageService
  • should be created
ManualComponent
  • should create
  • should change the dropdown status on the given index
MachineInitComponent
  • should edit the boundary value according to the pressed button on the touch keyboard
  • should read the machine status from the local storage
```

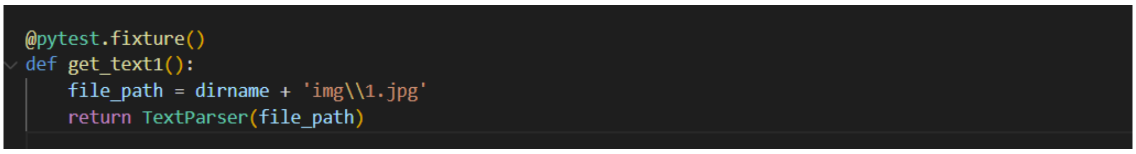
Figure 31. Result of the test execution in the browser

3.9.2 Computer vision algorithm tests

The artificial intelligence part of the backend code has two main classes, "image_recognition_class.py" and "text_parser_class", described above in section 3.3.

For the implementation of the tests, the python "pytest" library has been used. The pytest framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries. Fixtures are functions, which will run before each test function to which it is applied. Fixtures are used to feed some data to the tests such as database connections, URLs to test and some sort of input data. [9]

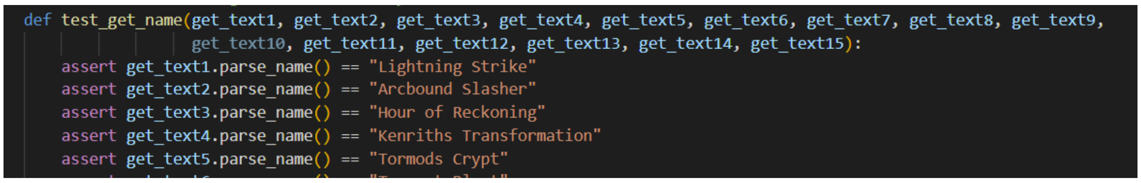
For TextParser class, fifteen test images have been taken from random collection cards to check if easyOCR methods applied to them work correctly.



```python
@pytest.fixture()
def get_text1():
    file_path = dirname + 'img\\1.jpg'
    return TextParser(file_path)
```

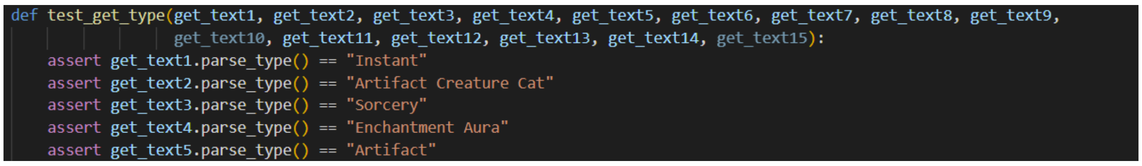
Code 41. Calling TextParser class with the image file path to create an instance of the class.

The following functions, which contains as parameter all the images, and runs the text extraction class in fifteen pictures, have been implemented.



```python
def test_get_name(get_text1, get_text2, get_text3, get_text4, get_text5, get_text6, get_text7, get_text8, get_text9,
                  get_text10, get_text11, get_text12, get_text13, get_text14, get_text15):
    assert get_text1.parse_name() == "Lightning Strike"
    assert get_text2.parse_name() == "Arcbound Slasher"
    assert get_text3.parse_name() == "Hour of Reckoning"
    assert get_text4.parse_name() == "Kenriths Transformation"
    assert get_text5.parse_name() == "Tormods Crypt"
```

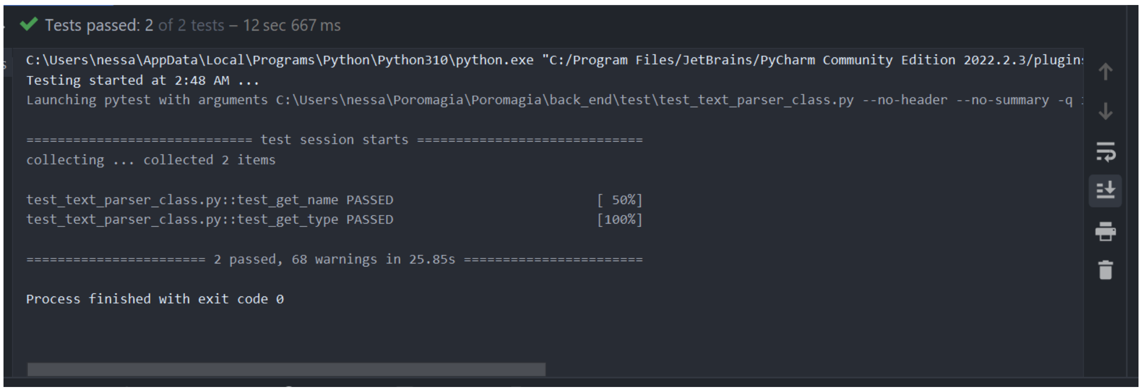
Code 42. Checking if the returned name is the same as the manually written one.



```python
def test_get_type(get_text1, get_text2, get_text3, get_text4, get_text5, get_text6, get_text7, get_text8, get_text9,
                  get_text10, get_text11, get_text12, get_text13, get_text14, get_text15):
    assert get_text1.parse_type() == "Instant"
    assert get_text2.parse_type() == "Artifact Creature Cat"
    assert get_text3.parse_type() == "Sorcery"
    assert get_text4.parse_type() == "Enchantment Aura"
    assert get_text5.parse_type() == "Artifact"
```

Code 43. Checking if the returned type is the same as the manually written one.

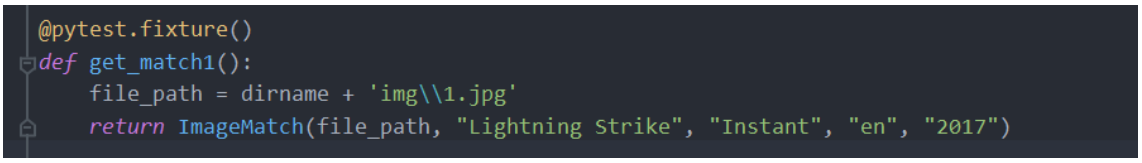
The console shows the results of the text extraction tests.



```
✔ Tests passed: 2 of 2 tests – 12 sec 667 ms
C:\Users\nessa\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm Community Edition 2022.2.3/plugin
Testing started at 2:48 AM ...
Launching pytest with arguments C:\Users\nessa\Poromagia\Poromagia\back_end\test\test_text_parser_class.py --no-header --no-summary -q

============================= test session starts ==============================
collecting ... collected 2 items

test_text_parser_class.py::test_get_name PASSED                          [ 50%]
test_text_parser_class.py::test_get_type PASSED                          [100%]

======================= 2 passed, 68 warnings in 25.85s ========================

Process finished with exit code 0
```

Code 44. Result of the text parser class tests

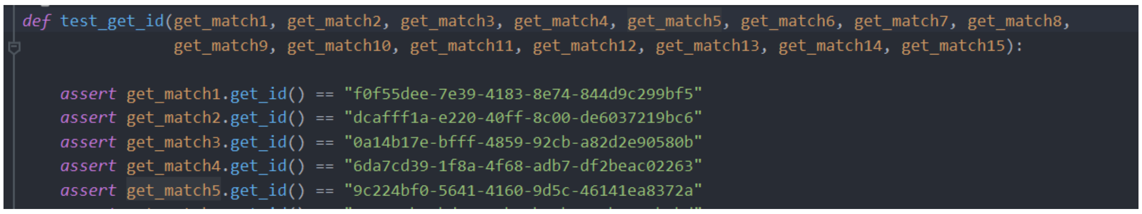
Regarding the ImageMatch class, the same fifteen test images have been used to recognize their Poromagia's ID. In this case, when calling the class, entry parameters need to be provided as it depends on the previous text extraction applied to get the name, type, language, and the year the card was made.



```python
@pytest.fixture()
def get_match1():
    file_path = dirname + 'img\\1.jpg'
    return ImageMatch(file_path, "Lightning Strike", "Instant", "en", "2017")
```

Code 45. Calling ImageMatch class with the image file path along with the card extracted type, name, language, and year to create an instance of the class.

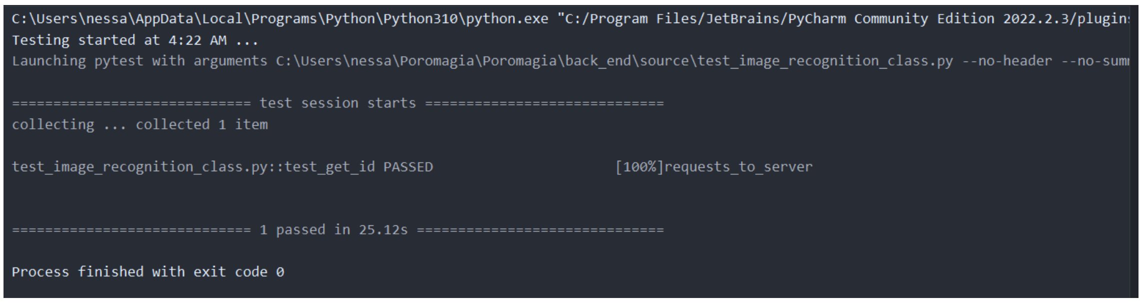
The following function, which contains as parameter all the images, creates multiple instances of ImageMatch class and after gets the ID of each one of the images.



```python
def test_get_id(get_match1, get_match2, get_match3, get_match4, get_match5, get_match6, get_match7, get_match8,
                get_match9, get_match10, get_match11, get_match12, get_match13, get_match14, get_match15):

    assert get_match1.get_id() == "f0f55dee-7e39-4183-8e74-844d9c299bf5"
    assert get_match2.get_id() == "dcafff1a-e220-40ff-8c00-de6037219bc6"
    assert get_match3.get_id() == "0a14b17e-bfff-4859-92cb-a82d2e90580b"
    assert get_match4.get_id() == "6da7cd39-1f8a-4f68-adb7-df2beac02263"
    assert get_match5.get_id() == "9c224bf0-5641-4160-9d5c-46141ea8372a"
```

Code 46. Checking if the returned id is the same as the manually written one.

The console shows the results of finding the image match tests.



```
C:\Users\nessa\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm Community Edition 2022.2.3/plugin
Testing started at 4:22 AM ...
Launching pytest with arguments C:\Users\nessa\Poromagia\Poromagia\back_end\source\test_image_recognition_class.py --no-header --no-sum

============================ test session starts ============================
collecting ... collected 1 item

test_image_recognition_class.py::test_get_id PASSED                      [100%]requests_to_server


============================ 1 passed in 25.12s =============================

Process finished with exit code 0
```

Code 47. Results of the image match class

3.10 List of materials

The list of materials used during the project development is described below. However, none of the parts used for the mechanical side have been included.

Material name	Price
Raspberry Pi 4, model B, 8GB	119 EUR
Arduino Mega 2560	50.95 EUR
Raspberry Pi 8.0 Mpix v2 kamera	35 EUR
Flex Cable for Raspberry Pi Camera	3.95 EUR
7 Inch Full View LCD IPS Touch Screen 1024*600 800*480 HD HDMI Display Monitor for Raspberry Pi - B	75.11 EUR
Official Raspberry Pi Micro USB Power Supply	10.20 EUR

4 Problems and Difficulties

The project has been very challenging because of its large size and many technologies. The first big challenge was the development of the software architecture. At the beginning, we had to translate the client's requests into technologies, since our duty was only to "create a machine that recognizes which card it is, and classify it by its price, among others", so we had to design a software system that allows us to make this sentence come true. We had problems determining the communication protocols, since we did not want to overengineer the system. It was also a big challenge to identify the technologies we wanted to use, since we used several programming languages, we had to think about whether their use was justified or if the process could be simplified.

On one hand, the biggest problem we had is related to the movement of the machine, since it is something that depended on another group and even if the software works, if the physical part of the project does not work, there is no final product. In the end, they failed to implement their part, and members of our group tried to make it work, and despite almost making it work, the delivery point of the project was very close, and we didn't make it on time.

Another big challenge was the implementation of the backend on the cloud server, as it took a lot of autonomous research, and a lot of debugging time, as installing all the dependencies without problems took a lot of time, plus we had never used AWS before. It should also be considered that we work with the free tier phase, so we had to save memory on the server with everything we can, so the structure of the software had to be adapted to this fact.

On the other hand, displaying the photo taken in real time on the raspberry on the front end of the page, and subsequently displaying the result of the recognized chart along with which box it goes to, was another big challenge. We had to wait for the backend in the cloud to work perfectly to be able to implement that functionality, and it was something that took several days since we had never done it before. What took the longest was to send the picture from the raspberry Pi to the server in the cloud, where the module multer was used.

Finally, the development of the computer vision model was another big challenge, as we did not know whether to do a machine learning model or to implement OCR, i.e., real-time object recognition. In the end, the second option was the one chosen since training a machine learning model with so many images were the first option, but it took a lot of time, and the biggest downside it had is that card recognition amount did not increase when new cards were published unless the system was trained again. Besides, getting the data to train such a large model was difficult. Therefore, we created the computer vision model, but it was also a challenge to create the approach for the text extraction and image recognition classes.

5 Future Prospects

For the future, there are several aspects that remain to be implemented or that can be further developed. A major aspect to improve is the movement of the machine. The mechanical part does not work perfectly, since the mechanical arm could not move due to the weight it was carrying, and the switches did not work effectively since the only working switch model was no longer in stock. Therefore, mechanical engineering students should take the project to fix the movement of the machine.

Regarding the machine software, one of the aspects to improve is the computer vision model, which can be changed to a deep learning model, which can reduce the margin of error, as it is difficult to recognize cards that do not have the same standard appearance model. It would also improve the processing time, as now it is around thirty seconds and with the use of a trained neural network model it could be a few seconds. In addition, the implementation of a cloud server with more resources would be a big positive change, since it could have a dedicated GPU which, when making use of the object recognition library, would reduce the time considerably, at least ten seconds. The illumination of the picture taken from the chart should also be considered. When taken upside down, the contrast often causes poor results which makes the photo not easily readable by the computer vision model, so probably led lights or a photo that illuminates the image from below would be the best option.

Another aspect to improve would be the implementation of the on/off button on the web page, which even though the change of state is published by MQTT, it does not really cause a great impact on the movement of the machine when it is already moving, since the proper tests could not be performed due to the lack of movement of the machine. Therefore, it is necessary to check the state of the machine when the user pauses the software.

6 Summary

In conclusion, we have implemented an IoT project for the Poromagia company, which wanted to create a machine that sorts collector cards in real time to avoid the manual work. The project was satisfactory, because although the mechanical part of the project needs further development, the software was implemented satisfactorily, and on time.

A variety of challenges were overcome, the development of the software architecture as well as the implementation of the cloud server, which also brought with a lot of background research work. Also, several communication protocols have been used, such as I2C, MQTT and Web Sockets, in addition to using four programming languages, which are Python for the computer vision model, JavaScript for the back end of the server, Typescript for the front end of the web page and C++ for the lower communication, which is for the movement of the machine.

In conclusion, the team has learned a lot not only in software and hardware development, but also in testing and deployment, and in meeting the needs of a company and its requirements.

7 References

1. TechTarget, „Definition. MongoDB,“ TechTarget, August 2020. [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/MongoDB.

2. Microsoft, „What is TypeScript?,“ Microsoft, [Online]. Available: https://www.typescriptlang.org.

3. C. Deshpande, „Introduction To Angular Service and Its Features,“ Simplilearn, 8 August 2022. [Online]. Available: https://www.simplilearn.com/tutorials/angular-tutorial/angular-service.

4. MQTT.org, „MQTT: The Standard for IoT Messaging,“ MQTT.org, 2022. [Online]. Available: https://mqtt.org/.

5. Google und contributors, „What is Angular?,“ Google, 18 Februrary 2022. [Online]. Available: https://angular.io/guide/what-is-angular.

6. StrongLoop, IBM und contributors, „Express,“ StrongLoop/IBM, 2017. [Online]. Available: https://expressjs.com/.

7. OpenJS Foundation und Node.js contributors, „About Node.js,“ OpenJS Foundation, [Online]. Available: https://nodejs.org/en/about/.

8. Your first suite," Jasmine Behavior-Driven JavaScript, 2017, [Online]. Available: https://jasmine.github.io/tutorials/your_first_suite

9. Pytest, ytest: helps you write better programs,” Holger krekel and pytest-dev team, 2015, [Online]. Available: https://docs.pytest.org/en/7.2.x/

10. Amazon, ,,What is Amazon EC2?” Amazon Web Services, 2022, [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

11. MongoDB, ,,Aggregation Operations,” MongoDB, Inc, 2022, [Online]. Available: https://www.mongodb.com/docs/manual/aggregation/

12. Magic: The Gathering, ,, Wikipedia, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Magic:_The_Gathering

13. Jained Ai, ,,EasyOCR” jaided.ai, 2022. [Online]. Available: https://www.jaided.ai/easyocr/

14. OpenCV, ,,About”, OpenCV team, 2022. [Online]. Available: https://opencv.org/about/

15. AWS, ,, Amazon Linux 2”, Amazon Web Services, Inc. 2022. [Online]. Available: https://aws.amazon.com/amazon-linux-2