UCLouvain

École polytechnique de Louvain

Gesture Recognition

Online gesture management system with microservices

Author: **Thibaut PIQUARD**
Supervisor: **Jean VANDERDONCKT**
Readers: **Nicolas BURNY, Suzanne KIEFFER, Nathan MAGROFUOCO**
Academic year 2018-2019
Master [120] in Computer Science

First of all, I would like to thank the Professor Jean Vanderdonckt for his involvement throughout the realization of this master thesis. I would also thank Nathan for his help on the recognition algorithms and especially Nicolas for his availability and feedback during the entire preparation of this thesis.

Lastly, I would like to thank my friends and especially my girlfriend for all the moral support you gave me. Thanks for everything, I finally got it.

Abstract

Since its emergence in the 1980s, the gesture recognition sector has been constantly expanding with new approaches, algorithms and tools. This diversity has made it possible to carry out a lot of research but has also generated a demand for a more global and flexible solution. This solution would make it easier to integrate these new tools and make them work together. To meet this demand, we have developed a software based on microservices architecture. This application aims to assemble existing programs and their benefits while removing any restrictions they may have. Its microservices architecture allows it to be easily modified and to be a solid basis for this objective. We will see how this solution is applicable to this sector and we will consider the future work to complete the application.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context of the problem

Gesture recognition is a subject that is beginning to have a certain seniority in the IT world. Since its first appearance in computer science (from glove-based control interfaces in the 80's, to the pen-controlled PDA in the 90's), Researchers have been constantly improving the algorithms, the infrastructure and the applications of these techniques. Over time, a lot of new approaches and visions have been brought to the domain, which translates in a significant progress. But with this tremendous number of enhancements comes an ever-increasing number of software products and tools. The gesture recognition scene has become saturated with a large number of approaches and tools that do not integrate well with each other. The domain of gesture recognition has a bright future in front of it with new applications emerging regularly but navigation through the existing tools and making them working together appropriately represents a major challenge to its adoption.

1.2 Motivations

A **global gesture management system** is therefore a huge advantage in this domain. Indeed, it would not only make it possible to standardize a large number of tools and make them work together but it would also make possible to integrate new tools in a seamsless fashion. Many methodologies and approaches used in the domain of gesture recognition are related to a single software. For this reason, a methodology that could be used in a larger number of applications represent a major improvement compared to the current situation.

Moreover, the majority of the tools available online were created a long time ago, and the languages or technologies used for their development are not suitable

for new developments anymore.

Finally, microservices architecture is increasingly present in today's software. This approach opens up new possibilities and has some advantages over more traditional architectures such as monolithic architecture. It is therefore interesting to look at it and propose a solution using this technology.

1.3 Objectives

Given the current context of gesture recognition, the objective of this thesis is to analyze, design and implement an online application allowing the creation, manipulation and management of gesture sets. This solution has to be accessible online for all potential users.

An important feature of the solution to be developed is flexibility. For that, this application will be built using microservices architecture. This solution will then have to be able to accommodate new modules and be easily modified to cope with new approaches/innovations in the gesture recognition domain.

Last but not least, due to the easily customizable nature of the solution, an objective is to propose different improvements that can be applied to it.

1.4 Overview of the structure

In the first section of this thesis, we will specify the state of the art in the field of gesture recognition and first begin with a theoretical background related to gesture creation, management and recognition. This part aims to explain and show the main structures used in gesture recognition and more particularly in the proposed solution. The second part of this state of the art will provide a global overview of the recent tools that have emerged and to highlight their features but also their limitations. The last part will focus on microservices. We will look at their characteristics, their advantages as well as their drawbacks.

The second section of this thesis will provide an analysis of the software architecture and its implementation choices. This section will contains a UML class diagram describing the data structures, various activity diagrams depicting the main prominent workflows in the application, and several implementation choices such as the platform security.

The next section will be a description of the application itself. It will consist of a complete overview of each page and each functionality of the software.

We will then reflect upon the functionalities/improvements that have been designed but that have not ultimately been part of the final application. Each of them will be described, commented on why they should be planned for future use.

Finally, a conclusion will be drawn on what has been achieved during this thesis. It provides a final summary of what this new approach brings and the possible future of such an application.

Chapter 2

State of the art

Before explaining in detail the existing gesture management system, we will try to give and define some technical terms in order to facilitate the subsequent understanding of the thesis. Then, we will provide an overview of some of the existing tools for gesture recognition. The next step will be to explain the functioning and architecture of the existing gesture management system. Lastly, we will explain microservices and microservices architecture that will be used for the conception of the application in this thesis.

2.1 Theoretical background

At the core of each gesture recognition algorithm is the concept of gesture. But what is exactly a gesture ? What is it made of and how could we identify a gesture or differentiate between two ? In an attempt to answer these questions, this section will cover the various underlying structures typically used to model a gesture in a bottom-up approach, starting by the smallest structure to the largest.

2.1.1 Point

In gesture modeling as well as analytic geometry, a point is a primitive notion upon which the geometry is built, meaning that a point cannot be defined in terms of previously defined objects. While its name seems self-explanatory, it is the building block upon which all other structures are based : it represents a single point on the 2D (or 3D) plane, characterized by an abscissa (horizontal axis or x), an ordinate (vertical axis or y) and (in the case of a 3D plane) an applicate (depth axis or z) value. If the point is define using quaternions, the point can have a fourth value name w.

2.1.2 Stroke

A stroke is "the set of points produced by the drawing implement between the time it contacts the surface and the time it breaks contact "[23].The first point of the stroke (referred as the starting point) and the last point of the stroke (referred as ending point) can be highlighted.

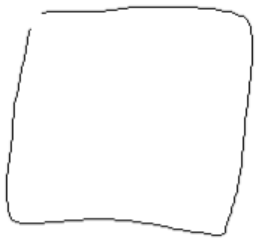

Figure 2.1: Representation of a stroke representing a square

An important characteristic to notice when modeling a stroke in any environment is the number of *sampling points* used. This number corresponds to the amount of points that will actually be recorded and processed while drawing, and it can drastically influence the computation time as well as the gesture recognition process.

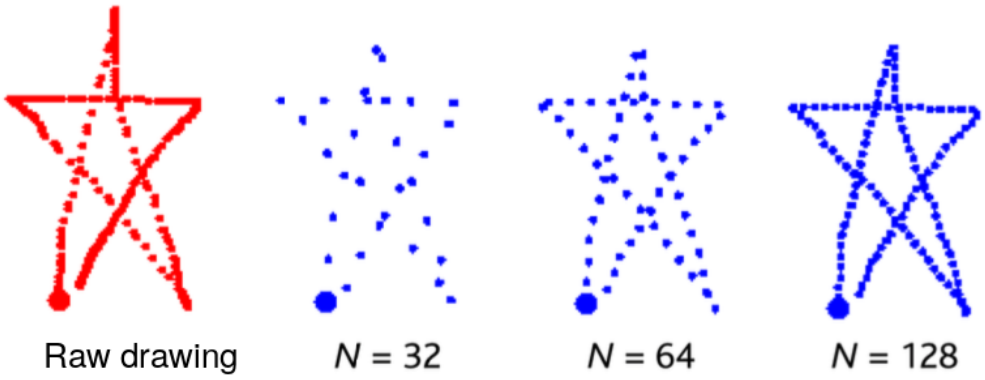

Figure 2.2: Impact of the number of sampling points in strokes.

2.1.3 Gesture

A gesture is an higher-level structure in gesture modeling. *Jacob O. Wobbrock, Andrew D.Wilson and Yand Li* [10] define a gesture as "a set of candidate points" and these candidate points are "sampled at a rate determined by the sensing

hardware and software". Each set of candidate points sampled belongs to a stroke. Furthermore, a gesture can be characterized by certain distinct properties relating to the number of strokes and their orientation:

- The **number** of strokes : a gesture is said to be *unistroke* (or single-stroke) if it is composed of a single stroke; or *multistroke* otherwise.
- The **orientation** of the strokes : a gesture is said to be *unidirectional* if all its strokes are directed in the same direction; or *multidirectional* otherwise.

we can also distinguish two particular types of gestures corresponding to different combinations of the aforementioned properties : a **flick** is an unistroke unidirectional gesture, and a **mark** is a multi-orientation gesture.

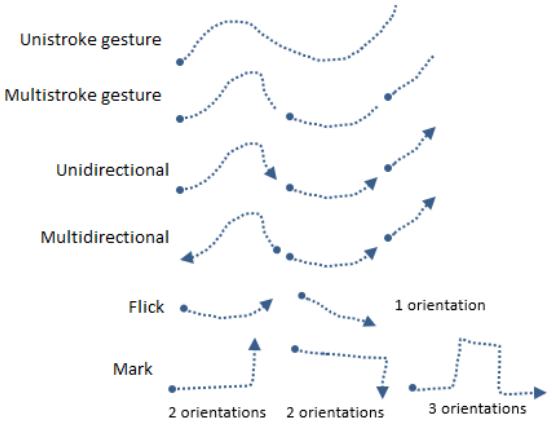

Figure 2.3: Different types of gestures. flick and mark.

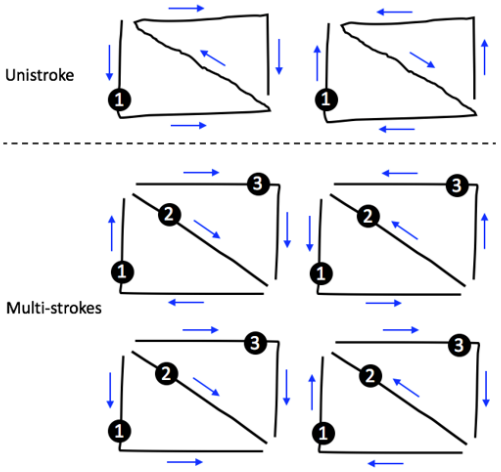

Figure 2.4: Different ways of decomposing a gesture.

Gestures can also be of different dimensions such as 2-dimensional (2D), three-dimensional (3D) or two and a half dimensional ($2D^{1/2}$).

2D gestures are existing in the 2-dimensional space $\mathbb{R}^2$, and are thus characterized by multiple pairs of parameters (x,y) (respectively horizontal axis and vertical axis). An example of such 2D gesture are the gestures sampled by a touch-sensitive device.

3D gestures are existing in the 3-dimensional space $\mathbb{R}^3$, and are characterized by a combination of triplets of coordinates (x, y, z) (respectively horizontal axis,

vertical axis and depth axis). For example, these gestures may correspond to a gesture performed through the air and sampled by an accelerometer or using any optical sensing device.

$2D^{1/2}$ gestures are the projection of 3D gestures on a 2-dimensional plane. These gestures are thus also occurring in the $\mathbb{R}^2$ space, and use pairs of parameters (x,y) to depict their position in the environment. An example of such gesture is depicted in the figure 2.5

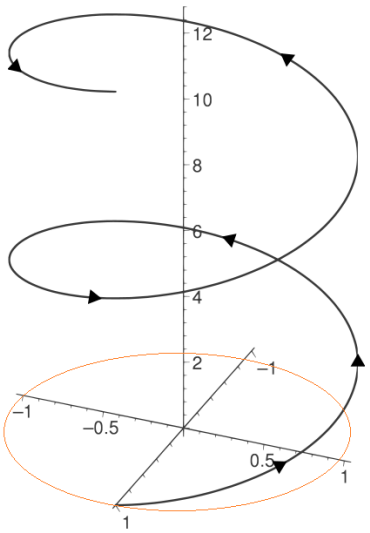

Figure 2.5: 3D gesture (black) "flattened" to obtain a $2D^{1/2}$ gesture (orange).

2.1.4 Gesture Set

To define a gesture set, we have to define first what is a gesture class. A gesture class corresponds to the label given to one specific type of gesture (e.g., “square”). Several gestures can therefore have the same class. A gesture set is therefore simply a set of predefined gesture classes.

2.1.5 Invariants

In addition to the different characteristics of the elements composing a gesture, several properties are used to further describe the gesture itself. These properties are the following :

- **Iso-chronic**, which states that the different points composing the strokes of the gestures are separated by the same amount of time (i.e. the difference between each timestamp is a constant value).

- **Iso-metric**, which states that the different points composing the strokes of the gestures are separated by the same distance (i.e. the distance between each point is a constant value).

- **Iso-parametric**, which states that the different points composing the strokes of the gestures are characterized by the same set of parameters.

- **Position invariant** stating that the gesture is independent of any translation applied to its coordinates; meaning that its recognition by a gesture recognizer will not be affected by its position.

- **Scale invariant** stating that the gesture is independent of any scaling factor (increasing or decreasing) applied to it; meaning that its recognition by a gesture recognizer will not be affected by a multiplication factor on its coordinates.

- **Rotation invariant** stating that the gesture is independent of any rotation applied to it; meaning that its recognition by a gesture recognizer will not be affected by its orientation.

2.2 Existing tools in the gesture recognition sector

The domain of gesture recognition is quite heterogeneous. A lot of researches have been carried out in the past and a significant number of tools have been created. In this section, we will focus on the most recent contributions and will try to come up with common points for improvement.

2.2.1 iGesture

iGesture [5] is a framework based on three main components: a recognizer, a management console and evaluation tools for testing and optimizing algorithms. The management console of the iGesture framework is a Java Swing application. This application can test gestures, define new gestures and create test sets.The graphical user interface is based on the Model-View-Controller (MVC) design pattern and can be easily modified. The iGesture framework has been helpful in implementing and testing existing algorithms as well as developing new algorithms. But this framework hard-code their gestures in their own format and cannot be captured and managed by different stakeholders in parallel.

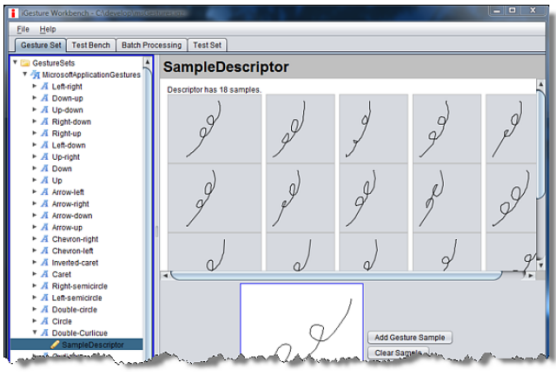

Figure 2.6: iGesture Workbench

2.2.2 GestureAnalyzer

GestureAnlyzer [11] is visual analytics system supporting identification and characterization of gesture patterns from motion tracking data. It enables UI designers to capture video sessions in which participants elicit gestures and annotate them afterwards to classify gestures. GestureAnalyzer enables rapid identification of frequent gesture patterns from a large data set without having prior knowledge of the data. But there are some limitation in this approach : it is limited to a specific gesture data format where the gestures start and end with a natural standing pose and since GestureAnalyzer is first and foremost a video annotation software, it partially supports the "Classify" stage, but does not store any gesture in any form other than video, thus preventing them from reusing the results later.

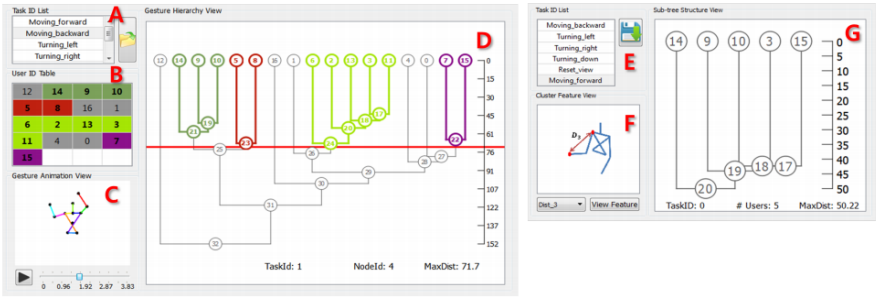

Figure 2.7: The GestureAnalyzer interface

2.2.3 KinectAnalysis

KinectAnalysis [17] is a system for recording, analyzing and sharing multimodal interaction elicitation studies by analyzing gestures produced by a Microsoft Kinect

(associated with Kinect software environment, such as XDKinect). It enables researchers to collect data on popular gestures, speech and multimodal interactions. This approach shows some limitations. The first is that implementation is subject to the limitations of the Kinect sensor and relies on Kinect's default skeletal tracking, which means rather basic gesture detection, no tracking of fingers, eye gaze, etc. The second limitation relates to completeness and expressiveness of the KinectScript language.

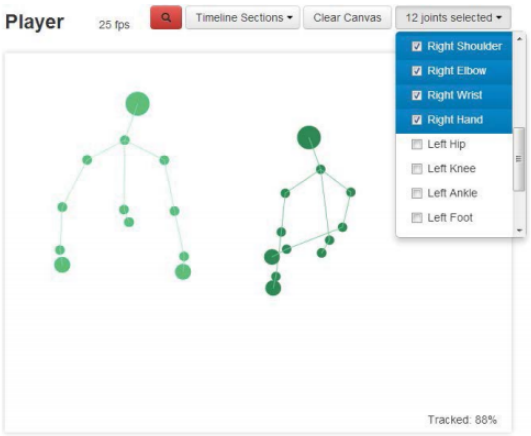

Figure 2.8: Recording of two skeletons; lower body parts exluded

2.2.4 GestureWiz

GestureWiz [25] prototyping environment provides designers an integrated solution for gesture definition, conflict checking, and recognition in nearly realtime using Wizard of Oz or crowdsourcing set-ups. It consists of three main components : a Requester UI for recording gestures and conflict checking; the Worker UIs (Original and 1 vs. 1) for recognizing recorded gestures; and the GestureWiz library to be used in applications that shall be enhanced with gesture recognition capabilities. First, they use the Requester UI in recored mode and capture a set of templates gestures. Then, the gesture set can be tested using the built-in conflict checker to resolve ambiguities and adjust the design of the gestures. Subsequently, arbitrary commands within the application can be mapped to any of the defined gestures and, finally, test gestures can be streamed for recognition. GestureWiz has numerous real-world applications, especially in situations with a lack of required hardware or mixed modalities.

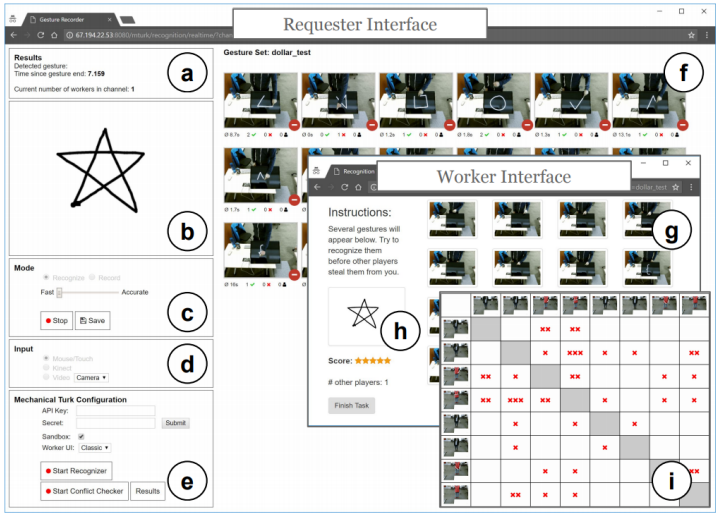

Figure 2.9: GestureWiz's requester and worker UI

2.2.5 Buzzi

Maria Claudia Buzzi, Marina Buzzi, Barbara Leporini and Amaury Trujillo [8] developed a software for capturing unistroke, single path gestures from three visually impaired participants each working on smart-phone simultaneously in the same usability laboratory. It's a simple wireless capture gesture system, based on the client-server model and using primarily web technologies. Their system consists of three components: a server dashboard for parameterize, monitoring, and visualizing gestures, a custom web application on a smart phone for capturing the parameterized gestures and a web server interacting between the two. The main purpose of this software is to serve their research in the first place but is still very interesting because of its innovative approach to capturing touch-based gestures from several users at the same time on mobile devices using web technologies.

2.2.6 AGATe

AGATe (AGreement Analysis Toolkit) [32, 31] is a C# toolkit running on MS Windows and using the .NET 4.5 framework. The toolkit reads data organized in a matrix format so that each referent occupies one column and each participant occupies one row. AGATe computes agreement, disagreement and coagreement rates for selected referents, and reports significant effects of selected referents over agreement rates at p = :05, :01, and :001 levels of significance based only on gesture ID (no capture and classification) and counts. Since it is not connected to the rest

of the workflow, the various stakeholders, such as the UI designers, are responsible for defining and conducting the method beforehand and to consolidate the results afterwards.

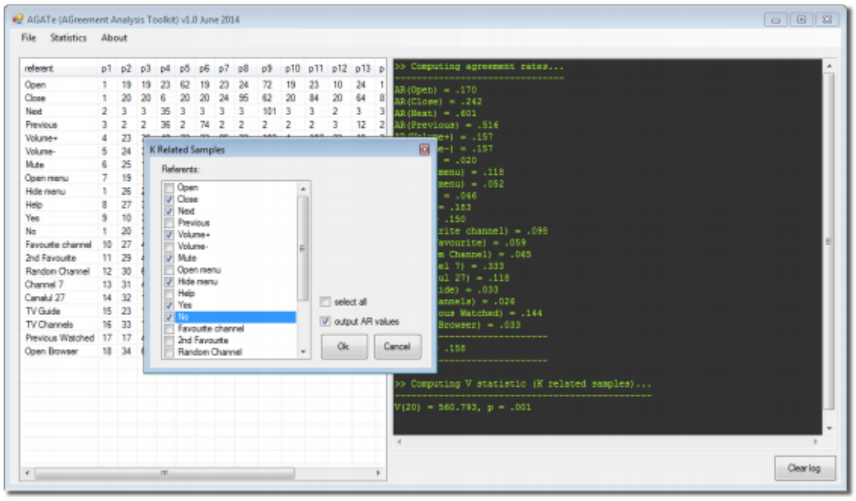

Figure 2.10: Agate User Interface

2.2.7 GestAnalytics

GestAnalytics [7] is a software for analyzing hand gestures captured by video by offering the following original features that go beyond a simple video annotation tool: simultaneous video monitoring, taxonomical tagging and filtering. Dividing long videos into meaningful small parts, matching the questionnaire data, examination of videos and the verification process are very time-consuming processes, GestAnalytics assists researchers in these problematic areas. But the software has some limitations. First, GestAnalytics is developed in Unity3D, which is a game engine and not optimized for video viewing, thus the performance of the software can increase if it is moved to another platform. And secondly, it is focusing on only one type of gesture and one medium, without considering the rest of the study.

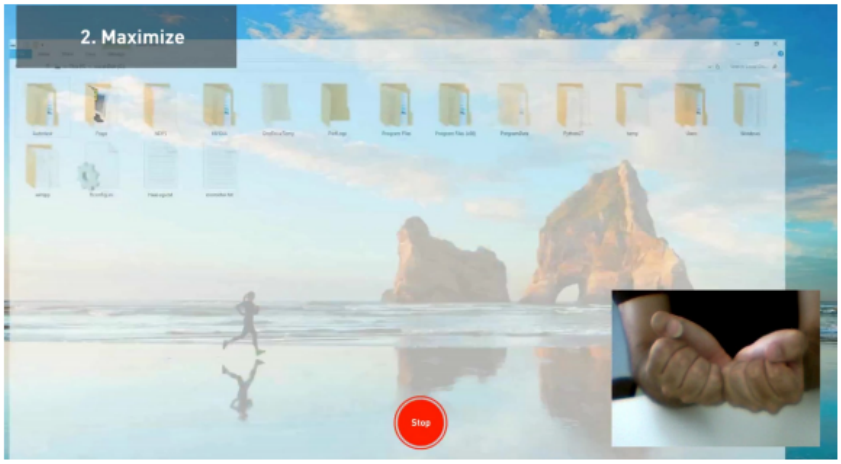

Figure 2.11: Recording screen of GestAnalytics

2.2.8 CrowdSensus

CrowdSensus [3] is a crowd-based software that enables experimenters to analyze the results of elicitation studies more efficiently than manually using subjective human judgment and automatic clustering. It demonstrates the possibility to use a crowd of non-experts in conjunction with automatic clustering algorithms to successfully analyze the results of an elicitation study. However, the symbols they elicited for the experiment were text strings and they suspect that richer multimedia symbols will require more time to analyze.

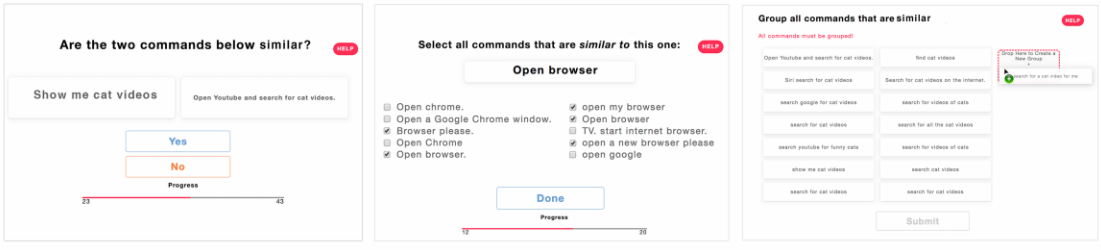

Figure 2.12: The direct comparison, lit comparison and grouping interface

2.2.9 GECKo

GECKo (GEsture Clustering toolKit) [15] is a software to assist a human operator in editing and correcting the cluster partition initially generated by an automated

clustering procedure. The goal is to increase the visual similarity of gestures exhibiting the same number of strokes, stroke directions and stroke orderings, and to visually highlight dissimilarity in any of these factors. Once finished, GECKo reports within- and between-subject agreement rates enabling users to audit the various clustering results. The main limitations of GECKo are its restricted input format files and the fact it is designed for one user on one machine (it's an executable file) so you can't directly share gestures observations with other researchers.

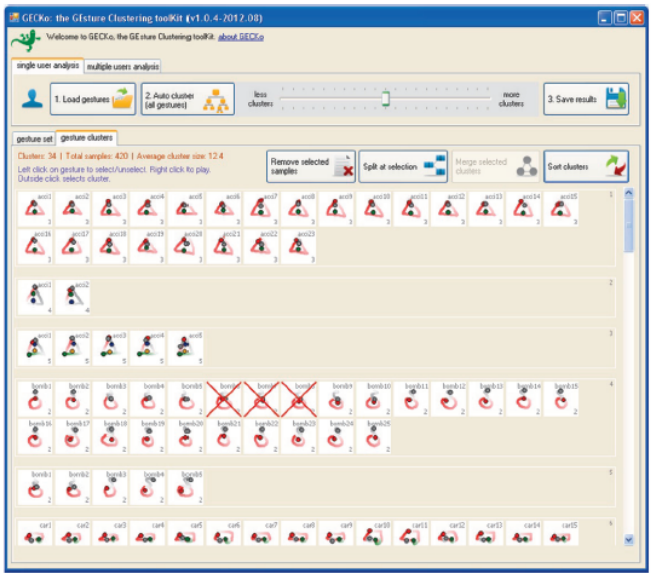

Figure 2.13: Gecko User Interface

2.2.10 GHoST

GHoST (Gesture HeatmapS Toolkit) [30] is an open source software that compute gesture heatmaps and chromatic confusion matrices for both user-dependent and independent scenarios. It allows the user to examine the articulation properties of gestures via heatmaps, which are color map visualizations of how gestures vary in time and space. The ultimate goal of heatmap analysis in Human Computer Interaction (HCI) is to pinpoint usability problems, to allow designers to understand their users' interaction patterns and improve their designs. The limitations of GHoST are similar with these of GECKo : we also have to give to GHoST a strict structure and it is designed for one user on one machine (also an executable file).

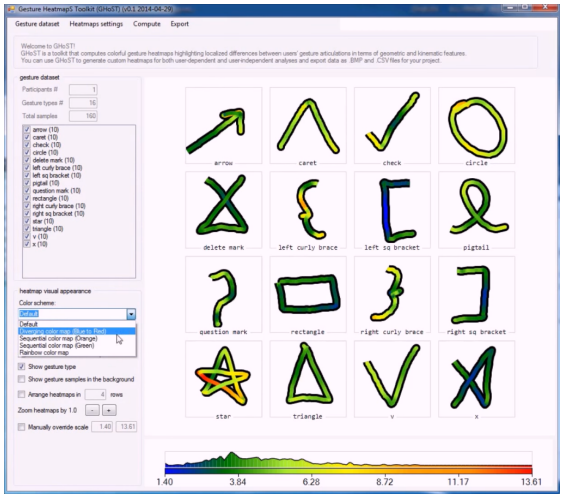

Figure 2.14: Ghost User Interface

2.2.11 GREAT

GREAT (Gesture RElative Accuracy Toolkit) [29] is an application for calculating relative accuracy measures for a set of gestures. There are twelve relative accuracy measures. These measures capture what happens during stroke gesture articulation, and are therefore more revealing than overall recognition accuracy. Once a gesture set has been processed, several reference gestures are determined by first computing the average gesture from the set, and then finding which articulated gesture is comparable to that average. Again, like GHoST and GECKo, GREAT suffers from the same limitations, i.e a very specific structure format to follow and an inability to work in a team online due to the fact that it is an executable file.

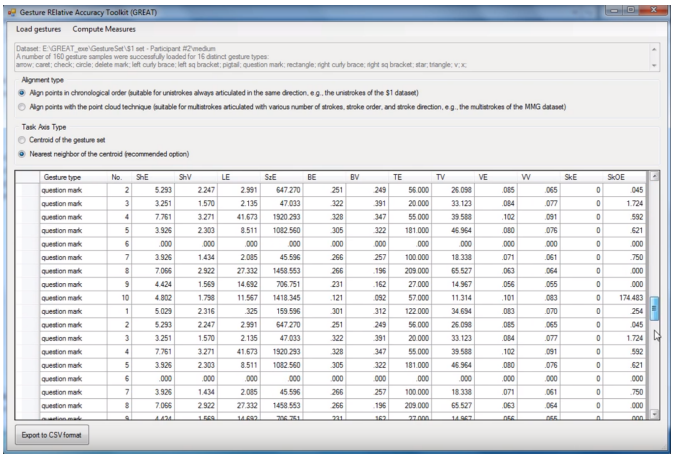

Figure 2.15: Great User Interface

2.2.12 Common features

After reviewing all these tools and contributions for the gestures recognition sector, we can highlight some features that are problematic or at least considered as limitations in these approaches. First, the data import and export format. It is sometimes impossible to export data after analysis or hard to import data with specific format files and structure. Secondly, the types of gestures handled are limited. Indeed, some software only manage a small number of types of gestures (unistrokes, 2D, etc). In addition to that, these softwares do not connect the entire process (create a gesture set, analyze a gesture set, train a gesture algorithm, ...) and they don't work together (as we have seen for GECKo, GREAT and GHost that are great tools but only work individually). A modular and complete system for gesture recognition is therefore of great interest.

2.3 Gestman

In this section, we will describe Gestman, a gesture management system built by *Romain Dizier and Zacharie Kerger*[13, 16].
Zacharie and Romain made the same observation about the gesture recognition domain and came to the same conclusion that there is a lack of a complete global system for gesture management. The purpose of their work was similar to the one of this thesis but they decided to work in a monolithic architecture. First, it is interesting to define Gestman. Gestman is a cloud-based application for stroke-gesture sets that implements management of gesture data from acquisition to processing, analysis and classification, composition of gesture sets, and routing of gestures to the next stages of the development process. We will see in detail each of these steps :

- **Gesture acquisition ;** One motivation of Gestman is to be able to manage a large number of different types of gestures. So Gestman handles contextual information as the user, the input device or the environment. Gesture samples can be user-dependent or user-independent, 2-D or 3-D gestures, ... This choice allows designers model gestures with more freedom than static representations. It is important to specify that Gestman allows the creation of samples (their acquisition is done via an HTML canvas) as well as the import via an external file.

- **Processing :** GestMan provides the following set of primitives : acquire, clear, save, recognize, translate, rotate, and scale. These adjustable manipulation functions are made available in order to offer the user a variety of user-friendly design tools.

- **Analysis and classification :** After an user has created a gesture sample, several features are computed based on its attributes. These attributes (thirteen in total) have been defined by Rubine [28] and are similar to the attributes computed by GREAT [29]. Gestman can also process gestures to maintain some invariant properties we have seen in section 2.1.5. With regard to classification, Gestman implements four recognizers : $1 [10], $P [20], !FTL and !NFTL [12] but GestMan also supports adaptability by enabling other recognizers, algorithms, and web services to be integrated.

- **Composition of gesture sets :** To support gesture integration, gestures can be composed by appending their strokes, decomposed from multi-strokes to unistrokes, and recomposed at any level. Gesture samples are stored in classes which are themselves stored in sets. Each modification of a sample or a class is saved in a log file history so the user can keep a trace of them.

- **Routing of gestures :** After manipulating the gesture sets, Gestman allows you to export them with all the parameters calculated/provided. It is possible to export only one class or to export the entire set directly. This will be exported in JSON format which can be reused later on Gestman if the user want to re-import this class. It is also possible to export the recognizers source code from Gestman in the form of a single JavaScript file.

So looking at Gestman and its capacities, we can conclude that it is indeed a complete system. Nevertheless, its completeness generate a complex architecture. Gestman has been built upon a monolithic architecture and modifying or adding new functionalities has become an expensive and complex task. In a domain as changing and evolving as gesture recognition, it is crucial to be able to integrate new features to the application easily in order for it not to become obsolete. So we will try a new approach with a more recent architecture that seems to suit better to our needs.

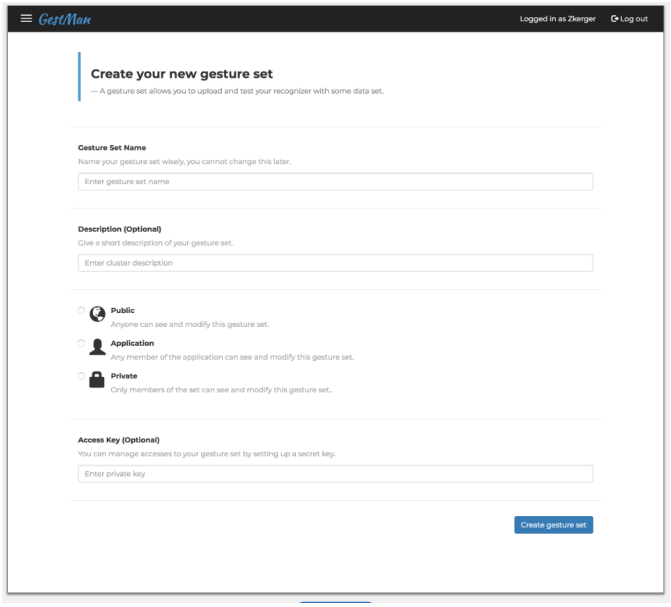

Figure 2.16: Gestman User Interface

2.4 Microservices as a software development technique

In this section, we will describe in details the microservices architecture. The purpose of microservices is to solve the issues encountered when working with monolithic architectures. We will first give a brief introduction to this monolithic architecture and describe the various issues encountered when working with this type of software design pattern. We will then explain what microservices architecture is and how it solves the problems inherent to monolithic architecture. But since there are no magic solutions, we will also highlight the drawbacks of using a microservices architecture.

2.4.1 Monolithic architecture

Monolithic architectures have been the main approach to software design for a long time and still today, a large number of applications run through a monolithic approach.

In software engineering, a monolithic application describes a single-tiered software application in which the user interface and data access code are combined into a single program from a single platform. The design philosophy with this

architecture is that each step needed to perform a function is done by the application.

Such architecture can have advantages but also drawbacks. *Dragoni*[18] gives a list of issues faced when working with monolithic architectures :

1. Difficulty to maintain and evolve large-size monoliths due to their complexity.
2. Dependency hell : adding more and more libraries results in inconsistent systems.
3. The need to reboot the whole application for any change in one module, which results in considerable downtimes.
4. Usually sub-optimal deployment due to conflicting requirements on the constituent model's resources (memory-intensive, computational-intensive,...).
5. Scalability limit : the usual strategy for handling increasing inbound requests is to create new instances of the same application and to split the load among these instances. However, it could be the case that the increased traffic stresses only a subset of the modules, making the allocation of the new resources for the other components useless.
6. Lock-in technology that requires developers to use the same languages and frameworks of the original application.

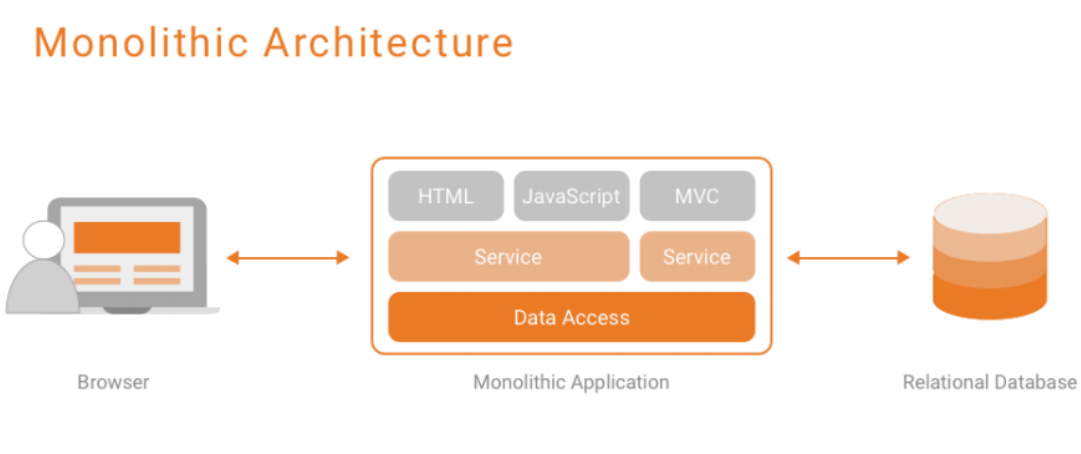

Figure 2.17: Illustration of a monolithic architecture.

2.4.2 Microservices architecture

Definition of Microservices

Microservices are a software development technique, a variant of the service-oriented architecture (SOA), an architectural style that structures an application as a collection of loosely coupled services. But in view of its recent emergence, the state of the art lacks consensus on the definition of microservices, their properties and their modeling techniques. We provide in this section some definitions encountered in the literature in order to have a global overview :

- *Lewis and Fowler*[14]

  "An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP-based API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services which may be written in different programming languages and use different data storage technologies."

- *Dragoni and al.*[18]

  "A microservice is a cohesive, independent process interacting via messages."

- *Thones* [26]

  "A microservice is a small application that can be deployed independently, scaled independently and tested independently and that has a single responsibility."

- *Pautasso, Zimmerman, Amundsen, Lewis and Josuttis* [9]

  "Microservices can be viewed as a substyle refining Service-Oriented Architecture with additional constrains (e.g less emphasis on central components such as Enterprise Service Bus (ESB), etc.), amended with recent advances in services realization and deployment (Continuous Integration, DevOps, ...). They can be characterized by the following properties: small, messaging enabled, bounded by contexts, autonomously developed, independently deployable, decentralized, built and released with automated processes."

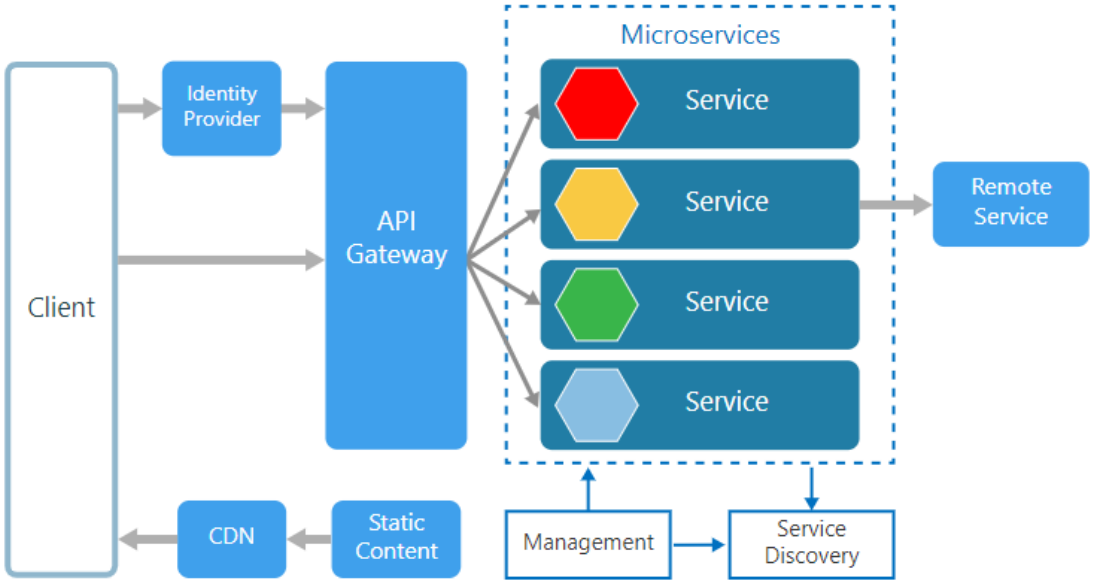

Figure 2.18: Illustration of a microservices architecture.

2.4.3 Characteristics of Microservices

Now that we have tried to define what a microservices architecture is, it is interesting to highlight its main characteristics. We will therefore talk here about modularity, smart endpoints and dumb pipes, scalability, data management and governance decentralization.

Modularity

Increased modularity is a direct impact of the use of microservices. It is indeed possible to have a modular monolithic application but in microservices architecture, systems are decomposed into a number of microservices, forcing developpers to apply some kind of modularity to their solution. Moreover, this modularity makes it easier to apply changes to a component and redeploy this component without shutting down the entire system.

Smart endpoints and dumb pipes

In a monolithic application, the various components are all in the same process, and are separated by a simple function call. But in a microservices environment, components are separated by hard boundaries. The fact is that this independent division requires a good communication system between the components. This

pattern is called *Smart endpoints and dumb pipes*. *Smart endpoints and dumb pipes* is a design principle that favors basic, time-tested, asynchronous communication mechanisms over complex integration platforms.[24] So we focus on complex independent microservices that talk to each others with using simple and reliable communication system.

The 'dumbness' of pipes refers then to the lightweight messaging protocols used for simple and the endpoints are qualified as "smart" because they encapsulate all the resources required for them to function effectively.

With this approach, small applications are easier to maintain and decoupling makes it easier to scale. Indeed, with simple communication between microservices, it is easy to add a new one or to increase traffic without overloading it.

Scalability

When we are using a monolithic application, it's quite easy to quantify the efficiency of the system. But evaluating the efficiency and achieving greater efficiency in a large ecosystem of microservices is much more difficult. Indeed, the more microservices in a system, the lower the efficiency of one microservice will impact the entire system. Developers must therefore make compromises between availability and complexity when they decompose a system in microservices. These compromises can be illustrated by the *Scale cube* [2].

As see in figure 2.19, microservices are themselves a form of scalability [6], the Y-axis. Y-axis scaling breaks the application into its components and services. The X-axis is traditional load-balance scaling. The Z-axis takes a similar approach to the X-axis running identical copies of code across multiple servers but in this approach, each server is responsible for only a subset of the data (data partitioning). When X-axis and Z-axis scaling improve the capacity and availability of the application, they can in the same time increase its complexity. The purpose of the Y-axis and therefore microservices scaling is to deal with such complexity increase and so use in a more efficient way the other two axis.

This can be illustrated with horizontal scaling. In a monolithic application, we can manage the increase in demand by running multiple instances of the application. This is the X-axis or horizontal scaling. But in a microservices architecture, we are able to duplicate only the microservices that need to be duplicated and not the entire application. Microservices thus make a much more effective use of the horizontal scalability pattern.

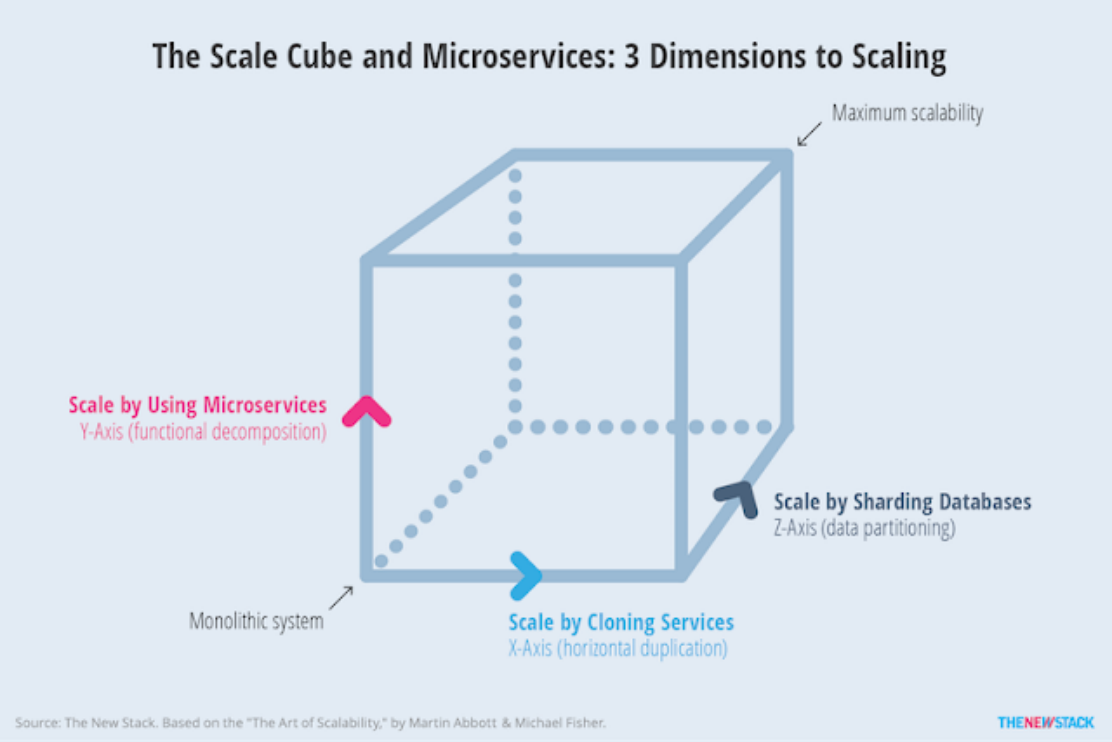

Figure 2.19: Illustration of the Scale cube.

Data management and Governance decentralization

Monolithic applications are typically backed by a large relational database, which defines a single data model common to all application components. In a microservices approach, such a central database would prevent the goal of building decentralized and independent components. Each microservice component should have its own data persistence layer.

2.4.4 Drawbacks of microservices

Even if microservices architectures show many interesting advantages, it is still important to be aware of their disadvantages and the complications they can bring. In this section, we will highlight some of these disadvantages and try to understand where they come from.

Developing complexity

The first drawback to be managed when deciding to develop an application using microservices architecture is an higher complexity compared to a classic monolithic

architecture [33]. Indeed, developing microservices requires a well-considered divisional work in advance. And since everything now is an independent service, developers have to carefully handle requests travelling between the microservices. This structure can be confusing and a poor decomposition of the system or inefficient interactions between modules can lead to a forced restructuring.

Increased network communications

This drawback is a direct consequence of the basic idea of microservices architecture. Indeed, as we have already seen, the division of the application in microservices requires to handle the communication between them. And even if we use *dumb pipes*, the number of requests between microservices is significantly higher compared to the number of request for a classic monolithic application.

This strong increase of requests is commonly called the *chattiness* of the network. The developer must do a compromise between chattiness of his network and number of services in his application. Having very chatty services can be considered bad and the developer will have to consider that perhaps they belong in the same service.

Furthermore, network calls made between microservices are relatively slow and brittle compared to local calls [19].

Security challenges

Compared to a monolithic application, microservices pose enormous security challenges due to the increases in inter-service communication over the network. All of these interactions create an opportunity to outside entities to gain access to the system [33].

Moreover, since depending on a single DB is not possible if data live within different microservices, extra effort is required to keep it consistent and have transactions. Moreover, multiple databases and transactions management systems can quickly become complex [19].

Chapter 3

Software architecture and Implementation

Now that we are done with the theoretical concepts and the state of the art for gesture recognition, we can get to the core matter of thesis, namely the design a gesture recognition system managed with a microservices architecture.

First of all, we will study the UML diagram, its assumptions as well as its choices of microservices divisions. Next, we will look at the workflow modeling and how to juggle with these microservices and their functioning. Then, we will follow up on a global vision of the system architecture and check if it meets the microservices architecture criteria. This chapter will conclude with specific implementation choices for the gesture recognition management system and why there are relevant.

3.1 UML modeling

The first thing to do before embarking on this type of application is to have a clear and precise vision of the structures we will use later. That can be done with what we call an *Unified Modeling Language* (UML). Of course, the first iteration of such a diagram is never definitive. An UML can be updated when structural changes are made to the project. That can be due to unforeseen circumstances, unsuccessful tests or simply implementation needs. This section will display the final version of the UML diagram (see figure 3.1) and will explain each structure of it.

3.1.1 UML classes

- User:
  It represents an user of the application but only at the identification level. So the User class has only necessary attributes like email and password. There is also a *Method* attribute that allows to know the user's connection way (like *Google* or *Facebook*). This attribute is an implementation need which will be discussed in more detail in the implementation choices section.

- Profile:
  It represents personal characteristics of a user of the application. It contains attributes such as his first and last name, his address and his organization (such as a school, an individual, a private company,...)

- Point:
  It's the smallest part of a Gesture Set. This class represents a single point and its 2,3 or 4 dimensional coordinates respectively x for abscissa, y for ordinate, z for depth and w as fourth dimension when representing gestures with quaternions. The class also has a timestamp attribute.

- Stroke:
  The purpose of this class is to display a stroke object made up of points.

- Gesture Sample:
  It characterizes a specific sample of a gesture class. There are many attributes. Simple ones like name, description but also a creation and last modification dates. The *ParticipantId* attribute allow the user to know which participant made which sample. *Gesture_class_id* is the link between a sample and its mother class. The Strokes attribute is only the list of its strokes. The "properties" attribute represent the specific features that a sample can have (see section 2.1.5)

- Gesture Class:
  It characterizes a specific class of a gesture set. This class is attached to its samples with *Gesture_samples_id* and is linked with a gesture set with the *Gesture_set_id* attribute. Its last attributes are its name, its description and the creation and last modification dates

- Gesture Set:
  Last class of our UML diagram, Gesture Set, is bound to gestures classes with the *Gesture_classes_id*. It is described using a name, a description but also a *UserId*. This last attribute is used to link the gesture set to its owner. There are also the creation and last modification dates attributes.

3.1.2 UML splits

As you can see in the figure 3.1, UML diagram is split into three parts. These part represent microservices splits and database splits. Indeed, as you will see in section 3.3, each microservice will handle its own database. These divisions must be carefully considered because our modules will be directly impacted by these decisions. So here we have three modules:

- Gestures (Yellow):
  It is the biggest one in terms of number of classes but its split did not seem a good option. Indeed, an additional division in this service would have generated a significant increase in the number of requests between these two new microservices. As we discussed in section 2.4.4, developer must be able to make compromises to avoid having too much communications over the network. Moreover, even if the number of classes is higher than others, the complexity of this microservice stay reasonable.

- Authentication (Red):
  This module possess only one class but the related microservice is quite complex because it manages the entire "user connection" part. We will talk in further detail about it in section 3.4.

- Profile (Green):
  This is the smallest microservice composing the application which purpose is to manage user's personal data.

The split between authentication and profile may seem confusing but it will be explained in section 3.4

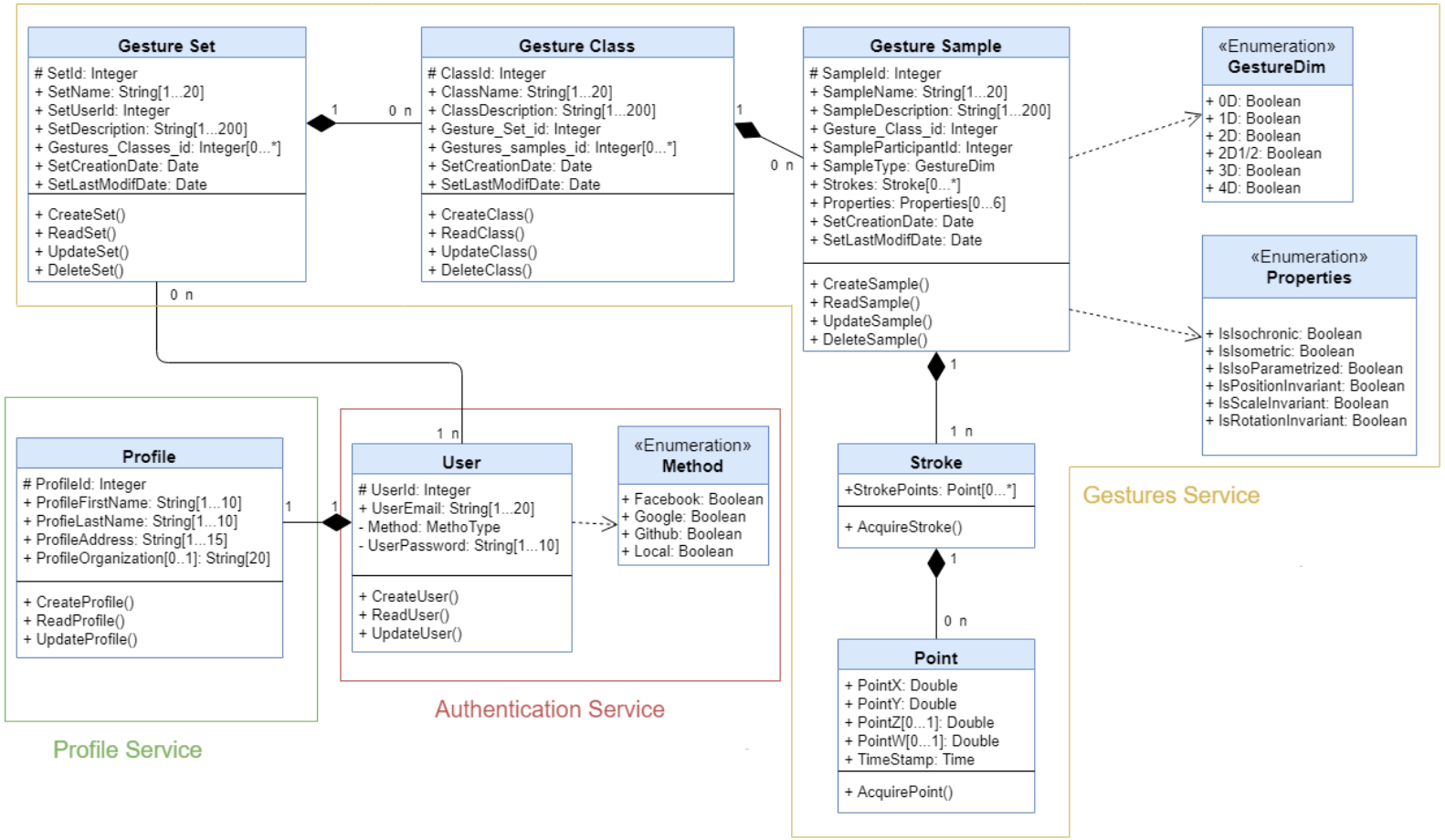

Figure 3.1: Final UML class diagram

3.2 Workflow modeling

In addition to the UML class diagram, there is another important building block in the conception of an application. This block will no longer focus on data structures but on the dynamic aspects of the system. Just as the data had UML diagram, that can be done with what we call an *Activity Diagram*. These activity diagrams will represent different workflows of progressive activities and actions. We will focus on three features and develop them with the help of activity diagrams: the creation and modification of a new user, the upload of a new gesture set by a user and the deletion of a gesture class.

3.2.1 Creation and modification of a new User

In this first activity diagram, we focus on user creation and modification. As you can see in figure 3.2, the activity diagram is divided in several groups. These groups are the User and all the microservices that are used during the process. Here in the present case: Gateway, Authentication and Profile.

First the user send his sign up request with credentials. The Gateway will then process the information and send these credentials to the good microservice, i.e the Authentication microservice. This service will verify the correctness of these credentials and create a new user if this is correct. It will also create a JSON web token(JWT). This token is a key feature in the application security but we will talk about it in detail in the section 3.4. All we have to know is that thanks to this JWT, when Gateway will send it to Profile service, this Profile service can retrieve the id of the newly created user and inject it when it create a new profile. Then Gateway transfer the JWT to the user. The user's browser will save it automatically and send it at each new action. When the user will complete the profile form and send these modification to the Gateway, Gateway will transfer all of this to the Profile service. Thanks to the JWT, Profile service can retrieve from which user the request comes and update user's profile.

3.2.2 Upload of a new gesture set

In this second activity diagram, we focus on the upload of a full gesture set. Before starting, it is relevant to say that we make two hypotheses on our workflow model. First, the JWT mentioned in the first activity diagram will be present but implicit for a practical reason. Secondly, it is the same for the ids of gesture sets , classes and samples. When it's says "link to", that means the Gestures service uses these ids to link them to each other.

These assumptions stated, we can look at the figure 3.3. We can go directly to the "submit file" phase as we already seen a process very similar to a log in process in the first activity diagram. When the Gateway receives the request, it receives with this a JSON file containing a full gesture set. This file can be quite heavy for a request. So in order for the gateway to accept such a load, we have to increase manually the limit. For security reason, we don't want that this limit be also increased in the Gestures microservice. Indeed, the Gateway doesn't have direct access to databases so if it crashes, that does not corrupt stored data. So it's up to the gateway to deal with the JSON data. It will first transfer gesture set attributes to the Gestures service. Then each gesture classes and for each of them all gesture samples. For each of these transfer, the Gestures service will check if fields sent to it are correct. Indeed, if the JSON file don't follow a standard structure or is corrupted, the fields will be incorrect. If this is the case, Gestures service returns an error. With this approach, we can easily find where is the error in the JSON file. If the fields are correct, Gestures service will create respectively gesture set, classes and samples and link them to each other.

3.2.3 Deletion of a gesture class

In this third and last activity diagram, we focus on the deletion of a gesture class and. Like the second activity diagram, the JWT is sent each time but this is implicit, except for the first transfer where JWT is mandatory to explain the process.

The figure 3.4 shows the activity diagram. We can see that the user must first try to access the gesture set page. An user can't delete gesture sets that belong to other users, therefore only gesture sets he has access to will be displayed. The Gateway microservice will then transfer the JWT of the user to the Gestures service. With this JWT, as we will see in the 3.4 section, we can retrieve the id of the user and so retrieve his gesture sets. If the user has no gesture set, it is of course impossible to him to delete one gesture class. With his gesture sets, a user can easily choose one gesture class contained in one of them in order to delete it. This time the Gateway service will transfer the information to the Gestures service and it is the Gestures service that will handle all the operations. Indeed, unlike the upload process we saw earlier, the JSON sent to the Gestures microservice is very limited and since it has its own database, it is much more appropriate for it to manage the whole process. Gestures service will then delete and unlink each sample present in the class and then delete the class and unlink it in the corresponding gesture set.

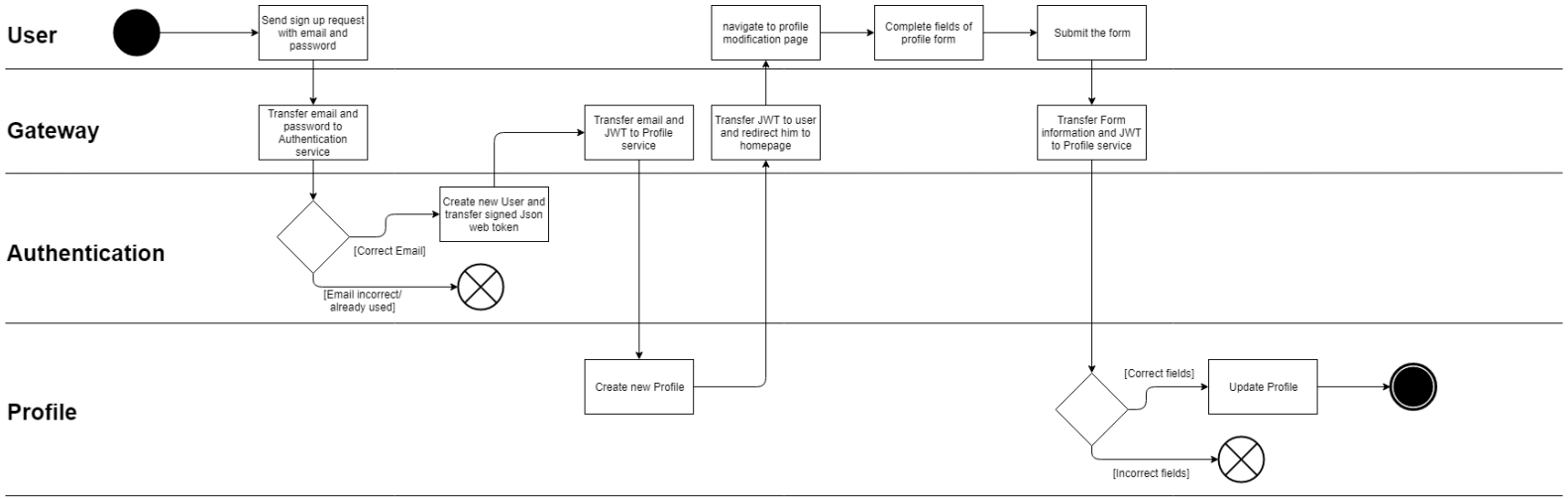

Figure 3.2: Workflow of creating a user and updating his personal information

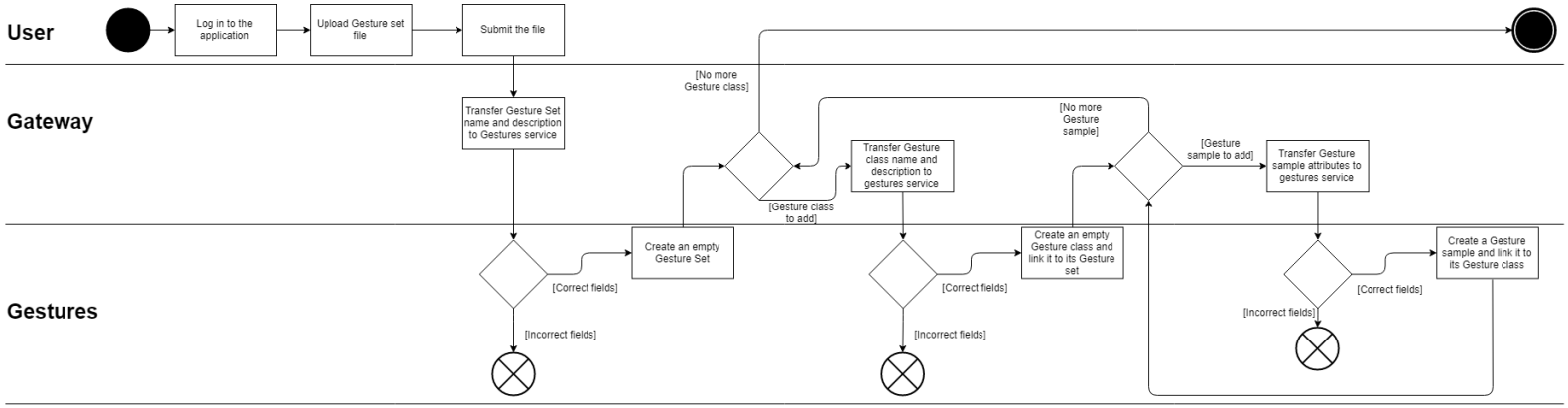

Figure 3.3: Workflow of creating a user and updating his personal information

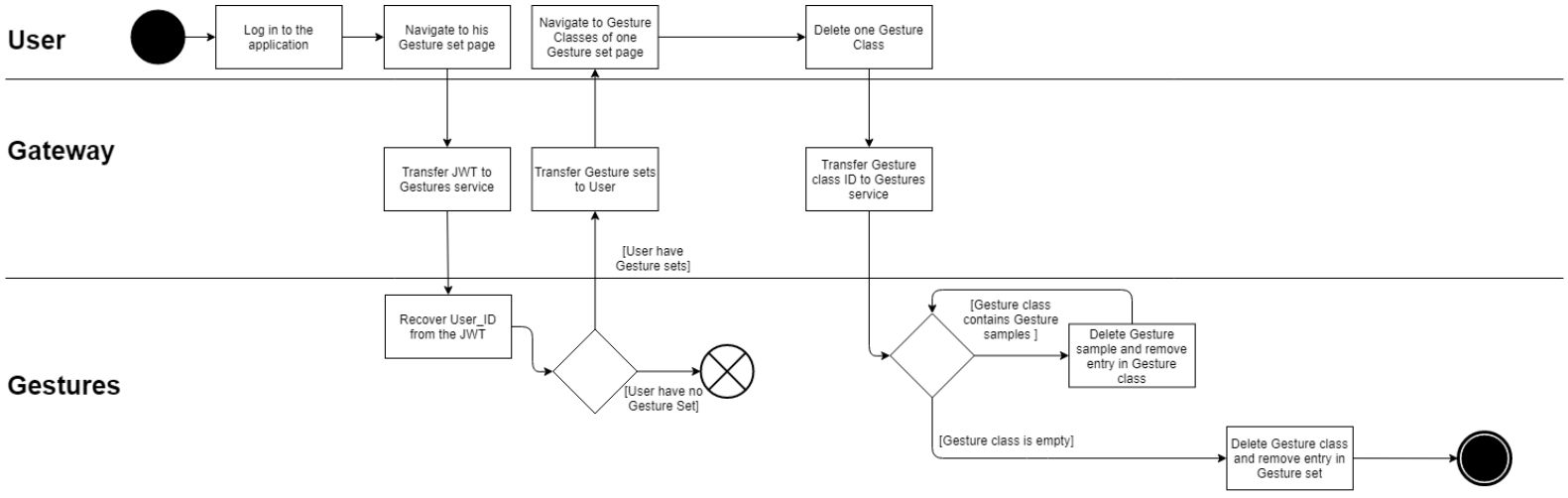

Figure 3.4: Workflow of creating a user and updating his personal information

3.3 Architecture

This section will cover the architecture of the application and main frameworks used during implementation. The purpose of this section isn't to go into details but to have a global vision of the architecture before explaining the implementation choices in the section 3.4.

As expected, the architecture of our application shown in the figure 3.5 is very similar to a classic microservices architecture as it can be seen in figure 2.18. The only link the client has with the back-end of the application is through the gateway microservice. This gateway will have the role of manager and will redirect according to the customer's needs to the other microservices. We can see that each microservice is an independent module running on a different server and of course each microservice has its own unique database.

The application is composed of some frameworks that we quickly define here under :

- MongoDB:
  MongoDB is an open source NoSQL database management system (DBMS) that uses a document-oriented database model. As a NoSQL database, MongoDB avoids the relational database's table-based structure to adapt JSON-like documents that have dynamic schemas called BSON. This makes data integration for certain types of applications faster and easier. MongoDB is built for scalability, high availability and performance from a single server deployment to large and complex multi-site infrastructures.
- Mongoose:
  Mongoose is an Object Data Modeling (ODM) library for MongoDB and NodeJS. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.
- NodeJS:
  NodeJS is an open source development platform for executing JavaScript code server-side. It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, and relies on its package ecosystem (npm) which is the largest ecosystem of open source libraries in the world.
- Express:
  Express is a web application framework that runs on NodeJS and provides a robust set of features for web and mobile applications.

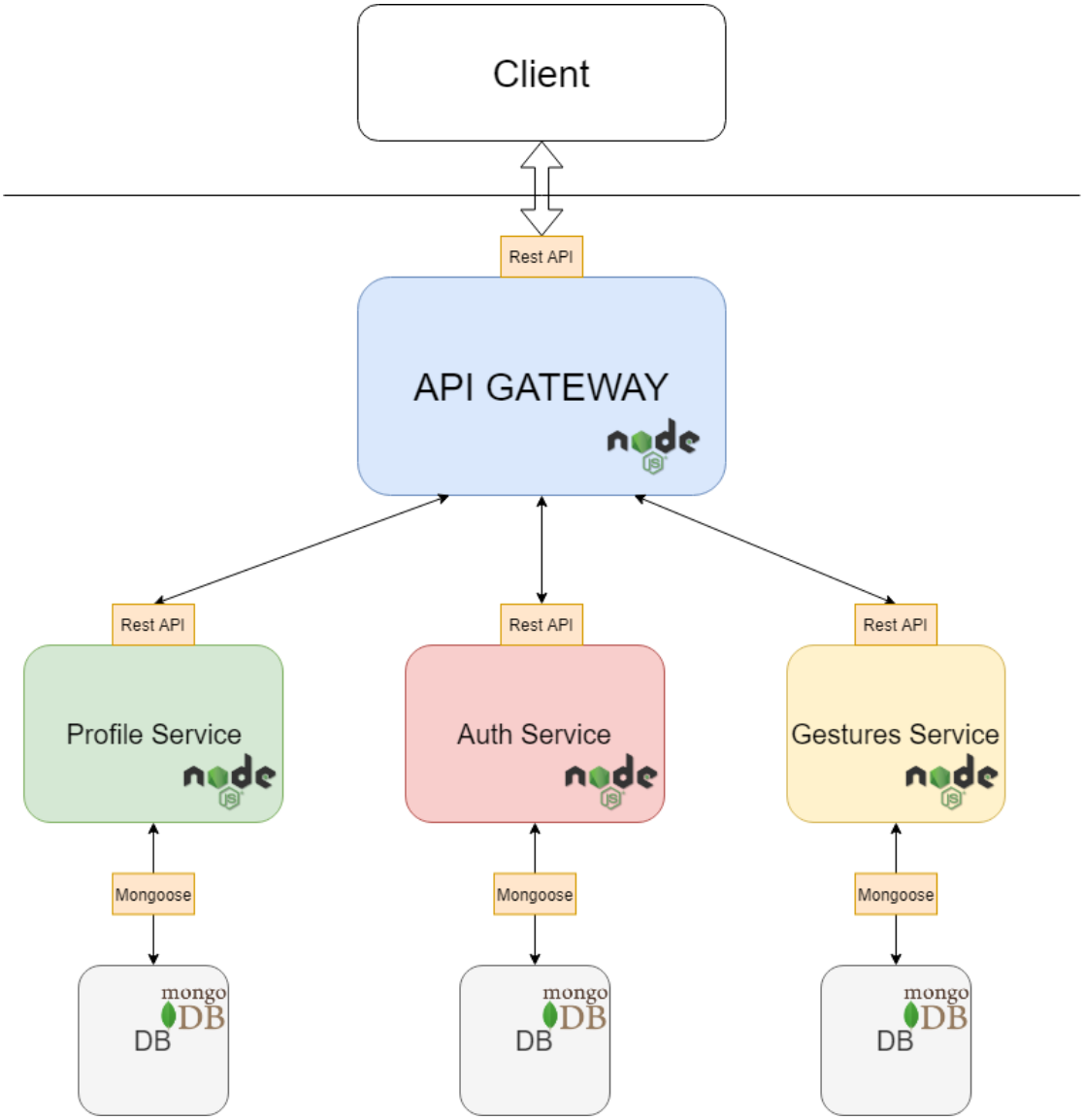

Figure 3.5: Architecture of the application.

3.4 Implementation choices

Now we have seen all structural views for our application, we can be interested in the different choices of implementations specific to our software. As often in software development, a multitude of potential solutions exists for a given problematic, and it is the developer responsibility to pick the more suitable one.

3.4.1 Microservices and splits

The main implementation choice of this application is the microservices architecture as we have already seen. But it is interesting to understand why it is relevant to use microservices in this situation and how we handle it.

Splits

Our software contains for now four independent microservices : Profile, Authentication, Gestures and Gateway.

Authentication microservice handles, as its name suggests, all the authentication part for the application. It is responsible for the security of the sign up and sign in operations. We will talk a bit later about how security works on the platform.

Profile microservice keeps user personal data as seen in the UML section.

Gestures microservice handle creation, modification and deletion of gestures sets, classes and samples. It is important to specify that this service does not perform calculations or classifications on gestures sets. Its purpose is to store and manage it correctly.

Gateway microservice is the orchestrator of the other services. It is responsible for recovering and sending data from one service to another microservice and handle client requests. Note that this service does not have a database linked to it. Indeed, the Gateway service is here to solely manage modules, not to store and handle data.

The split between Profile and Authentication actually comes from a security concern. If a malicious user can gain access to the profile database, this one still doesn't have access to authentication database and vice-versa. Furthermore, with this solution, we can keep authentication database clean and easier to manage.

Modularity

In a context like gesture recognition, new programs regularly emerge as well as new algorithms, approaches and ideas. Keeping a software easily modular and modifiable seems therefore a good option. With the solution presented in this thesis and the use of microservices, we can easily add new microservices (suchas gesture recognition algorithm, ...). All we have to keep is the basic structure of gestures, and even such structure can be easily customized without shutting down

the entire system. So the choice of microservices architecture in terms of the sector of activity in which we operate seems a wise choice.

Scalability

Just as it is complicated to know the evolution of the gesture recognition domain, it is difficult to know the potential use of such an application. If a large number of users come to use a specific mechanism such as simply uploading a significant number of gesture sets, we must be able to predict this scenario as it may result in an application slow-down given the important payload transferred through the application. With a microservices architecture, it is quite efficient to handle this type of situation. Indeed, instead of duplicating the entire application as we would do with a classic monolithic architecture, we just have to duplicate the overloaded service

3.4.2 Platform security

JSON Web Tokens (JWT)

First, we are entitled to ask ourselves: what is a Json Web Token? The JWT website [1] tell us :

"JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Signed tokens can verify the integrity of the claims contained within it. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it."

So JSON web token are principally used to protect the access to protected routes as well as data. In our application, this is the role of the authentication microservice to deliver a signed token after a new user sign up or an old one sign in with good credentials. This signed JWT will be saved locally (in local storage).

This signed token will be then used by others microservices when one user tries to access to a route or resource which requires an authentication. The client will send his JWT to the gateway using the header of the request and its *Authorization* field. After that the gateway will redirect this JWT to other microservices and these microservices will check the signature of the token. If it's verified, then the

user can have access to the data or route he wants. Given the stateless nature of microservices, this token need to be sent at each new request. Figure 3.6 shows an example of a JWT utilisation.

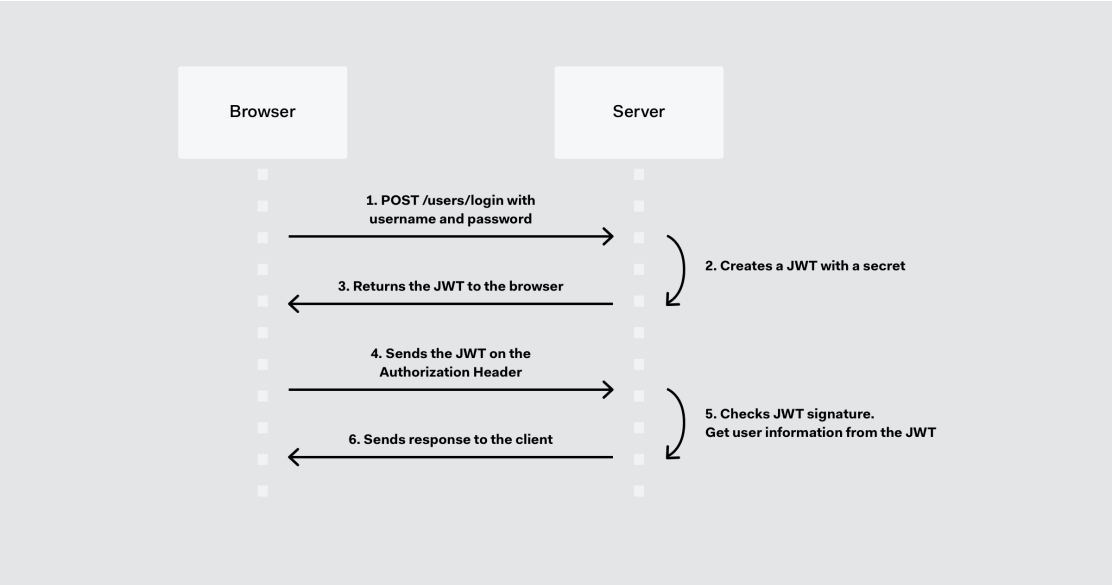

Figure 3.6: JSON Web token process example

A JWT is composed of three parts:

- Header :
  The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- Payload :
  The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data.
- Signature :
  To create the signature part, we have to take the encoded header, the encoded payload, a secret key, the algorithm specified in the header, and sign that. The signature is used to verify the message wasn't changed along the way, and in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

The figure 3.7 shows us how our JWT is signed and a good example. As we can see, the header is composed of two parts as mentioned above. The payload

is made of what we call *Registered claims* that are set of predefined claims which are not mandatory but recommended. **Iss** is application specific. It identifies the principal that issued the JWT. **Sub** allow us to know which user send the JWT. For us, it's the ID of the user who sent the request. **Iat** identifies the time at which the JWT was issued. It is directly related to the **exp** claim which identifies the expiration time after which the JWT *must not* be accepted for processing. For this application, the expiration time is set to one day after the JWT was issued. After this time, the user have sign up/sign in again.

The last part of our JWT is the signature. We can see the temporary private key set here. This private key is accessible for all microservices except the gateway. With this one they can verified that the Json Web Token is valid and authorize the user. Gateway microservice does not have access to JWT for a security concern. Indeed, we don't want that Gateway accept a user and then send requests without token to other microservices. With this approach, link between Gateway and other microservices would no longer be secure and it would have been easy to circumvent the gateway to access the data.

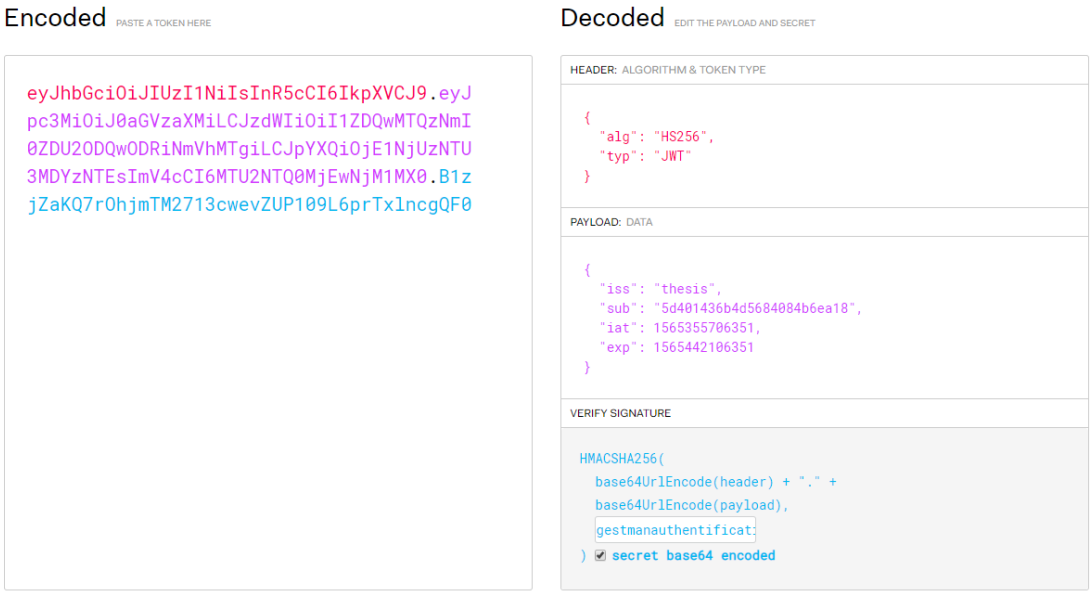

Figure 3.7: Our signed JWT and its decoded elements

Password security

For a security concern, all passwords present in the database are encrypted with the use of the *bcrypt* library for NodeJS. This protects the platform in the event of

a leak or stealing passwords.

Gestures sets and users

To prevent any user from having access to all gestures sets and modifying them, each gesture set has a dedicated user. So when an user creates a gesture set, his ID is associated to this gesture set. This ID will be recovered with the JWT at each new request and only that specific user will be able to update or delete the gesture set. This choice was made for practical reasons and is made in such a way that it can be easily modified. We will discuss possible improvements to this in the section 4.2

Chapter 4

Future work

The purpose of this chapter is to describe the various potential improvements that can be brought to the application developed in this thesis. This architecture of the platform is designed and built to accommodate new functionalities and therefore to become more and more complete over time. The first part of this chapter will be devoted to the necessary elements for the smooth running of future developments and we will follow with many ideas for reflection and addition to the software.

4.1 Further development

4.1.1 Prerequisites

To be able to add modules to the software, it is preferable to master these technologies to some extent:

1. Javascript
2. NodeJS
3. React
4. Express
5. Rest API
6. MongoDB
7. Microservices architecture

4.1.2 Configuration

It is important to specify that each microservice can be launched independently. So each of these modules have their own dependencies. Users must therefore navigate to each microservice directory and enter the following command in a terminal window :

```
$ npm install --save
```

This will install the dependencies. Then you can either decide to launch each module independently or to launch all the microservices with the command :

```
$ npm run start-dev
```

The microservices will be connected on port 5000 (Gateway) , 5001 (Profile), 5002 (Gestures) and 5003 (Authentication).

4.1.3 Folders structure

The following directory tree represents the file architecture of the application. Since each microservice has the same structure, we will develop only one of them:

```
Main Directory
├── package.json → script to install dependencies necessary to launch
│   microservices simultaneously and run the application
├── Authentication microservice
├── Gateway microservice
├── Profile microservice
└── Gestures microservice
    ├── Configuration
    │   └── index.js → secret key for Json Web Tokens
    ├── Controllers → functions called by Routes
    │   ├── classes.js
    │   ├── samples.js
    │   └── sets.js
    ├── Helpers
    │   └── routeHelper.js → functions to validate body or parameters
    │       of the request
    ├── Models → schemas of the database
    │   ├── classes.js
    │   ├── samples.js
    │   └── sets.js
    └── Node_modules → dependencies, auto-generated with the npm install
        command
```

```
├── Routes → roads of the microservice, manages the requests (post,
│   get, ...)
│   ├── classes.js
│   ├── samples.js
│   └── set.js
├── app.js → microservice launcher (configure,connect,launch)
├── package.json → script to install dependencies necessary to launch
│   this microservice and run it
├── passport.js → manages the security aspect (with JWT , google
│   OAUTH, ...)
└── README.md
```

4.2 Community integration

As we have seen in the section 3.4, the access to gesture sets of the user is currently granted via the use of a JSON web token (JWT) and an attribue *User_id* in the gesture set class. This solution works properly and ensures that each user have only access to their own gesture sets. Nevertheless, this solution has some limitations. Indeed, the community part of the application falls through the cracks. This can be easily modified by transforming the attribute into an array of ids and adding the ability for users to add users to it. But here again, it is possible to improve it. That can be done with the use of a new microservice: a Permission microservice. This module would be based on Role-Based Access Control structure [21]. This structure allows users to have multiple roles and permissions on these gesture sets. For example, only the admin of the gesture set will be authorized to delete his gesture set but this does not prevent several users from accessing it. In the figure 4.1, you can see an example of a role based access UML model, made by *Loddersted,Basin and Doser* [27] and that can be converted into a microservice.

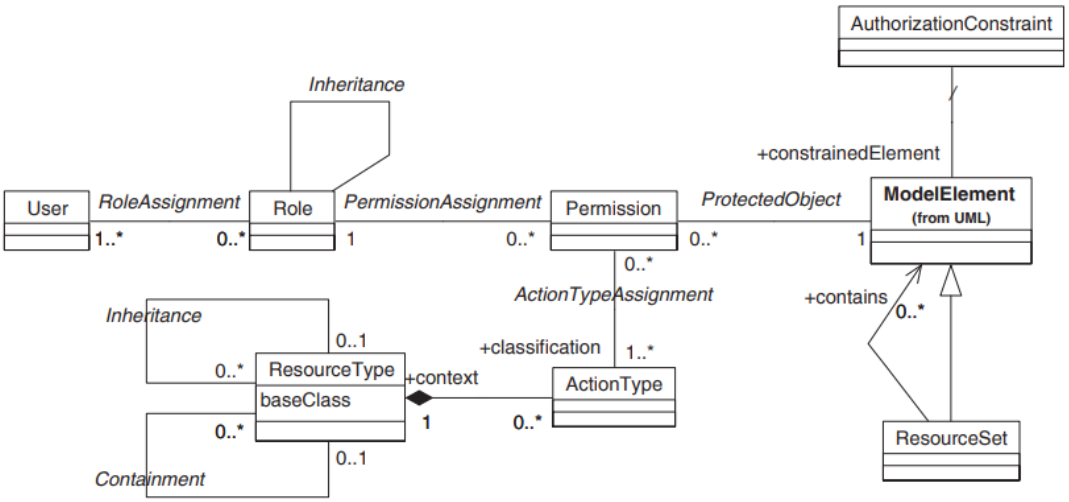

Figure 4.1: Our signed JWT and its decoded elements

4.3 Gesture samples creation

Currently, our application doesn't handle the creation of gesture samples directly on the platform. Indeed, it requires a complete import from a gesture set to a JSON format. This file must follow a precise structure otherwise the import can't be done properly. Even if the JSON format is a widely used format and it is easy to create a file with the correct structure, it is necessary to build another module allowing the creation of samples directly via the application in order for it to be complete. This microservice could have similar features with the previous version of Gestman [13] and its gesture samples creation work.

4.4 Gesture recognition algorithms

After the creation, recognition and study of gestures with the use of dedicated algorithms is a feature that will ultimately have to be implemented. These algorithms can be of two types:

- Proposed by the software itself. These algorithms will then be stored in a microservice and it will be able to launch them at the user's request. The advantage of this technique is to ensure the proper functioning of the algorithms but it limits the number of algorithms offered to those integrated in the core of the application.

- Uploaded by the users themselves. This technique shows a significant advantage which is that it allows a much larger number of algorithms and approaches. But this approach also shows a much higher security risk. Indeed, inject foreign code and run it in the software can be very dangerous. Luckily, the microservices architecture allows us to overcome this problem. Indeed, since microservices are by definition independent of each other, a microservice dealing with this problem could very well isolate the code and contain it in the case a problem would occur. The only potentially damaged part of the application will then be the microservice itself.

4.5 Message Passing

The software currently use REST (Representational State Transfer)to communicate between microservices. This request system has many advantages but it also shows some drawbacks, especially in the microservices architecture. One of these drawback is the *blocking*. Indeed, when invoking a REST service, your service is blocked waiting for a response. This hinders the application performance because it could be processing other requests in the meantime. If request processing is fast and the number of requests is relatively low, the application may not exhibits performance issues. But if we have requests that take more time or a larger number of requests, it may become a performance bottleneck. A striking example is the import of gesture set. This request ask to the application microservices a significant number of internal requests and actions. It takes a considerable amount of time, during which time the blocking effect of the REST can be problematic.

This is where the message-passing technique comes in. The message-passing technique with microservice is based on event-driven architecture. Event Driven Microservices (EDM) [22] are inherently asynchronous and are notified when it is time to perform work. With this approach, microservices should perform as efficiently as possible, and it is a waste of resources to have many threads blocked and waiting for a response. With asynchronous messaging applications can send a request and process another request instead of waiting for a response.

4.6 Docker deployment

The last but not least part of the application development is its deployment. A technology very useful for the microservices deployment is *Docker*. First, it is relevant to define Docker. *Charles Anderson* [4] give us an explanation :

"Docker is a container virtualization technology. So, it's like a very lightweight virtual machine [VM]. In addition to building containers, we provide what we call a developer workflow, which is really about helping people build containers and applications inside containers and then share those among their teammates."

Therefore, Docker allows us to create and deploy an isolated environment. This is exactly what we're looking for with a microservices architecture-based application. We can create containers for each of our microservices and launch them independently. With this technique we ensure the safety of the application but also the containment of errors. For example, as we saw earlier when mentioning gesture recognition algorithms.

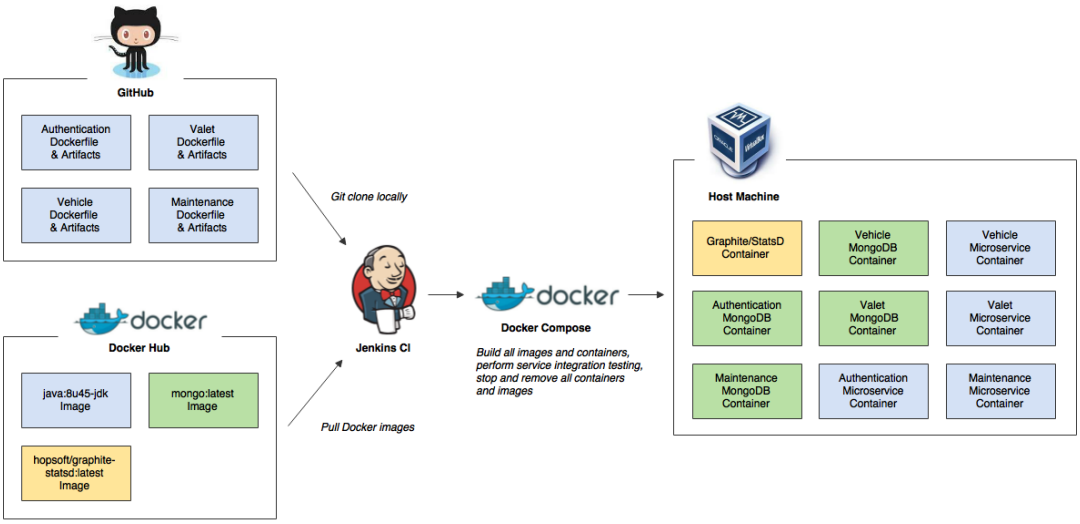

Figure 4.2: Example of docker build and compose before the deployment

Chapter 5

Conclusion

The main goal of this thesis was to develop a platform based on a microservices architecture. This platform must allow to easily and quickly integrating, modifying and deploying new modules, whether they are based on existing tools or new ones. For this purpose, the software must have a clear architecture and a neat code to allow other developers to easily continue the development of the application.

After a detailed description of the operations, architecture and implementation of the application in chapters 3 and 4, the main objectives have been achieved. The developed platform meets the requirements and may serve as a solid basis for future improvements.

The application has a microservices architecture that adapts well to the domain of gesture recognition, providing a flexibility and scalability that are beyond the capabilities of a monolithic architecture. It allows quick adaptation to new modules and efficient maintenance, which gives a real advantage for future developments.

In order to develop this gesture management application, it was necessary to acquire a solid background in gesture recognition and related theoretical domains. It was also necessary to understand and master several tools such as NodeJS, MongoDB or ExpressJS.

As mentioned, this application is designed to be improved. A particular attention was therefore paid to these future improvements and the work to be done to achieve them. With this additional work and ongoing monitoring, the application has the potential to provide a better understanding and cohesion in the gesture recognition domain. We hope that this will be the case.

Bibliography

[1] Introduction to json web tokens. `https://jwt.io/introduction/`.

[2] MARTIN L. ABBOTT and MICHAEL T. FISHER. *THE ART OF SCALABILITY : Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 2009.

[3] Meredith Ringel Morris Abdullah X. Ali and Jacob O. Wobbrock. Crowdsourcing similarity judgments for agreement analysis in end-user elicitation studies. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 177–188, New York, NY, USA, 2018. ACM.

[4] Charles Anderson. Docker [software engineering]. In *IEEE Software*, pages 102–c3. IEEE, 2015.

[5] Moira C. Norrie Beat Signer, Ueli Kurmann. igesture: A general gesture recognition framework. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2*, 2007.

[6] Tony Bradley. The challenges of scaling microservices. `https://techbeacon.com/app-dev-testing/challenges-scaling-microservices`, November 2015.

[7] Oğuz Turan Buruk and Oğuzhan Özcan. Gestanalytics: Experiment and analysis tool for gesture-elicitation studies. In *Proceedings of the 2017 ACM Conference Companion Publication on Designing Interactive Systems*, DIS '17 Companion, pages 34–38, New York, NY, USA, 2017. ACM.

[8] Maria Claudia Buzzi, Marina Buzzi, Barbara Leporini, and Amaury Trujillo. Exploring visually impaired people's gesture preferences for smartphones. In *Proceedings of the 11th Biannual Conference on Italian SIGCHI Chapter*, CHItaly 2015, pages 94–101, New York, NY, USA, 2015. ACM.

[9] Mike Amundsen James Lewis Cesare Pautasso, Olaf Zimmermann and Nicolai Josuttis. Microservices in practice, part 1: Reality check and service design. In *IEEE Software 34*, pages 91–98. IEEE, 2017.

[10] Andrew D. Wilson Jacob O. Wobbrock and Yang Li. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. *User Interface Software and Technology (UIST)*, 2007.

[11] Sujin Jang, Niklas Elmqvist, and Karthik Ramani. Gestureanalyzer: Visual analytics for pattern analysis of mid-air hand gestures. In *Proceedings of the 2Nd ACM Symposium on Spatial User Interaction*, SUI '14, pages 30–39, New York, NY, USA, 2014. ACM.

[12] Paolo Roselli Jean Vanderdonckt and Jorge Luis Pérez-Medina. !ftl, an articulation-invariant stroke gesture recognizer with controllable position, scale, and rotation invariances. *Proc. of ICMI '18*, 2018.

[13] Zacharie Kerger and Romain Dizier. *GestMan, towards an online gesture management system.* PhD thesis, Université catholique de louvain, 2018.

[14] James Lewis and Martin Fowler. Microservices a definition of this new architectural term. *Objektspektrum*, 2015.

[15] Radu-Daniel Vatavu Lisa Anthony and Jacob O. Wobbrock. Understanding the consistency of users' pen and finger stroke gesture articulation. *Proceedings of Graphics Interface*, 2013.

[16] Jean Vanderdonckt Jorge Luis Pérez-Medina Radu-Daniel Vatavu Nathan Magrofuoco, Paolo Roselli. Gestman: A cloud tool for stroke-gesture datasets. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2019.

[17] Michael Nebeling, David Ott, and Moira C. Norrie. Kinect analysis: A system for recording, analysing and sharing multimodal interaction elicitation studies. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '15, pages 142–151, New York, NY, USA, 2015. ACM.

[18] Alberto Lluch Lafuente et al. Nicola Dragoni, Saverio Giallorenzo. Microservices: Yesterday, today, and tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[19] MARCEL OTTO. Microservices: when not to use them.... `https://www.qualogy.com/techblog/it-development-and-operations/microservices-when-not-to-use-them\%E2\%80\%A6`, June 2018.

[20] Lisa Anthony Radu-Daniel Vatavu and Jacob O. Wobbrock. Gestures as point clouds: A $p recognizer for user interface prototypes. *International Conference on Multimodal Interaction*, 2012.

[21] H.L. Feinstein R.S. Sandhu, E.J. Coyne and C.E. Youman. Role-based access control models. In *Computer (Volume: 29 , Issue: 2)*, pages 38–47. IEEE, 1996.

[22] Jonathan Schabowsky. Rest vs messaging for microservices – which one is best?, June 2017.

[23] Stahovich T. Sezgin, T. M. and R. Davis. Sketch based interfaces: early processing for sketch understanding. In *Proceedings of the 2001 workshop on Perceptive user interfaces*, pages 1–8, New York, NY, USA, 2001. ACM.

[24] John Spacey. What is smart endpoints and dumb pipes. `https://simplicable.com/new/smart-endpoints-and-dumb-pipes`, January 2017.

[25] Maximilian Speicher and Michael Nebeling. Gesturewiz: A human-powered gesture design environment for user interface prototypes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 107:1–107:11, New York, NY, USA, 2018. ACM.

[26] Johannes Thönes. Microservices. In *IEEE Software*, pages 116–116. IEEE, 2015.

[27] Jürgen Doser Torsten Lodderstedt, David Basin. Secureuml: A uml-based modeling language for model-driven security. In Stephen Cook Jean-Marc Jézéquel, Heinrich Hussmann, editor, *UML 2002 — The Unified Modeling Language*, pages 426–441. Springer, Berlin, Heidelberg, 2002.

[28] Jean Vanderdonckt, Nathan Magrofuoco, Nicolas Burny, Paolo Roselli, and Jorge Luis Perez-Medina. A survey of 2d gesture recognition techniques. *ACM Comput. Surv.*, 2017.

[29] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. Relative accuracy measures for stroke gestures. *ICMI '13 Proceedings of the 15th ACM on International conference on multimodal interaction*, 2013.

[30] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. Gesture heatmaps: Understanding gesture performance with colorful visualizations. *ICMI '14 Proceedings of the 16th International Conference on Multimodal Interaction*, 2014.

[31] Radu-Daniel Vatavu and Jacob O. Wobbrock. Formalizing agreement analysis for elicitation studies: New measures, significance test, and toolkit. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1325–1334, New York, NY, USA, 2015. ACM.

[32] Radu-Daniel Vatavu and Jacob O. Wobbrock. Between-subjects elicitation studies: Formalization and tool support. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3390–3402, New York, NY, USA, 2016. ACM.

[33] Phil Wittmer. Microservices disadvantages advantages – tiempo development. `https://www.tiempodev.com/blog/disadvantages-of-a-microservices-architecture/`, July 2019.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**