MARCO ANTÓNIO GUERREIRO DE FREITAS

MICROSERVICES APPLIED TO WEB AND MOBILE APPLICATIONS

INTERNSHIP

View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Sapientia

UNIVERSIDADE DO ALGARVE

INSTITUTO SUPERIOR DE ENGENHARIA

2018

MARCO ANTÓNIO GUERREIRO DE FREITAS

MICROSERVICES APPLIED TO WEB AND MOBILE APPLICATIONS INTERNSHIP

Mestrado em Engenharia Elétrica e Eletrónica
Especialidade em Tecnologias de Informação e Telecomunicações

Trabalho efetuado sob a orientação de:
Prof. Dr. João Rodrigues

UNIVERSIDADE DO ALGARVE
INSTITUTO SUPERIOR DE ENGENHARIA
2018

MICROSERVICES APPLIED TO WEB AND MOBILE APPLICATIONS INTERNSHIP

Declaração de autoria de trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

I hereby declare to be the author of this work, which is original and unpublished. Authors and works consulted are properly cited in the text and included in the reference list.

(Marco António Guerreiro de Freitas**)**

©2018, MARCO ANTÓNIO GUERREIRO DE FREITAS

A Universidade do Algarve reserva para si o direito, em conformidade com o disposto no Código do Direito de Autor e dos Direitos Conexos, de arquivar, reproduzir e publicar a obra, independentemente do meio utilizado, bem como de a divulgar através de repositórios científicos e de admitir a sua cópia e dis-Tribuiçãoparafinsmeramenteeducacionaisoudeinvestigaçãoenãocomerciais, conquanto seja dado o devido crédito ao autor e editor respetivos.

The University of the Algarve reserves the right, in accordance with the terms of the Copyright and Related Rights Code, to file, reproduce and publish the work, regardless of the methods used, as well as to publish it through scientific repositories and to allow it to be copied and distributed for purely educational or research purposes and never for commercial purposes, provided that due credit is given to the respective author and publisher.

Resumo

Micro serviços é uma arquitetura de software que cresceu em popularidade e utilização nos anos mais recentes. Devido à sua simplicidade e facilidade de implementação é hoje em dia o padrão referência para serviços web. Este relatório foca os conceitos gerais de micro serviços e a sua aplicação num âmbito profissional.

O relatório foi dividido em 3 pontos principais: (a) conceitos gerais de micro serviços, (b) reconhecimento facial e de emoções aplicado a uma interface adaptativa de utilizador utilizando micro serviços e (c) reestruturar uma aplicação web existente seguindo uma arquitetura de micro serviços.

O primeiro ponto, o (a) conceitos gerais de micro serviços, introduz a definição de um micro serviço e quais as principais características que um micro serviço deve obedecer. No segundo ponto, (b) descreve-se a implementação de uma aplicação mobile com uma interface adaptativa de utilizador recorrendo a micro serviços para reconhecimento facial e de emoções. Neste ponto é possível compreender como consumir micro serviços desenvolvidos por terceiros. No terceiro ponto, (c) descreve-se a reestruturação de uma aplicação web existente, através de uma arquitetura de micro serviços, exemplificando os conceitos mencionados no ponto (a) e pequenos excertos de código de um projecto open-source para demonstrar a sua utilização.

Palavras-chave: Micro serviços, Interface de Utilizador Adaptativa, Testes unitários, NancyFx, Aplicação web, .NET

Abstract

Microservices is a software architecture that has seen an increase in popularity and use in recent years. Due to its simplicity and ease of implementation it is nowadays the standard reference for web services. This report focuses on the general concepts of microservices and their real-world application.

The report was divided into three main points: (a) general concepts of microservices, (b) facial and emotion recognition applied to an adaptive user interface using microservices and (c) restructure an existing web application following a microservices architecture.

The first point, (a) general concepts of microservices, introduces the definition of a microservice and what are the main traits that a microservice should have. The second point, (b) describes the implementation of a mobile application with an adaptive user interface that uses microservices for facial and emotional recognition. In this point it is possible to understand how to consume microservices developed by third parties. The third point, (c) describes the restructuring of an existing web application, through a microservices architecture, exemplifying the concepts mentioned in point (a) and small excerpts of code from an open-source project to demonstrate its real-world code usage.

Keywords: Microservices, Adaptive User Interface, Unit testing, NancyFx, Web application, .NET

Acknowledgements

The elaboration of this work would not have been possible without the collaboration, commitment of several people. I would therefore like to express my full gratitude and appreciation to all those who, directly or indirectly, contributed to this task become a reality. I want to express my sincere thanks.

In first place, my friend Professor João M. F. Rodrigues whom I have known for ten years. The notes of his orientation, the usefulness of his recommendations and the cordiality with which he always received me. I am grateful for all of them and also for the freedom of action that it was decisive for this work to contribute to my personal development. As teacher was the maximum exponent, opened horizons, taught me mainly to think. It was, and is, fundamental in the experiences, in the creation and solidification of knowledge and in my small successes. As a friend is what we all want, is always by our side without needing to ask for anything. To my friend and colleague Emanuel Ey for accepting the task of being my internship supervisor at Evodeck Software whose open attitude, knowledge and total availability, were remarkable. Your availability unrestricted, its demanding, critical and creative way of arguing the ideas presented, facilitated the achievement of the objectives proposed in this thesis.

To all my colleagues at Evodeck Software for friendship and companionship. To my friends who have never been absent, I thank the friendship and the affection that always made me available.

To my father and to my mother, for the solid formation given in my youth, which provided me with the continuity of my studies until the arrival of this master's degree, my eternal thanks. Finally, to my girlfriend, I thank you for all your love, affection, admiration, and the tireless presence with which she supported me throughout the elaboration of this thesis.

Thank you all for allowing this thesis to be a reality.

Table of Contents

List of Figures

List of abbreviations

BDD	Business Driven Development
CI	Constructor Injection
DI	Dependency Injection
DSL	Domain Specific Language
DTO	Data Transfer Object
EF	Entity Framework
ERP	Enterprise Resource Planning
HTTP	Hyper Transfer Text Protocol
IoC	Inversion of Control
JSON	JavaScript Object Notation
Nancy	NancyFX
OS	Operating System
PI	Property Injection
SDK	Software Development Kit
SOA	Service Oriented Architecture
SRP	Single Responsibility Principle
SUT	System Under Test
TDD	Test Driven Development
UI	User Interface
URI	Uniform Resource Identifier
UTC	Coordinated Universal Time
UWP	Universal Windows Platform
XML	Extensible Markup Language

1 Introduction

It is known that Portugal has competent professionals in the field of information technology. This was proven during the economic crisis that the country and the whole world faced, and are still facing, where several professionals emigrated to find better economic and working conditions and to see their professional value much more recognized. Although there was an enormous export of qualified labor neither Portugal, nor any of the most developed countries in Europe, can suffice the hard demand for qualified labor in the area of software development [1].

JavaScript has been around for a couple years now and has become a standard language, nowadays, for the development of web applications and even mobile applications. Surveys show that it is also one of the most popular programming languages [2] around and one of the easiest to learn [3]. The recent demand for ReactJS developers [4] and React-Native developers [5] has been on the rise with a great number of projects and technical barriers being overthrown with easiness of development that the languages have been introducing and also because of the cost of developing mobile apps in native language is very high [6].

Microservices have been serving the purpose of replacing old, big monolithic applications into lightweight applications that are easily replaceable, testable and deployable to production environment, reducing the cost of develop and maintenance and optimizing company resources [7]. Bearing all this in mind Evodeck Software [8] was founded in Faro to valorize the abundant quality of software developers and avoid qualified labor from emigrating out of the Algarve region.

Evodeck Software is a startup founded in March 2017 with its headquarters in Faro, specializing in the area of information technology, namely software development, while using working methodologies such as Agile and Scrum [9]. Evodeck works mostly with JavaScript programming languages such as ReactJS, React Native or Angular [10] for projects more focused on web and mobile applications. Their projects originate from international customers, mostly from Germany, due to the founder's nationality, which allows for more technologically complex projects than the ones proposed from Portuguese customers. Within that context the author works as a software developer with tasks other than just developing software such as cooperating with the customers for continuous planning of projects and perform Q&A to guarantee that all products are deployed with the minimum set of errors.

The internship in the ambit of the masters of Electric and Electronic Engineering specialization in Technologies of Information and Telecommunication is a fit for the study of microservices architecture [11], using microservices on a React Native mobile app, developing a web application in .NET using the framework NancyFX [12] following a Service Oriented Architecture (SOA) [13] to replace the current application. The application will manage and facilitate product reselling and additional services between

suppliers and resellers. It will also manage additional services from one or more external providers and will have to integrate and consolidate with current services and/or products.
The three major capabilities will be:

a) Manage and integrate different types of products and services;
b) Manage external entities and their services;
c) Access control politics.

Since the focus is to develop a web application that relies on microservices the internship objectives will be:

1) Write a simplified state of the art about apps that include the same kind of functionalities;
2) Study microservices architecture;
3) Study React Native: acknowledge common development issues (content share, development limitations, etc.);
4) Study the framework NancyFX: how to implement scalable and testable microservices;
5) Development of the web application;
6) Write the internship report.

The report is organized as follows. In Chapter 1 it was introduced the subject and the goals of the internship, in Chapter 2 we describe what are microservices and the general guidelines on how to implement them. In Chapter 3 we describe a mobile application that consumes microservices to apply facial and emotion recognition for adaptive user interfaces. In Chapter 4 we describe the practical implementation of the internship by exemplifying how microservices were implemented. In Chapter 5 we present our conclusions about the internship and its goals.

2 Microservices

The term microservices first arouse in 2012 [14] to reference a common architectural style that had seen an increase in usage. This style promoted the usage of loose-coupling services, small focused blocks of code that provide one usage to be consumed by other services or applications, behind a Uniform Resource Identifier (URI) [15] connected through pipelines providing flexibility, simplicity and robustness to changes. This would turn large and complex systems into smaller chunks of abstractions in a SOA.

It continued evolving and adapting its architecture and patterns to the needs of the ever-evolving web and nowadays the common definition of a microservice is of a service with one, and only one, restrict capability that a remote API allows access to the rest of the system. For every capability in a system a microservice should be implemented to become isolated and narrow focused. This allows isolated deployment for a microservice, dedicated data store so there is no interference with other microservices data stores, execution in a separate process and collaboration with other microservices to complete its own action. This approach is much scalable and even allows for a short lead time from the start of implementation to deployment in production since the microservices are more isolated, which is a great benefit compared to traditional service-oriented approaches and monolithic architectures.

Microservices are agnostic to the programming language, the platforms where they can be executed are wide and the most common are IIS [16], Node [17] or NGINX [18]. What allows them to communicate using different platforms is the communication protocol, being the most common scenario communication over HTTP, as shown in Figure 2.1.

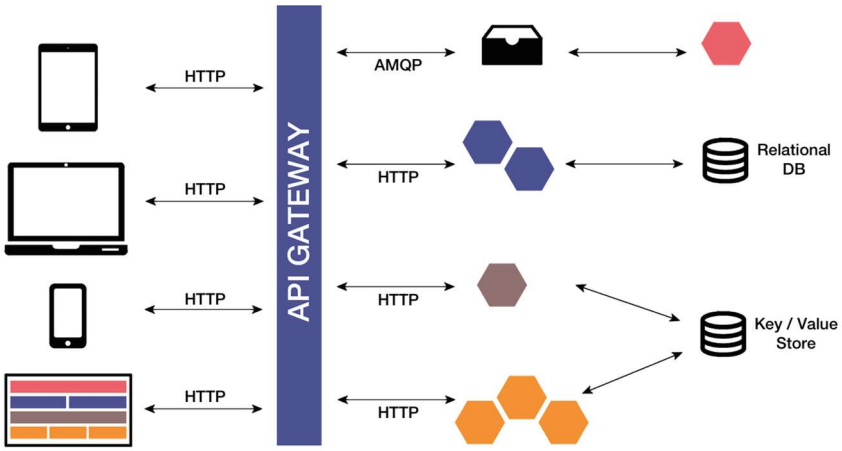

Figure 2.1 - An example of microservices architecture (adapted from [19])

A microservice should feature six traits: be responsible for a single capability, be individually deployable, consist of one or more processes, own its own data store, be maintainable by a small team and be replaceable.

2.1 Responsible for a single capability

The Single Responsibility Principle (SRP) [20] can be applied to a microservice to achieve the same result: be responsible for the required capability. Any other capabilities should be implemented elsewhere so that the required capability is the only cause of changing for that microservice. This definition can even be stretched to strive to fully implement the capability in the microservice so that only that microservice must change when the capability needs to be changed. The reasons to change a capability derive from the type of capability: business or technical.

A business capability is something the system does that contributes to the purpose of the system, e.g., calculating prices for items in a shopping cart. A technical capability is a capability that several other microservices need to use, e.g., integration of a third-party system.

Karma [21] described this trait, in an article post [22], as being what makes a microservice work best, even when using third party dependencies to make sure they don't have to think about them on other parts of an app.

2.2 Individually deployable

When a microservice is changed it should be able to be deployed to the production environment without touching any parts of the system. This allows the remaining microservices to continue executing during the deployment of the changed microservice and continue executing once the deployment has finished. This is an important trait because it is very common to have many microservices in a system having collaboration between them and development work can occur on a few or all the microservices, even in parallel. If there is the need to deploy all of them or even in small batches this can be quite risky. Whereas if the microservices are deployed in small changes results in small, low-risk deployments.

This also has its drawbacks, meaning the microservice's interface must be backward compatible to make sure other microservices are able to continue collaborating with the new version exactly as they did before. It also means that a microservice must expect other services to fail occasionally and must be able to continue working as best as possible. A failure in a microservice should not result in other microservices failing due to downtime during deployment. It should result in reduced functionality or a longer processing time.

2.3 Consist of one or more processes

A microservice should run in a separate process or in separate processes to remain independent of other microservices in the same system. The same is true if a microservice is to remain individually deployable. If there are two or more microservices running in the same process this leads to an undesirable coupling between them, where problems might arise from each other. It also means that when considering deploying a new version of a microservice would cause all other microservices to be redeployed too.

2.4 Own its own data store

A microservice should own the data store to store any required data, a disadvantage of being a complete capability since most business capabilities require data storage. Since each microservice owns its data store it prevents the direct access from any other microservice and forces the usage of the available public interface, as shown in Figure 2.2.

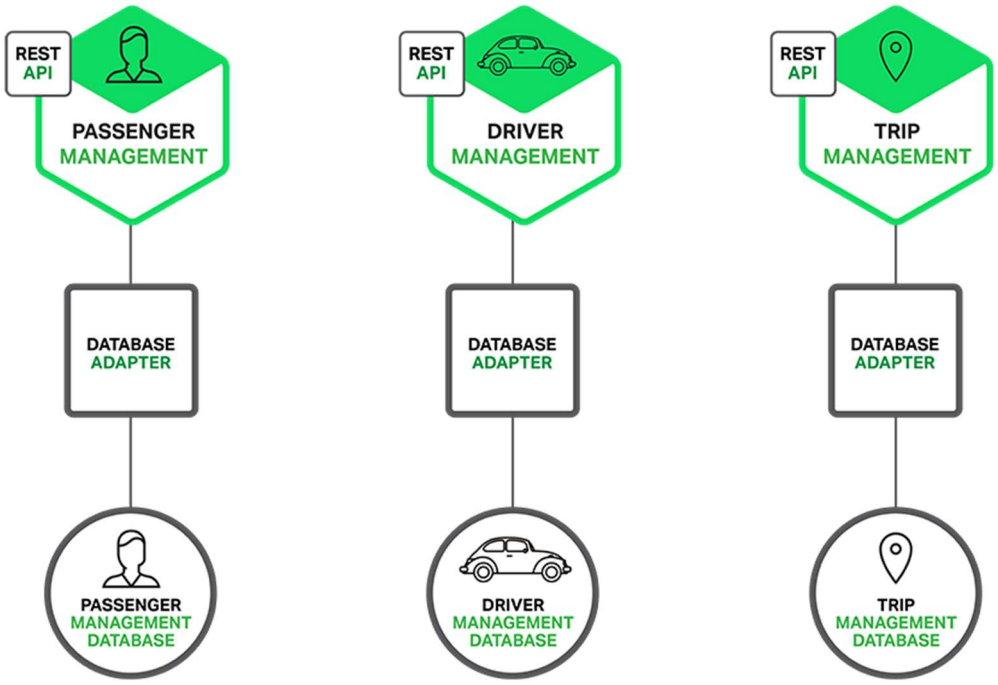

Figure 2.2 - A data store for each microservice (adapted from [23])

This approach also allows the usage of different database technologies on each microservice since the chosen database technology is part of the implementation, remaining hidden from the view of other microservices, leading to benefits in terms of development time, performance and scalability. Another drawback for this approach is the need to administer, maintain and work with more than one database. The

best benefit of a microservice owning its own data store is the ability to change a database for another in a later stage.

Amazon's CTO, Wener Vogels, talked about this feature in an interview [24] when asked about Amazon's experience with services in 2011, where he focused that no direct database access is allowed from outside the service and there's no data sharing among the services.

2.5 Maintainable by a small team

The term micro in microservice is a mere hint about what should be the size of a microservice but not in a completely transparent way. We should consider the amount of work involved in maintaining a microservice to the point where a small team, e.g., up to five members, should be able to maintain a handful of microservices. This means that a small team must be able to develop new functionalities, create new microservices out of old ones when these grow too big or out of scope, run them in production environments, monitor, test, fix bugs and all other required steps to function properly.

2.6 Replaceable

For a microservice to be replaceable it should be rewritten from scratch within a reasonable time frame meaning the team maintaining the microservice should be able to replace the current implementation with a completely new one within the normal pace of their work. This trait is a constraint on the size of a microservice. If it grows too large it can become expensive to replace but if small it can be replaced easily. The top reasons to replace a microservice tend to be changes in business requirements over time or the code base has grown to a point where it needs rewriting to be easily maintained.

2.7 Microservices architecture with continuous delivery

Continuous delivery [25] can be considered when implementing a microservices architecture. Quick development and modification of microservices backed up by automated testing [26] and considering individual deployment is what makes continuous delivery possible in microservices gathering the benefits of having reliable releases, risk reduction and improved product quality. This ensures that product releases can be deployed to production at any time, always a business decision, and is usually deployed after hitting source control.

To able to achieve this microservices need to be in a fully functional state, i.e., they need to have a high degree of test automation and development in small increments to guarantee they have the necessary quality. This process must also be repeatable, reliable and fast to have frequent production deployments.

The downside of choosing continuous delivery is the required technical skill alongside the process itself and the culture of close collaboration between all parties involved in development, security experts, business people and even system administrators, i.e., it requires a DevOps [27] culture where development and operations collaborate and learn from each other.

2.8 Downsides of a microservices architecture

There are drawbacks when choosing a microservices architecture that should be considered when choosing to follow, or not, this type of architecture. Microservices systems are distributed systems [28] which by itself carry a toll, they are harder to test than traditional monolithic systems, communication across process boundaries can be much harder and troublesome. Even communication across networks can be troublesome and is slower than in-process method calls.

Usually microservices systems are the result of a group, or a set of groups, of microservices and each one of them needs development, deployment and maintenance when in production which might result in a great number of deployments or an elaborate production setup.

Lastly, each microservice is a separate codebase which means that when refactoring code from one microservice to another it can be very difficult. To avoid this the scope of each microservice needs to be foreseen early on.

Despite its downsides implementing microservices, or implementing calls to microservices, is not an arduous task and can be easily achieved if the chosen architecture, or approach to the calls, is correctly implemented as exemplified on Chapter 3. Since we have presented the most common traits about microservices we will illustrate an application that consumes microservices. As this application is about facial and emotional recognition we will start with the state of the art.

3 Facial and Emotion Recognition for Adaptive User Interfaces

Accompanying the evolution of software and hardware, facial recognition has also evolved to the point where nowadays we can easily find this feature incorporated into our day to day devices such as a laptop, desktop computer, tablet or a smart-phone.

This kind of functionality is usually applied, solely or as another layer of security, to locking or unlocking mechanisms such as a login. There is also a wide range of implementation types that can be hardware alone where a device incorporates a camera and all the required algorithms (firmware software) to perform the matching process or a combination of software, implemented in an Operating System (OS) or a library, and hardware devices such as a webcam or the camera on a phone or tablet.

Another use case is the example of Adaptive User Interfaces [29] that harvest user information to differentiate and specify which information should be show to which type of user [30], using emotion recognition to shift content display to the corresponding emotion or using facial recognition to unlock an app and to show content, only, for the recognized face or minimize content display when different users are present [31].

More recently microservices have been emerging as a solution that allows shifting the implementation details of facial recognition to an online service whereas the developers need only to care about designing a client capable of handling the data exchange and User Interface (UI) of their application.

Section 3.1 presents the state of the art, section 3.2 presents the implementation of Face API, section 3.2.1 presents facial recognition and section 3.2.2 facial emotion recognition.

3.1 State of the Art and Contextualization

With a great number of local software solutions available on the market either incorporated on an OS, such as Windows Hello [32] or Google Smart Lock [33], or by a means of a Software Development Kit (SDK) [34] such as Luxand SDK [35] allowing for custom development of features that integrate facial recognition.

Windows Hello is a feature for Windows 10 [36] that supports facial recognition. This requires a webcam on a PC or laptop, or a special array from Intel [36, 37] providing a hardware solution with its RealSense [36–38] cameras. With a RealSense camera, either built in or an external one, a user can setup Windows Hello to log into the computer without touching a button, verifying the user identity or even complete purchases in the Windows Store [39].

But to set this feature the user also must set a PIN code as an extra layer of security if, for whatever reason, the recognition fails.

As depicted on reference 32 there is also an option to improve the recognition or to update it by removing the current user facial data and setting up a new one. It can also automatically unlock the screen upon a positive match or require an extra security step where the user turns the head left and right to unlock the screen.

Google Smart Lock [27] is a security feature available for Android devices, Chromebooks, Chrome browser and select apps. Like Windows Hello it enables the user to lock or unlock a device supporting not only facial recognition but other biometric data as well. The user can set up a trusted face [39, 40], improve the device's facial recognition or remove a trusted face. Google claims that this of security is less secure than a PIN, pattern or password so it is usually set up with one of those features. Smart Lock does not store photos of the user. Data used to recognize the user's face is kept on the device, apps cannot use the data and it is not backed up on Google servers.

Luxand SDK [29] is a multi-platform software development kit (SDK) that can perform facial recognition, identification and facial feature detection among other features. It also supports a wide range of programming languages such as Microsoft Visual C++, C#, Objective C, VB, Java and Delphi. It has a great adherence from the academic world [41] being used by many companies in their products. It is also able to perform facial detection on a live video feed.

The expansion of cloud computing generated an advent of online services that can replace the use of local features such as facial recognition. One of these cloud-base services is Microsoft's Cognitive Services [42], a set of APIs that provide Image-processing services, among others, to recognize faces or even facial emotions based on picture or video streams. It is backed up by an Artificial Intelligence (AI) platform that provides free access, although limited on the number of calls to the services, to almost all their APIs.

The Face API [43] is a service that can detect human faces and compare with similar ones, organize images into groups based on similarity or even identify previously tagged people in images. For a feature like facial recognition the Face API provides a face verification feature that returns the score of the likelihood of two faces belonging to one person based on two images provided.

Creating a mobile app to implement the cognitive services the choice of programming language must fall to a light-weight yet capable language that is efficient and fast. Given the trends of the global mobile OS market share [44] where the two top trends are Android [45] and iOS [46] the obvious choice, following also what was mentioned in the introduction, falls to React-Native [47]. React-Native is a JavaScript [48] library, based on Facebook's React [49], that has been around since 2015 that allows creating mobile apps for Android, iOS and even for Universal Windows Platform (UWP) [50] using the same code base. It is mainly supported by open-source libraries build on top of it.

The purpose of the app is to implement the use of the cognitive services, facial recognition and emotion recognition, so it can register a user and restrain the use of the app for that user only. When the user unlocks the app, he/she can then access the app options. For this case scenario the options available are limited: take a picture and recognize the facial emotion or lock the app to proceed with facial recognition once again.

3.2 Implementing the Face API

The implementation of the cognitive services is based on a guide [51] provided by Microsoft that details each required step with simple code examples.

The first step was to create a Microsoft account that allows for registering in the Cognitive Services, which gave a subscription key for each API registered and must be included in the Hyper Transfer Text Protocol (HTTP) [52] request header identified by the key-value pair "ocp-apim-subscription-key": "<subscription key>".

In this case there was the need to obtain two subscription keys: one for the Face API, to recognize a face to unlock the app, and another one for the Emotion API, to recognize the emotion on a person's face but since the Emotion API was integrated into the Face API we ended up using only the Face API subscription key.

Next, we needed to create a person group to hold the user face images to cross-match with a picture taken from the mobile app. For this application it was not considered allowing the creation of person groups to maintain the focus of testing the facial recognition and facial emotions for a single user.

To create a person group, we used the PersonGroup – Create API [53] which states that a HTTP PUT request must be made to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/persongroups/{personGroupId} where location is the Geo-located API server to use and personGroupId is the id we want to assign to the person group, which in our case we assigned "facerecog".

The request body contained a JavaScript Object Notation (JSON) [54] object with a mandatory string field, name, for the person group id display name and an optional string field, userData, for user-provided data as shown below:

```
{
  "name": "group1",
  "userData": "user-provided data attached to the person group."
}
```

After manually creating the person group followed the second step, creating the user registration screen so a user can register itself on the "facerecog" person group.

The user must input his name, to be later showed on-screen after a successful recognition (login) and take as many pictures as needed to associate with the user account. Note that although the app registers all pictures taken, and are confirmed by the user, they are not displayed on the user registration screen. Figure 3.1 presents the picture taken by the user and the confirmation screen to associate with user account.

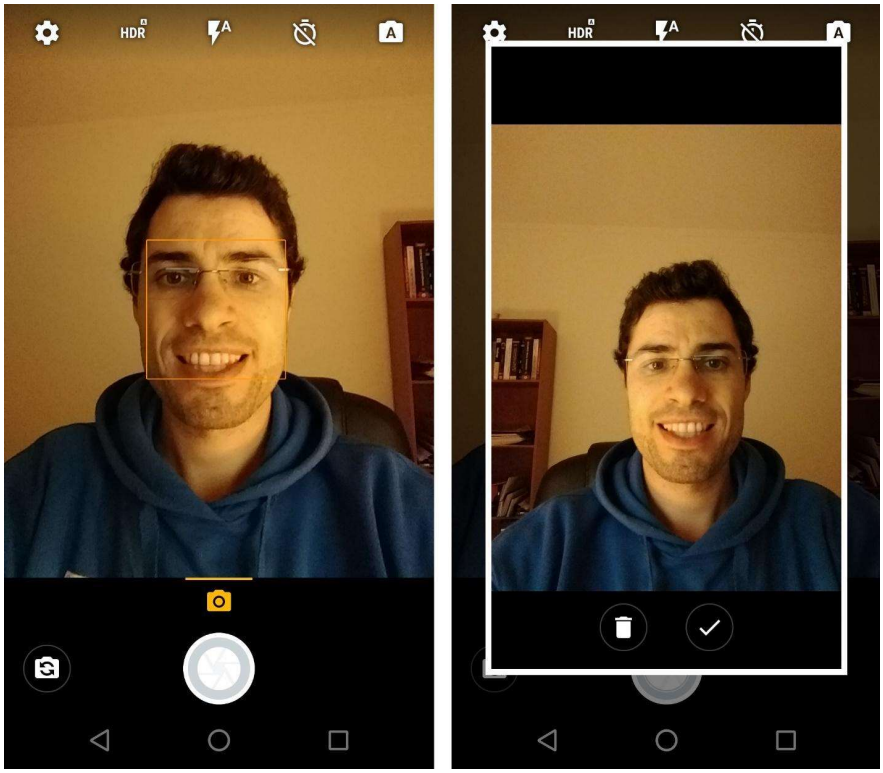

Figure 3.1– left: camera taking picture; right: picture confirmation screen

When all steps are done the user can then press the button “Create user” to initiate the user registration process on the “facerecog” person group. The process starts by creating a user through a HTTP POST request to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/persongroups/{personGroupId}/persons where location is the Geo-located API server and *personGroupId* is “facerecog”. The request body is a JSON object with a string field, name, the display name of the person where the length is limited to 128 characters, and an optional string field, *userData*, for user provided data limited to 16 Kilobytes [55] (kB), e.g.:

```
{
  "name": "Person1",
  "userData": "User-provided data attached to the person."
}
```

If the person is created successfully a person id will be returned in the response body in a JSON object:

```
{
  "personId": "25985303-c537-4467-b41d-bdb45cd95ca1"
}
```

Afterwards the app will process each picture the user has taken and group them with the newly created person id, so it can be used for the matching process, through a HTTP POST request to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/persongroups/{personGroupId}/persons/{personId}/persistedFaces where location is the Geo-located server and *personId* is the retrieved id for the created person on the person group.

The request body contains the image in binary data, so the request must have an additional header with the key-value pair "Content-Type": "application/octet-stream". If the response is successful the response body contains a JSON object with the field *persistedFaceId*, a string, that is a persistent id of the face that was added for the person for that person group, e.g.:

```
{
  "persistedFaceId": "B8D802CF-DD8F-4E61-B15C-9E6C5844CCBA"
}
```

The end process will be something like Figure 3.2.

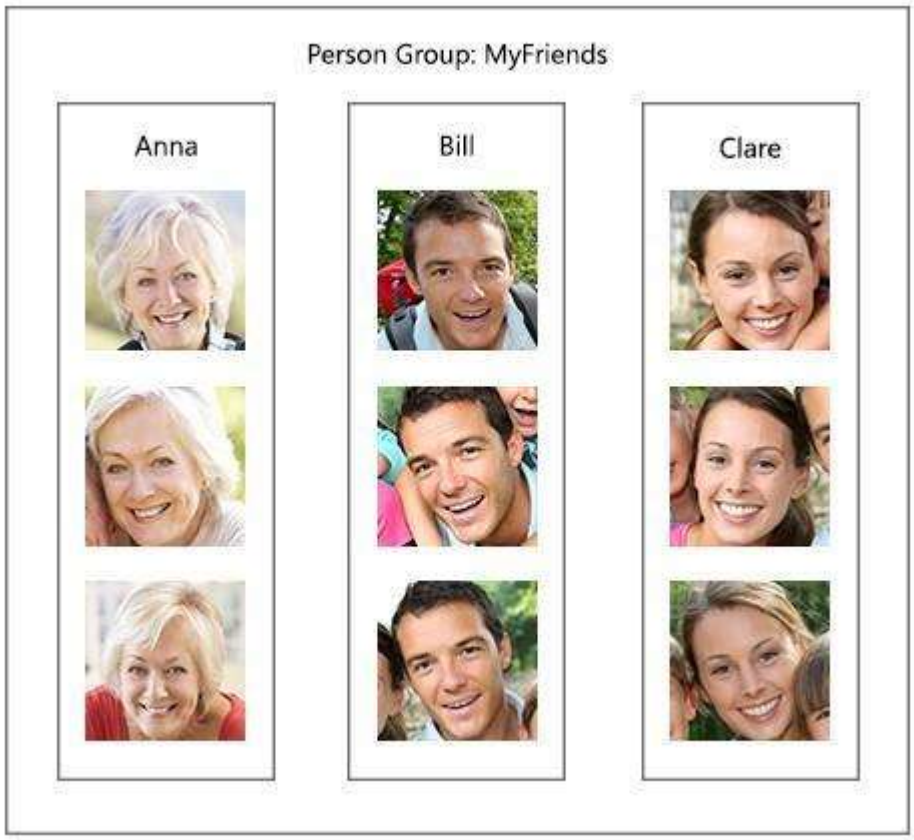

Figure 3.1 – Persons faces grouped inside a person group

After all images are uploaded we dispatched a request to train the person group so that the service could extract all required facial features to be able to perform facial recognition later. The HTTP POST request is then dispatched to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/persongroups/{personGroupId}/train where location is the Geo-located server and *personGroupId* is “facerecog”, with an empty body.

The training task is an asynchronous task where training time depends on the number of person entries and their faces in a person group. Since our person group consisted of one person alone it took only a few seconds. To keep updated about the training status we dispatched a HTTP GET request to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/persongroups/{personGroupId}/training with an empty request body. The response body contained a JSON object with four, string, fields: *status*, *createdDateTime*, *lastActionDateTime* and *message*. The *status* field holds the training status (*notstarted*, *running*, *succeeded* or *failed*) while *createdDateTime* is a UTC date and time that describes person group created time and *lastActionDateTime* is the last modify UTC date and time which can be null when the person group is not successfully trained. The *message* field holds the failure message when training fails:

```
{
   "status": "succeeded",
   "createdDateTime": "12/21/2017 12:57:27",
   "lastActionDateTime": "12/21/2017 12:57:30",
   "message": null
}
```


As these are all asynchronous operations that occur in the background a spinner was shown to inform the user that the process is ongoing.

3.2.1 Using the Face API to unlock the app

After registering the user, the app updated the screen to show a live feed of the frontal camera, displaying an unlock button on bottom as shown on Figure 3.3. This unlock button when pressed takes a picture and sends it to the server to match it against all the facial features that were already extracted when the user registered.

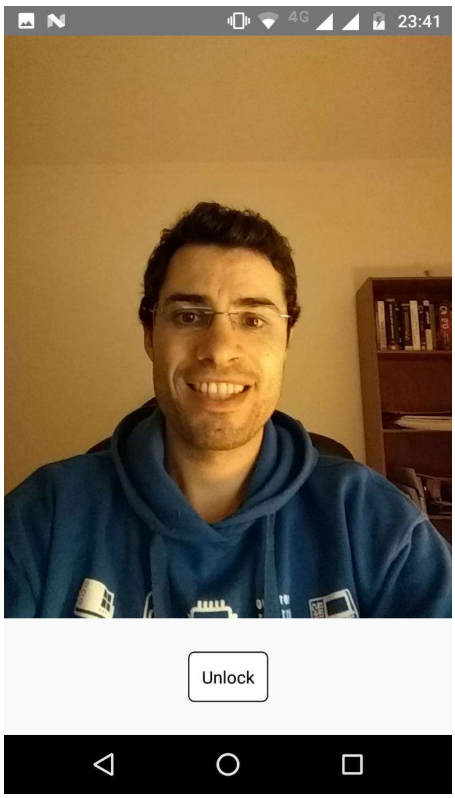

Figure 3.3 – Unlock screen after registering user

The HTTP POST request goes to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/detect[?returnFaceId] where location is the Geo-located server and *returnFaceId* is an optional Boolean request parameter to return the detected face ids or not. In our case we set it to true. The request body contained only binary data from the captured image to match. The received response body contained a JSON object holding an array of face entries. An empty response would indicate that no faces were detected.

For each face entry a JSON object containing the field *faceId* was received. The field *faceId* is of type string and holds a unique face id for the detected face created by the detection API that expires 24 hours after the detection call, e.g.:

```
[
  {
    "faceId": "c5c24a82-6845-4031-9d5d-978df9175426"
  }
]
```

Each *faceId* was retrieved from the received array and a new request was sent to the server to see if there was a match for one of the *faceId*s. The HTTP POST request is sent to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/identify where location is the Geo-located server and the request body contains a JSON object with fields *faceIds* and *personGroupId*. The field *faceIds* is the array that was already retrieved and *personGroupId* is "facerecog", e.g.:

```
{
  "faceIds": [
    "c5c24a82-6845-4031-9d5d-978df9175426",
    "65d083d4-9447-47d1-af30-b626144bf0fb"
  ],
  "personGroupId": "facerecog"
}
```

A successful call returns the identified candidate person(s) for each query face. The response body contains a JSON object with an array of identified faces. Each identified face is a JSON object with fields *faceId* and *candidates*, where *faceId* is a string with the faceId of the query face and *candidates* is an array that contains the identified person candidates for that face (ranked by confidence). If no person is identified, it will return an empty array.

```
{
  [
    {
```

```
        "faceId": "c5c24a82-6845-4031-9d5d-978df9175426",
        "candidates": [
          {
            "personId": "25985303-c537-4467-b41d-bdb45cd95ca1",
            "confidence": 0.92
          }
        ]
      }
    ]
}
```


After receiving the possible identities, we verified each identity and selected the person id with the best confidence value. If a valid person id is returned, we dispatch a new request to get person data for that id. A HTTP GET request is sent to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/persongroups/{personGroupId}/persons/{personIp} where location is the Geo-located server, *personGroupId* is "facerecog" and *personId* is the returned person id, with an empty request body. For a successful response it will contain a JSON object in the response body containing the fields *personId*, *persistedFaceIds*, *name* and *userData*. The field *personId* is a string containing the person id of the retrieved person, *persistedFaceIds* is an array with a *persistedFaceId* for each registered face for that person, *name* is a string with the person's display name and *userData* is the user-provided data, e.g.:


```
{
  "personId": "25985303-c537-4467-b41d-bdb45cd95ca1",
  "persistedFaceIds": [
    "015839fb-fbd9-4f79-ace9-7675fc2f1dd9",
    "fce92aed-d578-4d2e-8114-068f8af4492e",
    "b64d5e15-8257-4af2-b20a-5a750f8940e7"
  ],
  "name": "Ryan",
  "userData": "User-provided data attached to the person."
}
```


We then updated the user data within the app and showed the user display name on the top left corner, to show that the user was identified, and we also displayed two buttons on the bottom: "Read emotion" and "Lock app" as shown on Figure 3.4.

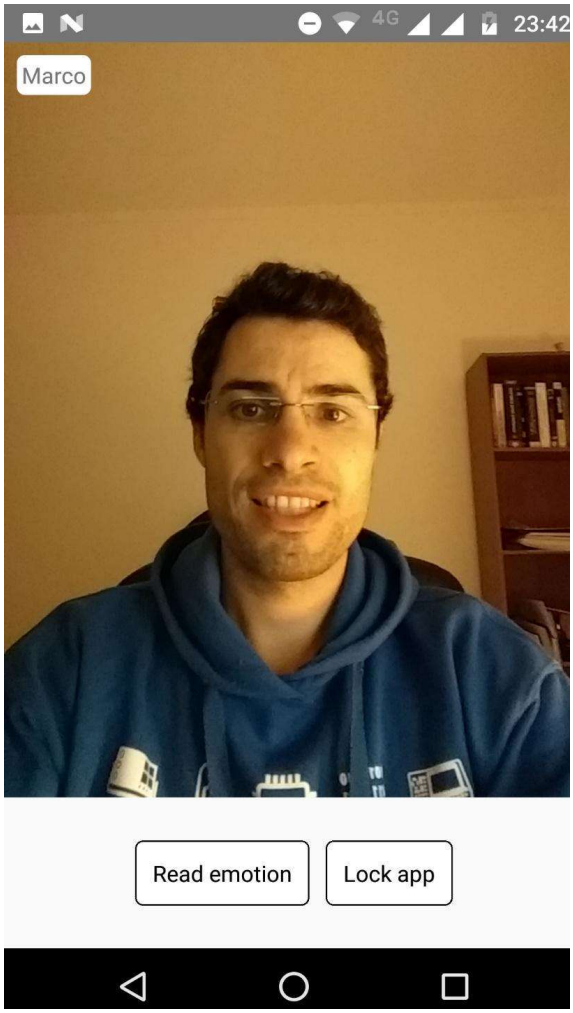

Figure 3.4 – Unlocked screen options

If an error had occurred during the unlocking process or if the user was not correctly identified the spinner would stop showing and the “Unlock” button would be displayed once again.

Each of the endpoints used to identify the user is a distinct microservice with a single responsibility (detect faces, identify persons within a person group and fetch person data) that when combined represent the full feature to identify a user.

3.2.2 Using the Face API to recognize an emotion

To recognize an emotion a user must express an emotion type even if it is neutral. while aiming the frontal camera carefully enough so that the light conditions are good and in a frontal or near-frontal face position to have the best results. When set, the user must click the button “Read emotion” to take a picture and send the picture to the server for emotion recognition.

A HTTP POST request was sent to the URL https://[location].api.cognitive.microsoft.com/face/v1.0/detect?returnFaceAttributes=emotion, where location is the Geo-located server and we added the request parameter returnFaceAttributes=emotion to specify that we only wanted the emotion data. The request body contained the picture taken in binary data. If successful, the response body will contain an array of face entries and their associated face attributes with emotion scores, ranked by face rectangle size in descending order. An empty response indicates that no faces were detected. An emotion entry contains the fields *faceId*, *faceRectangle* and *faceAttributes*. The field *faceRectangle* is an object containing the rectangle location of face in the image and *faceAttributes* is an object that holds the emotion object scores, e.g.:


```
[
 {
  "faceId": "01f413da-a3af-4c28-b249-eb64f8275c1c",
  "faceRectangle": {
   "left": 68,
   "top": 97,
   "width": 64,
   "height": 97
  },
  "faceAttributes": {
  "emotion": {
   "anger": 0.00300731952,
   "contempt": 5.14648448E-08,
   "disgust": 9.180124E-06,
   "fear": 0.0001912825,
   "happiness": 0.9875571,
   "neutral": 0.0009861537,
   "sadness": 1.889955E-05,
   "surprise": 0.008229999
  }
 }
}]
```


The best emotion was then chosen, and the captured image appeared as a modal view (on top of the live video feed) with the name of the emotion and the associated score. Figure 3.5 presents five different emotions that were recognized: happiness, neutral, surprise, sadness and fear.

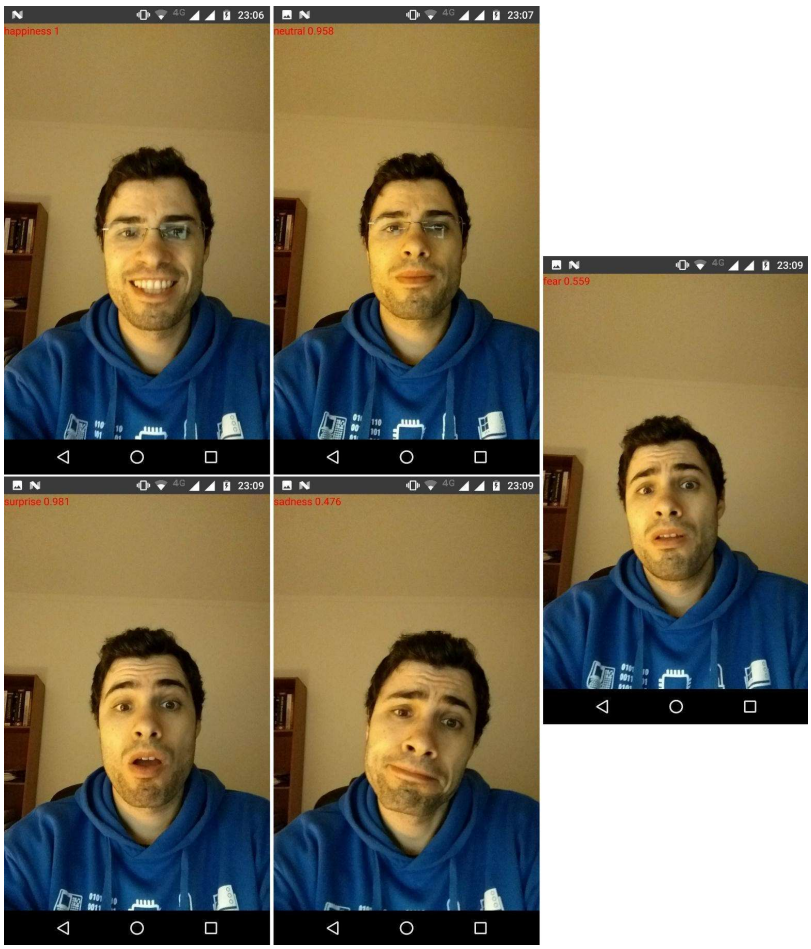

Figure 3.5 – Emotions recognized and their score. Top left: happiness (1), top right: neutral (0.958), bottom left: surprise (0.981), bottom right: sadness (0.476), middle: fear (0.559)

The endpoint used for the emotion recognition is another example of a distinct microservice used for one purpose alone (a single responsibility). The whole procedure can be seen as the initial step of an application that could be used in conjunction with another set of custom microservices, for an enterprise application for example, acting as an extra security layer or as the full feature login.

4 Replacing an Enterprise Resource Planning application

As described on Chapter 1 the internship is focused on replacing the current web application, an Enterprise Resource Planning (ERP) [56] application, with a new one following a microservice architecture using NancyFX as the framework of choice.

NancyFX (Nancy) is an open source .NET-based web framework built with the explicit goal of giving developers an easy approach for developing web applications and services. It even came up with its own term for their approach called the Super Duper Happy Path to describe the core values behind it.

Nancy comes pre-configured with templates that contain defaults for everything leaving room for few or no tweaks at all to just run it out of the box. It is easy and highly customizable, it offers customization right to the core in every aspect allowing for replacements of almost every component. It has been designed to be flexible enough so there is little friction between the code and its APIs. It is also built in a test-driven fashion allowing for Test Driven Development (TDD) [57] including its own library for testing. Nancy is not built on any specific hosting technology and has out of the box support for ASP.NET//IIS [58], WCF [59], self-hosting and OWIN [60] meaning of wide variety of hosting types are available.

Nancy handles calls by forwarding requests to the matching endpoint, a NancyModule type, where the endpoint will call the required service or services to fulfill the request. The request is either fulfilled or an error is thrown and will back track the original request stack where it is handled and returned to the caller with proper status code as seen in figure 4.1.

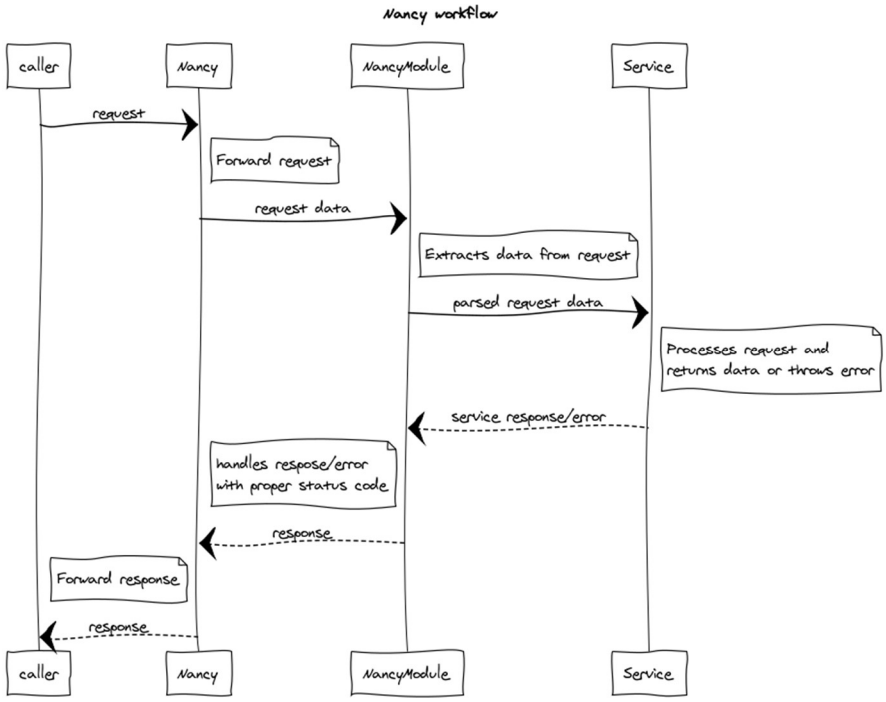

Figure 4.1 - Sequence diagram for the workflow of Nancy

4.1 Nancy modules

Nancy handles calls to endpoints, URIs, through implementations of a Nancy module, meaning that a custom module must inherit from NancyModule to handle calls. These custom modules are automatically discovered by Nancy so there is no configuration required to hook them to the startup process and register the declared routes. The route declaration is done using Nancy's internal Domain Specific Language (DSL) [61] for dealing with HTTP.

A custom module may adopt certain patterns required for Dependency Injection (DI) [62] where the preference goes to the Constructor Injection pattern due to its easiness, readability and reliability. This also contributes heavily for testing and programming to an interface [63] since both are related and have gained a notorious rise in popularity with the introduction of DI. As a result custom modules are cleaner and more readable since all dependencies can be replaced with easiness because Nancy uses a bootstrapper, with the help of the framework TinyIoC [64], that allows mapping a type to a given implementation and fulfills any dependencies throughout the application based on the Inversion of Control (IoC) [65] principle. A Nancy module is the first entry point for a request to be fulfilled and calls the required services to return a response.

4.1.1 Implementing a module

Figure 4.2 represents a sequence diagram for an example request of the current UTC DateTime to a Nancy endpoint [66].

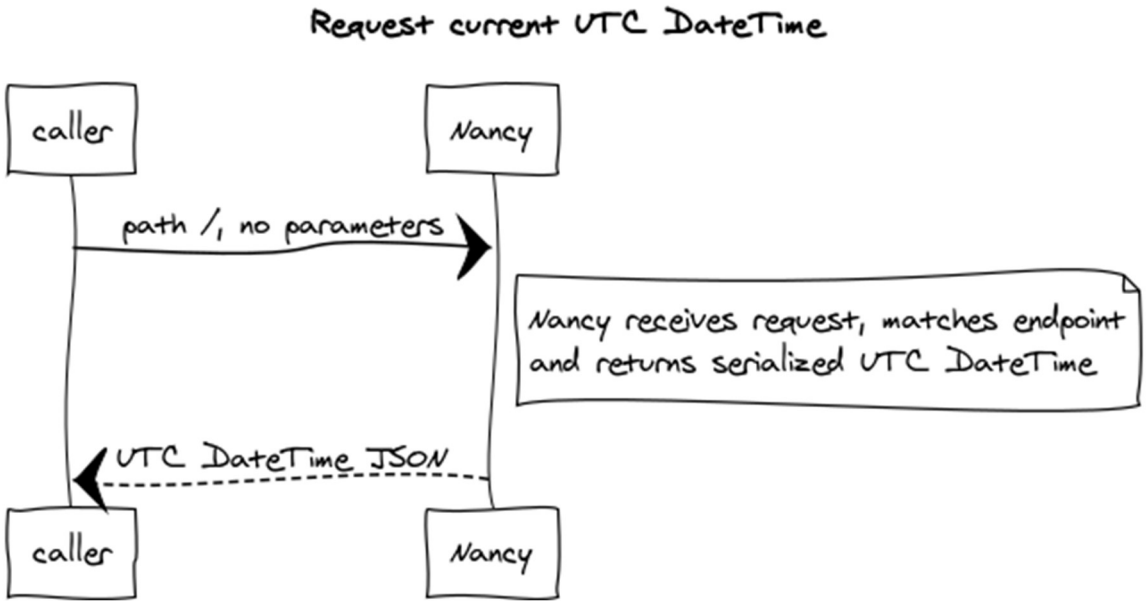

Figure 4.2 - Sequence diagram for an example request

The code to handle the request in Nancy is as follows:

```
namespace NancyModuleExample
{
        using System;
        using Nancy;

        public class CurrentUtcDateTimeModule : NancyModule
        {
                public CurrentUtcDateTimeModule()
                {
                    Get("/", _ => DateTime.UtcNow);
                }
        }
}
```

We declare a namespace [67] for the module, in this case NancyModuleExample, to specify the scope. We declare the class CurrentUtcDateTimeModule as having a base (parent) class from NancyModule which is what makes our class automatically discoverable by Nancy and allows us to setup a route declaration.

The route declaration is done inside the body of the public constructor CurrentUtcDateTimeModule() with the required HTTP method [68] (Nancy supports DELETE, GET, HEAD, OPTIONS, POST, PUT and PATCH) for each route we wish to handle in our module. In this case we declared a GET for the path / with the expression Get("/",...). This tells Nancy that any GET request to / should be handled by this method. The lambda expression [69] _=> DateTime.UtcNow is responsible for handling capture segments, if any, and handling the response. In our case there are no capture segments to handle so _ is used, a Nancy convention for lambda parameters that are not used on the right side of the lambda arrow, so we just need to return the current Coordinated Universal Time (UTC) [70].

To access this, we could point a browser to the application address, e.g. http://localhost:5000, but it would display an error page since we are only returning data. Nancy also allows responses for endpoints with views, web pages, but when handling microservices that require data exchange alone this is not used. A proper call to consume this microservice would be from another microservice, for example, to the given address and the reply would either come in JSON or Extensible Markup Language (XML) [71]. An example of a response serialized as JSON would be "2016-06-06T19:50:09.2556094Z".

4.1.2 Managing multiple modules

To handle a diversity of modules we can have them on a single project, all modules are centralized on a single microservice, e.g., AllModulesMicroservice. What this means is that this microservice is solely responsible for handling modules and their respective endpoints, falling under the SRP discussed in Section 2. Although it is responsible for all endpoints it does not mean that they need to be registered in a single code file. For such an approach each module can fall under a category and/or each capability under a module class, e.g.:

```
// ShoppingCart.cs
namespace MyApp.Modules.ShoppingCart
{
        public class ShoppingCartModule : NancyModule
        {
                public ShoppingCartModule()
                {
                        Get("/shoppingCart/content", _ => {
                                // Return data for shopping cart
                        });
                }
         }
}
// Products.cs
namespace MyApp.Modules.Product
{
        public class ProductModule : NancyModule
        {
                public ProductModule()
                {
                        Get("/product/search", _ => {
                                // Return search result
                        });
                }
         }
}
```

4.1.3 Endpoints with capture segments, RegEx and request parameters

Nancy supports capture segments [72] in a route or request parameters in the body or as query parameters. To access the capture segments of a route we need to access the parameters declaration of the lambda expression for the route handler, e.g.:

```
Get("/product/{name}", parameters => {
        // Gets the product name
        var productName = parameters.name;
});
```

This is a variable declared on the route that is passed to the route handler. The route can even be constrained [73] to a given type for each route segment. It can be useful for certain scenarios where the endpoints should handle calls only for a specific type of data, e.g.:

```
Get("/product/{serialNumber:long}", parameters => {
        // Gets the product serial number
        var productSerialNumber = parameters.serialNumber;
});
```

Even regular expressions are supported on an endpoint, allowing for a more dynamic route matching specification, e.g:

```
Get("/user/age/(?<age>[\d]{1,2])", parameters => {
        var age = (int)parameters.age;
        // Returns all users that are of a certain age
});
```

But these examples are for certain scenarios where we need to request single parameters. If we need to call endpoints with more request data we can also implement handling calls to a route that has query parameters in the request. This is a more common approach for search scenarios or other scenarios where passing the data explicitly on the route would not be the best approach. Consider searching for a product with name, EAN and serial number as criteria where the request would be http://www.mywebsite.com/product/search?name=Sonasol&ean=12345&serialNumber=6789, e.g.:

```
Get("/product/search", _ => {
        // Gets the product name to search
        var productName = this.Request.Query[“name”];
        var productEAN = this.Request.Query[“ean”];
        var productSerialNumber = this.Request.Query[“serialNumber”];
        // Returns search results
});
```

In this scenario we access the request parameters not through the lambda expression input parameter but through Nancy's Request object which comes inherited from NancyModule. This allows access to the request query and its data.

Another approach would be to pass the data in the body of a request and bind it to a model. This is called model binding. The model for a product could be:

```
public class Product
{
        public long Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
}
```

The request would have a body, in Json, like:

```
{
        "Id " : 1
        "Name" : "Sonasol"
        "Price" : 10.5
}
```

The data in the request body would be binded through Nancy's Bind method, e.g.:

```
Get("/product/search", _ => {
        // Gets the product data to search
        var productToSearch = this.Bind<Product>();
        // Returns search results
 });
```

This way the data in the body of the request can be mapped automatically to the model for all matched members and if we want to know if any member isn't mapped we would need to verify its members, either manually or with Nancy's Fluent Validation library [74].

4.1.4 Dependency Injection

Dependency Injection is a design pattern that aims at abstracting the dependencies of a class either through Property Injection (PI) or through Constructor Injection (CI). PI is a less used pattern and is generally viewed as an optional injection pattern whereas CI is the most preferred one due to the simplicity of having all dependencies being injected through the constructor of a class and because the dependencies are abstracted through an interface type.

When declaring a module any dependencies the module may have should be abstracted through interfaces creating loose coupled dependencies that can easily be replaced during unit tests. This also helps to keep the scope of the dependencies in a module simple and refactor them, when required, much quickly than through a class type since the coupling is abstract. An example of CI is the injection of the actual product search service on a product module, e.g:

```
namespace MyApp.Modules.Product
{
        public class ProductModule : NancyModule
        {
                private readonly searchService;

                public ProductModule(ISearchService searchService)
                {
                    this.searchService = searchService;
                    Get("/product/search", _ => {
                                // Return data from searchService
                        });
                }
    }
}
```

When ProductModule is instantiated, it is injected with a dependency that implements the interface ISearchService. This means that any class that implements ISearchService can be injected to this module. Nancy however has a bootstrapper [75] that allows registering types to be injected throughout the application meaning that there is no explicit necessity of sufficing the dependencies where required because Nancy will fulfill them once the app starts. Even if there are multiple dependencies this will not be a problem since Nancy will fulfill them all as long as the required types are registered.

The simplest way to achieve this is through a custom bootstrapper that needs to derive from the base bootstrapper, DefaultNancyBootstrapper, and override the configuration of the request container as shown in the following example:

```
namespace MyApp.Modules
{
        public class Bootstrapper : DefaultNancyBootstrapper
        {
                protected override void ConfigureRequestContainer(TinyIoCContainer container, NancyContext context)
```

```
            {
                container.Register<ISearchService, SearchService>();
            }
    }
}
```

In the example we register the search service implementation, SearchService, to the type ISearchService so that each time it is requested the container will provide a new instance of SearchService.

4.1.5 Controlling the content to return

When the request is handled and the module returns the result, the response is sent through a content negotiation [76] pipeline if it is not a Response object or if it is not castable to a Response object. Most often the negotiation is controlled through the helper Negotiate [77] which exposes an API that helps control the content negotiation in a user-friendly way, e.g.:

```
public ProductModule(ISearchService searchService)
{
        this.searchService = searchService;
        Get("/product/search", _ => {
                // Gets the product name to search
                var productName = this.Request.Query[“name”];
                return Negotiate
                .WithModel(this.searchService.FindProductByName(name)
                .WithStatusCode(HttpStatusCode.OK);
        });
}
```

In this example the product search endpoint retrieves the product name to search for from the request body and passes it to the search service. The search service result goes through content negotiation where the model is adapted to the most suitable format for requested media types that come from the Accept headers [78]. Also, with the returned body we set a 200 - OK HTTP status code [79] to indicate that the operation occurred without errors.

4.1.6 Handling errors

While handling a request an error might occur, so it is necessary to add error handling inside the module. It is usually a good practice to add support for unhandled exceptions inside a module. This

prevents any corner case scenario that might not have been covered through any type of tests to be taken care and not crash the app, e.g.:

```
public ProductModule(ISearchService searchService)
{
        this.searchService = searchService;
        Get("/product/search", _ => {
                try
                {
                        // Return data from searchService
                }
                 catch(SearchServiceException)
                 {
                        // Handle exception raised from the call to searchService
                        Return Negotiate
                                .WithModel(new { Message = “The request was not ok” })
                                .WithStatusCode(HttpStatusCode.BadRequest);
                 }
                catch(System.Exception)
                {
                        // Some other logic to handle the error
                        return HttpStatusCode.InternalServerError;
                 }
        });
}
```

In this case we handle a SearchServiceException by returning a body with a custom message and a HTTP status code 400 - Bad Request to indicate that the request was malformed. We could set a more complex body, or use a class, to return a more detailed object to help identify the problem with the request. The return of the HTTP status code is also dependent on the type of exception since we want to return an appropriate error value for the request. On the case of an unhandled exception it makes sense to return 500 - Internal Server Error since there is no specific handling for an unhandled exception.

4.1.7 Unit testing a module

To test a module, we need a test project. Each test project mimics the project of the microservice under test in terms of folder and file layout and tests only the same scope of the microservice. This means that the tests will only verify partial behavior from the expected behavior on a call to an endpoint. All

other remaining aspects of that call to an endpoint have to be tested separately. Considering the example of the Product module:

```
namespace MyApp.Modules.Product
{
	public class ProductModule : NancyModule
	{
		private readonly searchService;

		public ProductModule(ISearchService searchService)
		{
		    this.searchService = searchService;
		    Get("/product/search", _ => {
				try
				{
					// Gets the product name to search
					var productName = this.Request.Query[“name”];
					return Negotiate
						.WithModel(this.searchService.FindProductByName(name))
						.WithStatusCode(HttpStatusCode.OK);
				}
				catch(System.Exception)
				{
					Return Negotiate
						.WithModel(new { Message = “Something went bad!” })
						.WithStatusCode(HttpStatusCode.InternalServerError);
				}
			});
		}
	}
}
```

We have a search service being injected to the module and the product name to search for in the request body. We pass it to the search service, so the method FindProductByName can search for a product that can be matched through the product name and return the result. The endpoint would require tests such as verify if the product name is being passed on to the FindProductByName method from

SearchService or if a response status code is returned when expected. A test to verify the product name could be:

```
namespace MyApp.Tests.Modules.Product
{
        using Moq;
        using Nancy.Testing;
        using Xunit;

        public class ProductModuleUnitTests
        {
                [Fact()]
                public It_should_receive_shampoo_as_product_name()
                {
                        // Arrange
                        var searchServiceMock = new Mock<ISearchService>();
                        var browser = new Browser((with) =>
                        {
                                with.Module<ProductModule>();
                                with.Dependency(searchServiceMock.Object);
                        }
                        // Act
                        browser.Get("/product/search", (with) =>
                        {
                                with.HttpRequest();
                                with.Query("name", "shampoo");
                        }
                        // Assert
                        searchServiceMock.Verify(searchService =>
                                        searchService.FindProductByName("shampoo"));
                }
         }
}
```

The test follows the Arrange, Act and Assert pattern [80] and each section is commented for identification purposes. We use Xunit as the test runner and we identify the test method with the attribute [Fact()]. We start by creating a mock [81] for the search service using the framework Moq [82]. It creates

an abstraction of a type based on an interface, so we can control how the instance can behave or to enable verification of methods or properties. On the next step we create the browser context, meaning we fulfill the dependencies for that module. This way we have puppeteered the response mechanism when a call to the endpoint is performed.

With every behavior defined we proceed with the call to the endpoint, configuring the call to be made with HTTP and a query parameter with the value “shampoo”. This is a fake browser that Nancy provides and enables calling endpoints with all supported methods and is very customizable.

Since the call is done we need to verify if the search service received the expected query parameter value. With the created mock we use the method Verify to configure a validation expression that if matched the test passes, otherwise it will throw an assert exception and cause the test to fail.

If we wanted to test if unhandled exceptions would return the expected status code, we could write a test like this:

```
namespace MyApp.Tests.Modules.Product
{
        using FluentAssertions;
        using Moq;
        using Nancy.Testing;
        using Xunit;

        public class ProductModuleUnitTests
        {
                [Fact()]
                public It_should_handle_unhandled_exceptions()
                {
                        // Arrange
                        var searchServiceMock = new Mock<ISearchService>();
                        searchServiceMock.Setup(instance =>
                                instance.FindProductByName(It.IsAny<string>())
                                .Throws<Exception>();
                        var browser = new Browser((with) =>
                        {
                                with.Module<ProductModule>();
                                with.Dependency(searchServiceMock.Object);
                        }
                        const HttpStatusCode expected = HttpStatusCode.InternalServerError;
```

```
                    // Act
                    var actual = browser.Get("/product/search", (with) =>
                    {
                            with.HttpRequest();
                            with.Query("query", "shampoo");
                    }
                    // Assert
                    actual.Should().Be(expected, because: "Unhandled exceptions should return 500 -
Internal Server Error");
                }
            }
}
```

In this test we changed Moq to handle any call to FindProductByName with any text to throw an exception. This means that any value we pass to the method will always raise an exception of type Exception. We also set the expected value to be InternalServerError and we save the fake browser response into the variable actual. Our assert is a little bit different because we are using FluentAssertions [83] to assert the expected value in a Business Driven Development (BDD) style. We set the expectation of variable actual to be the same as variable expected and we define an error message in case it fails to match, for the parameter because.

4.2 Data Access

Data repositories are all centralized on one microservice that uses Entity Framework (EF) [84] to access a database. EF creates an abstraction to ease the access to the data through a code first approach [85]. By creating a model, which is stored on a models microservice, to represent a data table the data access becomes easier to represent. A nice example is a products table to store products to search and retrieve when the search is matched. The model for the products table could be:

```
namespace MyApp.Models
{
        public class Product
        {
                public long Id { get; set; }
                public string Name { get; set; }
                public decimal Price { get; set; }
        }
```

```
}
```

A simple data representation that holds the product id, product name and price with all members having read/write access. The data access microservice is the bridge connecting models, and the data retrieval, where EF is configured to use existing models and the data access is made available through a data access pattern such as DbContext [86], Unit of Work [87] or Repository [88].

Whenever a service needs to access data from a database it uses one of these patterns from the data access microservice which in turn uses one the models from the models microservice to complement the access to the data. This way the service knows exactly what type of data is retrieved from the database and can be consumed right away, e.g.:

```
namespace MyApp.DataAccess
{
        public interface IProductsRepository
        {
                Product FindProduct(name);
        }

        public class ProductsRepository : IProductsRepository
        {
                Product FindProductByName(string name)
                {
                        // Access the database to find the product by name
                        return dbContext.Products.Find(product => product.Name.Equals(name));
                }
        }
}
```

On this example we assume dbContext has been implemented and we have access to it. As dbContext can access the database, has the Product model mapped to perform a search to match a product by its name it returns the first match to the caller.

4.3 Services

Services play an important role on micro-services, they are the ones that get the job done per se. When a module is called it subsequently calls a service or a set of services to fulfill the initial call and the retrieved data is mapped onto a Data Transfer Object (DTO) [89] to be returned by the module.

A DTO is a simple data container used for moving data between layers of an application. It holds only state, no behavior, with all members having read/write access. It is a standard procedure to encapsulate data to be transferred over the wire. The data we want to transfer to the caller can come from multiple sources since a service can also call other services and map the data retrieved from those calls to a DTO. If the data retrieved from the services would to be changed the DTO would remain unchanged but the mapping of the data would need to be updated. DTOs are stored on their own microservice.

Services are located on their own microservice to stay decoupled from any other dependencies. Considering the SearchService we have seen referenced previously on ProductModule:

```
namespace MyApp.Services
{
	using AutoMapper;
	using MyApp.DTOs;
	using MyApp.DataAccess;

	public interface ISearchService
	{
		SearchResult FindProductByName(string name);
	}

	public class SearchService : ISearchService
	{
		private readonly IProductsRepository productsRepository;

		public SearchService(IProductsRepository productsRepository)
		{
			this.productsRepository = productsRepository;
		}

		SearchResult FindProductByName(string name)
		{
			// Access the database to find the product by name
			// and store the result
			var result = this.productsRepository.FindProduct(name);
			return Mapper.Map<SearchResult>(result);
		}
```

```
        }
}
```

For the SearchService to find a product by name it accesses the products table through the products repository and the result is mapped to the type SearchResult.The SearchResult type is a class that is defined as:

```
namespace MyApp.DTOs
{
        public class SearchResult
        {
                public string Name { get; set; }
                public decimal Price { get; set; }
        }
}
```

In the case of our SearchResult it holds the name and price of the product found. They would be filled in automatically by AutoMapper, a framework that maps objects that have the same properties or that have a mapping configuration. What we end up getting from the call to the search endpoint is an object of type SearchResult that is then converted to the appropriate media (JSON, XML or other supported type) by Nancy.

To test the service, we could stub the response from the database to see if the mapping would be executed correctly, e.g.:

```
namespace MyApp.Tests.Services
{
        using FluentAssertions;
        using Moq;
        using MyApp.Models;
        using MyApp.DTOs;
        using Xunit;

        public class SearchServiceUnitTests
        {
                [Fact()]
                public It_should_return_the_matched_product()
                {
                        // Arrange
                        var product = new Product
```

```
                {
                        Name = “Sonasol”,
                        Price = 3.95m
                };
                var repoStub = new Mock<IProductsRepository>();
                repoStub.Setup(instance =>
                        instance.FindProduct(It.IsAny<string>())
                        .Returns(product);
                var sut = new SearchService(repoStub.Object);
                var expected = new SearchResult
                {
                        Name = “Sonasol”,
                        Price = 3.95m
                };
                // Act
                var actual = sut.FindProductByName(“Sonasol”);
                // Assert
                actual.ShouldBeEquivalenetTo(expected);
        }
    }
}
```

On this test we create the product to return from the products table and we stub the call to the repository, meaning any call with any string value will always return the same result. Next we create the System Under Test (SUT) and inject the stub into it. Afterwards we create the expected search result and we act to receive the actual value from the call. Lastly, we assert that the actual object we received is equal to the expected we had created.

4.4 External Services

To access external services, we create a microservice for each external service. Once again it follows the SRP and keeps the scope to each microservice. This also brings advantages when mapping data from combined services and external services since we can choose what data to map for each specific purpose. In the SearchService scenario we could add a call to an external service that also provided products data and, in the end, compare the retrieved data, or timestamp of the product data, and choose which one would be most the most recent, e.g.:

```
public class SearchService : ISearchService
{
	private readonly IProductsRepository productsRepository;
	private readonly IProductsExternalService productsExternalService;

	public SearchService(
		IProductsRepository productsRepository,
		IProductsExternalService productsExternalService
	)
	{
		this.productsRepository = productsRepository;
		this.productsExternalService = productsExternalService;
	}

	SearchResult FindProductByName(string name)
	{
		// Access the database to find the product by name
		// and store the result
		var result1 = this.productsRepository.FindProduct(name);
		// Access the external database to find the same product
		var result2 = this.productsExternalService.FindProduct(name);
		// Compare and store the latest product information
		var result = DateTime.Compare(result1.TimeStamp, result2.TimeStamp) > 0 ? result1 :
result2;
		return Mapper.Map<SearchResult>(result);
	}
}
```

External services are implemented the same way as any other microservice, following the same styles and guidelines. In the previous example we could see that there is no difference in the code on how to use an external service. The same applies for the unit tests, they follow the same structure as tests for a service with the particularity of having the connection components mocked just as you would mock the database of the products repository.

4.5 Practical implementation

All the previous sections describe, although through small examples, what consists of the new ERP application. Despite not having authorization from Evodeck Software to demonstrate real world code, through samples or screenshots, we can exemplify real world code with the open-source project NancyAzureFileUpload [90], which uses Nancy to serve an endpoint to upload files to Microsoft Azure [91].

Figure 4.3 is partial content of the upload endpoint implementation. This content exemplifies uploading a file as a multipart-form [92] where the posted file is fetched from the request body, if it exists, otherwise the posted file value is set to null. The posted file will then be uploaded to Azure Storage using a formatted URL and stored credentials through an asynchronous stream [93]. After the posted file has been uploaded a DTO is created to hold dispatch information from the request body and the URLs for the stored file on Azure Storage.

This newly created dispatch information object is then added to a database, asynchronously, by using a file service that is injected on the module constructor. If no file is uploaded the endpoint simply returns the message “No files uploaded”.

On both cases Nancy will reply with a 200 - OK HTTP status code [94] since no explicit error handling or error throwing is implemented.

```
public class UploadModule : NancyModule
{
    public UploadModule(IDispatchFileService _fileService,
                        IConfiguration _config)
    {
        Post("/upload", async (args, ct) =>
        {
            var postedFile = Request.Files.FirstOrDefault();
            var queryInfo = Request.Form;
            DispatchFile fileInfo;

            if(postedFile != null)
            {
                //Check file type
                var url = $"https://{_config["StorageAccount:User"]}.blob.core.windows.net/{_config
                  ["StorageAccount:containerName"]}/{postedFile.Name}";
                var secondary = $"https://{_config["StorageAccount:User"]}-secondary.blob.core.windows.net/{_config
                  ["StorageAccount:containerName"]}/{postedFile.Name}";

                //Upload file to Azure Storage
                var creds = new StorageCredentials(_config["StorageAccount:User"], _config["StorageAccount:Key"]);
                var blob = new CloudBlockBlob(new Uri(url), creds);

                await blob.UploadFromStreamAsync(postedFile.Value);

                // Create object to save to table
                fileInfo = new DispatchFile
                {
                    DispatchId = Convert.ToInt32(queryInfo?.DispatchId?.Value ?? 0),
                    Filename = postedFile.Name,
                    Filetype = postedFile.ContentType,
                    PrimaryUrl = url,
                    SecondaryUrl = secondary,
                    ItemType = queryInfo?.ItemType?.Value
                };
                // Call our Dapper IDispatchFileService.cs
                await _fileService.Add(fileInfo);

            }
            else
            {
                return "No files uploaded.";
            }

            return fileInfo;
        });
```

Figure 4.3 - Upload file endpoint (adapted from [95])

Figure 4.4 is the content of DispatchFileService, the implementation injected to UploadModule, where we see how the posted file is added to the database. The service is injected with a configuration for the database connection string that is then used to create a new connection every time it needs to add data to the database. The data to add is parsed from the DTO created in UploadModule by matching the declared variable names on query string with the public members of the DTO.

```
public class DispatchFileService : IDispatchFileService
{
    private string dbConn;

    public DispatchFileService(IConfiguration _dbConnString)
    {
        dbConn = _dbConnString.GetConnectionString("DefaultConnection");
    }
    internal IDbConnection Connection
    {
        get
        {
            return new SqlConnection(dbConn);
        }
    }
    public async Task Add(DispatchFile postedFile)
    {
        string sQuery = @"INSERT INTO dbo.Files
        (DispatchId, PrimaryUrl, SecondaryUrl, ItemType, Timestamp,
        Filename, Filetype)" +
        "VALUES(@DispatchId, @PrimaryUrl, @SecondaryUrl, @ItemType, getdate(), @Filename, @Filetype)";
        using (var conn = Connection)
        {
            conn.Open();
            await conn.ExecuteAsync(sQuery, postedFile);
        }
    }
}
```

Figure 4.4 - Dispatch file service (adapted from [96])

5 Conclusions

Microservices play an essential role in today's web and mobile applications. Not only they have evolved to the point of being able to suffice many of the technical challenges that were not possible a couple of years ago, but they do it with a straightforward approach and always include the need of testing to be sure they are fit for production.

The mobile application is a simple yet nice example of how easy it is to use microservices and that this is something that can be achieved without an extensive knowledge of web applications. It also shows that today's mobile technologies are targeted at easy to use, or code first, approach to reduce the development process, duration and deployment interval.

Chapter 4 described most, if not all, the scenarios where microservices are needed. For data access, for external services, internal services and the endpoints themselves. Comparing their implementations with the general specifications of microservices shows that there is not such a straightforward follow up and both approaches have different needs. In our case having a microservice per technical trait instead of per functionality feels about right for the scope of the project and still maintains the SRP. There are certain microservices that tend to get a bit big or a little bigger than what we would want but that is to be expected when following this SOA.

The chosen SOA does not apply for the remaining specifications. The microservices are not individually deployable, updating a microservice following CD would cause, for example, all endpoints to fail and could result on a severe down time for the customer.

The microservices run in one process alone since there must be an entry point microservice and it usually falls to the endpoints microservice with this type of SOA. The data store relied on one microservice alone with one database technology.

As far as replaceability each one of them is replaceable but since they carry one technical trait it makes them harder and longer to replace with the continuous growth of the project. A feasible approach to replace each microservice would be to start replacing the implementations one by one but it would prove to be very tedious and time consuming.

The team size is adequate for the project needs but replacing a microservice might reveal tricky.

One could claim that since the general specification is more focused on each functionality it pays off on the long run, but we believe that that also depends on the size of the project, duration and team or teams' size. The team itself is also a great factor on this analysis since the team needs to be well coordinated and have the same level of knowledge to support a steady pace of development.

In sum comparing the general specifications with the internship experience shows that there are certain projects and/or scenarios where they would be applicable, but it wasn't our case. The common

specification is the SRP which was applied on a much larger scope than specified but we believe it remained propitious on the outcome of the project so far.

Every example on Chapter 4 is based on the practical implementation of the internship, where we believe they contained enough technical detail, without disclosing any business information, to demonstrate how a request is handled by a microservice and how it is possible to test the expected behavior of a microservice.

Future work will consist of the continuous integration of the ERP until the customer sees fit all business requirements, having to integrate different external service providers, each one with different communication protocols, provide continuous formation about the new ERP to the customer employees and customer support for bug fixes or additional requirements.

6 References

1. Skill shortages in Europe: Which occupations are in demand – and why. http://www.cedefop.europa.eu/en/news-and-press/news/skill-shortages-europe-which-occupations-are-demand-and-why. Accessed 28 Apr 2018
2. 13 Best Programming Languages to Learn in 2017! - Usersnap. https://usersnap.com/blog/programming-languages-2017/. Accessed 28 Apr 2018
3. The 10 easiest programming languages to learn. https://www.techrepublic.com/article/the-10-easiest-programming-languages-to-learn/. Accessed 28 Apr 2018
4. This Week in Numbers: Industry Demand for React Developers - The New Stack. https://thenewstack.io/week-numbers-industry-demand-react-developers/. Accessed 28 Apr 2018
5. Toptal: Angular and React Native skills are in high demand - SD Times. https://sdtimes.com/webdev/toptal-angular-2-react-native-skills-high-demand/. Accessed 28 Apr 2018
6. The Cost of Native Mobile App Development is Too Damn High! https://hackernoon.com/the-cost-of-native-mobile-app-development-is-too-damn-high-4d258025033a. Accessed 28 Apr 2018
7. What are microservices? Lightweight software development explained. https://www.infoworld.com/article/3237697/application-development/what-are-microservices-lightweight-software-development-explained.html. Accessed 28 Apr 2018
8. Evodeck Software - Agile Web Development. https://www.evodeck.com/. Accessed 28 Apr 2018
9. Schwaber K (2002) Scrum and Agile 101. In: Lecture Notes in Computer Science. pp 266–267
10. Angular Docs. https://angular.io/. Accessed 5 Jul 2018
11. (2016) Microservice Architecture: Aligning Principles, Practices, and Culture. “O’Reilly Media, Inc.”
12. Nancy. https://github.com/NancyFx/Nancy. Accessed 10 Jun 2018
13. Internet Information Services - Wikipedia. https://en.wikipedia.org/wiki/Internet_Information_Services. Accessed 18 Apr 2018
14. Microservices - Definition, Principles and Benefits - HowToDoInJava. https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/. Accessed 18 Apr 2018
15. URIs, URLs, and URNs: Clarifications and Recommendations 1.0. https://www.w3.org/TR/uri-clarification/. Accessed 18 Apr 2018
16. Internet Information Services - Wikipedia. https://en.wikipedia.org/wiki/Internet_Information_Services. Accessed 18 Apr 2018
17. Node.js. https://nodejs.org/en/. Accessed 18 Apr 2018
18. nginx news. https://nginx.org. Accessed 18 Apr 2018
19. Microservices 101. http://bits.citrusbyte.com/microservices/. Accessed 29 Apr 2018

20. The Secret behind the Single Responsibility Principle. https://hackernoon.com/the-secret-behind-the-single-responsibility-principle-e2f3692bae25. Accessed 19 Apr 2018

21. Karma 2018 | Karma. https://yourkarma.com. Accessed 24 Jun 2018

22. How we build microservices at Karma – Spread the WiFi — The Official Karma Blog. https://blog.karmawifi.com/how-we-build-microservices-at-karma-71497a89bfb4. Accessed 24 Jun 2018

23. Introduction to Microservices | NGINX. https://www.nginx.com/blog/introduction-to-microservices/. Accessed 29 Apr 2018

24. A Conversation with Werner Vogels - ACM Queue. https://queue.acm.org/detail.cfm?id=1142065. Accessed 24 Jun 2018

25. Continuous delivery - Wikipedia. https://en.wikipedia.org/wiki/Continuous_delivery. Accessed 25 Apr 2018

26. Automation Testing Resources & Best Practices - Test Automation Made Easy: Tools, Tips & Training Awesomeness. https://www.joecolantonio.com/2017/03/02/automation-testing/. Accessed 25 Apr 2018

27. DevOps - Wikipedia. https://en.wikipedia.org/wiki/DevOps . Accessed 25 Apr 2018

28. A primer on distributed computing. http://billpg.com/bacchae-co-uk/docs/dist.html. Accessed 25 Apr 2018

29. Adaptive user interfaces for health care applications. https://www.ibm.com/developerworks/web/library/wa-uihealth. Accessed 29 Apr 2018

30. Rodrigues, J.M.F., Pereira, J.A.R., Sardo, J.D.P., Freitas, M.A.G., Cardoso, P.J.S., Gomes, M., Bica, P. (2017) Adaptive Card Design UI Implementation for an Augmented Reality Museum Application. In: Antona M, Stephanidis C (eds) Universal Access in Human-Computer Interaction. Springer International Publishing, Cham, pp 1–11

31. Veiga, R.J.M., Bajireanu, R., Pereira, J.A.R., Sardo, J.D.P., Cardoso, P. J.S., Rodrigues, J.M.F (28 October, 2017) Indoor Environment and Human Shape Detection for Augmented Reality: an initial study. In: 23rd edition of the Portuguese Conference on Pattern Recognition. p 21

32. Windows 10 Tip: How to set up Windows Hello on your PC - Windows Experience Blog. https://blogs.windows.com/windowsexperience/2016/10/31/windows-10-tip-how-to-set-up-windows-hello-on-your-pc/. Accessed 15 Mar 2018

33. Smart Lock. https://get.google.com/smartlock. Accessed 15 Mar 2018

34. Software development kit - Wikipedia. https://en.wikipedia.org/wiki/Software_development_kit . Accessed 15 Apr 2018

35. Detect and Recognize Faces and Facial Features with Luxand FaceSDK. https://www.luxand.com/facesdk. Accessed 15 Mar 2018

36. Windows 10 - Wikipedia. https://en.wikipedia.org/wiki/Windows_10 . Accessed 15 Mar 2018

37. Intel - Wikipedia. https://en.wikipedia.org/wiki/Intel . Accessed 15 Mar 2018

38. Intel® RealSense Technology | Intel® Software. https://software.intel.com/en-us/realsense . Accessed

15 Mar 2018

39. Microsoft Store (digital) - Wikipedia. https://en.wikipedia.org/wiki/Microsoft_Store_(digital). Accessed 15 Mar 2018

40. Set your device to automatically unlock. https://support.google.com/nexus/answer/6093922. Accessed 29 Apr 2018

41. Luxand in Academic Research. https://www.luxand.com/press/papers.php. Accessed 5 Mar 2018

42. Cognitive Services | Microsoft Azure. https://azure.microsoft.com/en-us/services/cognitive-services/. Accessed 29 Apr 2018

43. Face API - Facial Recognition Software | Microsoft Azure. https://azure.microsoft.com/en-us/services/cognitive-services/face/. Accessed 27 Dec 2017

44. Mobile OS market share 2017 | Statista. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. Accessed 16 Mar 2018

45. Android. https://www.android.com/. Accessed 16 Mar 2018

46. iOS 11. https://www.apple.com/ios/ios-11/. Accessed 16 Mar 2018

47. React Native · A framework for building native apps using React. https://facebook.github.io/react-native/index.html. Accessed 16 Mar 2018

48. JavaScript. In: JavaScript.com. https://www.javascript.com. Accessed 16 Mar 2018

49. React - A JavaScript library for building user interfaces. https://reactjs.org/index.html. Accessed 16 Mar 2018

50. What's a Universal Windows Platform (UWP) app? https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide. Accessed 16 Mar 2018

51. Identify faces in images with the Face API - Microsoft Cognitive Services. https://docs.microsoft.com/en-us/azure/cognitive-services/face/face-api-how-to-topics/howtoidentifyfacesinimage. Accessed 26 Dec 2017

52. Hypertext Transfer Protocol - Wikipedia. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol . Accessed 21 Mar 2018

53. Microsoft Cognitive Services. https://westus.dev.cognitive.microsoft.com/docs/services/563879b61984550e40cbbe8d/operations/563879b61984550f30395244. Accessed 29 Apr 2018

54. JSON. https://json.org. Accessed 21 Mar 2018

55. Kilobytes Megabytes Gigabytes Terabytes. https://web.stanford.edu/class/cs101/bits-gigabytes.html. Accessed 22 Mar 2018

56. What Is ERP | Oracle. https://www.oracle.com/applications/erp/what-is-erp.html. Accessed 27 Apr 2018

57. What is Test Driven Development (TDD)? In: Agile Alliance. https://www.agilealliance.org/glossary/tdd/. Accessed 11 May 2018

58. Home : The Official Microsoft IIS Site. https://www.iis.net/. Accessed 10 Jun 2018

59. What Is Windows Communication Foundation. https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf. Accessed 10 Jun 2018

60. OWIN — Open Web Interface for .NET. http://owin.org/. Accessed 10 Jun 2018

61. Domain-specific language - Wikipedia. https://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=827901987. Accessed 10 Jun 2018

62. What is Dependency Injection C#? How It Works, Types of Dependency Injections in C#, and More. In: Stackify. https://stackify.com/dependency-injection-c-sharp/. Accessed 10 Jun 2018

63. Interface based programming in C# - CodeProject. https://www.codeproject.com/articles/702293/interface-based-programming-in-csharp. Accessed 10 Jun 2018

64. grumpydev/TinyIoC: An easy to use, hassle free, Inversion of Control Container for small projects, libraries and beginners alike. https://github.com/grumpydev/TinyIoC. Accessed 10 Jun 2018

65. Inversion of Control – An Introduction with Examples in .NET. http://joelabrahamsson.com/inversion-of-control-an-introduction-with-examples-in-net/. Accessed 10 Jun 2018

66. Communication endpoint - Wikipedia. https://en.wikipedia.org/wiki/Communication_endpoint. Accessed 5 Jul 2018

67. namespace (C# Reference) | Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/namespace. Accessed 10 Jun 2018

68. HTTP Methods for RESTful Services. http://www.restapitutorial.com/lessons/httpmethods.html. Accessed 10 Jun 2018

69. Lambda Expressions (C# Programming Guide). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions. Accessed 10 Jun 2018

70. UTC - Coordinated Universal Time. https://www.timeanddate.com/time/aboututc.html. Accessed 10 Jun 2018

71. Extensible Markup Language (XML). https://www.w3.org/XML/. Accessed 10 Jun 2018

72. Pattern. https://github.com/NancyFx/Nancy/wiki/Defining-routes#pattern. Accessed 10 Jun 2018

73. Route Segment Constraints. https://github.com/NancyFx/Nancy/wiki/Defining-routes#route-segment-constraints. Accessed 10 Jun 2018

74. Nancy.Validation.FluentValidation 1.4.1. https://www.nuget.org/packages/Nancy.Validation.FluentValidation/. Accessed 26 Jun 2018

75. Bootstrapper. https://github.com/NancyFx/Nancy/wiki/Bootstrapper. Accessed 10 Jun 2018

76. Content Negotiation. https://github.com/NancyFx/Nancy/wiki/Content-Negotiation. Accessed 10 Jun 2018

77. Controlling the negotiation. https://github.com/NancyFx/Nancy/wiki/Content-

Negotiation#controlling-the-negotiation. Accessed 10 Jun 2018

78. HTTP/1.1: Header Field Definitions. https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html. Accessed 10 Jun 2018

79. 200 OK — httpstatuses.com. https://httpstatuses.com/200. Accessed 10 Jun 2018

80. Unit Test Basics. https://msdn.microsoft.com/en-us/library/hh694602.aspx. Accessed 23 May 2018

81. Mock object - Wikipedia. https://en.wikipedia.org/wiki/Mock_object. Accessed 24 May 2018

82. moq moq/moq4: Repo for managing Moq 4.x. In: GitHub. https://github.com/moq/moq4. Accessed 10 Jun 2018

83. dennisdoomen/fluentassertions: Fluent Assertions is a set of .NET extension methods that allow you to more naturally specify the expected outcome of a TDD or BDD-style test. https://github.com/dennisdoomen/fluentassertions. Accessed 10 Jun 2018

84. Entity Framework. https://docs.microsoft.com/en-us/aspnet/entity-framework. Accessed 10 Jun 2018

85. What is Code-First? http://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx. Accessed 10 Jun 2018

86. DbContext Class (System.Data.Entity). https://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext(v=vs.113).aspx. Accessed 10 Jun 2018

87. Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10). https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application. Accessed 10 Jun 2018

88. How to implement the Repository design pattern in C#. https://www.infoworld.com/article/3107186/application-development/how-to-implement-the-repository-design-pattern-in-c.html. Accessed 10 Jun 2018

89. P of EAA: Data Transfer Object. https://martinfowler.com/eaaCatalog/dataTransferObject.html. Accessed 10 Jun 2018

90. nandotech/NancyAzureFileUpload. https://github.com/nandotech/NancyAzureFileUpload. Accessed 9 Jul 2018

91. Microsoft Azure Cloud Computing Platform & Services. https://azure.microsoft.com/en-us/. Accessed 9 Jul 2018

92. RFC1341(MIME) : 7 The Multipart content type. https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html. Accessed 9 Jul 2018

93. Asynchronous programming. https://docs.microsoft.com/en-us/dotnet/csharp/async. Accessed 9 Jul 2018

94. 200 OK — httpstatuses.com. https://httpstatuses.com/200. Accessed 11 Jul 2018

95. NancyAzureFileUpload/UploadModule.cs at master · nandotech/NancyAzureFileUpload. https://github.com/nandotech/NancyAzureFileUpload/blob/master/src/NancyAzureFileUpload/Modules/UploadModule.cs. Accessed 9 Jul 2018

96. NancyAzureFileUpload/DispatchFileService.cs at master · nandotech/NancyAzureFileUpload. https://github.com/nandotech/NancyAzureFileUpload/blob/master/src/NancyAzureFileUpload/Services/DispatchFileService.cs. Accessed 9 Jul 2018