

---

Extensión de navegador para la eliminación de metadatos  
A browser plug-in for metadata removal

---



Trabajo de Fin de Grado  
Curso 2019–2020

**Autor**

Diego Ambite Varona  
Guillermo García Mansilla  
Rosa Olivia Zumaeta Sánchez

**Director**

Enrique Martín Martín

Grado en Ingeniería Informática  
Facultad de Informática  
Universidad Complutense de Madrid



Extensión de navegador para la  
eliminación de metadatos  
A browser plug-in for metadata removal

**Trabajo de Fin de Grado en Ingeniería Informática**

**Autor**

Diego Ambite Varona  
Guillermo García Mansilla  
Rosa Olivia Zumaeta Sánchez

**Director**

Enrique Martín Martín

**Convocatoria: Junio 2020**

**Grado en Ingeniería Informática  
Facultad de Informática  
Universidad Complutense de Madrid**

**26 de JUNIO de 2020**



# Resumen

Los metadatos, datos sobre los archivos que generamos digitalmente, dicen más sobre nosotros de lo que creemos, pudiendo llegar a ocasionar un grave riesgo para nuestra privacidad e incluso para la seguridad de nuestros equipos. Nuestro proyecto ha consistido en la elaboración de una aplicación web en forma de extensión de navegador fácilmente manejable por el usuario y que requiere muy poca interacción con ella, ofreciendo la posibilidad de limpiar los metadatos presentes en los archivos, concretamente en imágenes, que se van a subir a la plataforma Google Drive.

Para la consecución de esta meta, hemos organizado el proyecto siguiendo una arquitectura basada en microservicios ofreciendo un fácil acoplamiento a nuevos formatos cuyos metadatos se quieran tratar. Para el desarrollo de estos microservicios y de la extensión, hemos usado tecnologías ampliamente usadas hoy en día como pueden ser JavaScript, Spring Boot.

*Palabras clave:* Metadatos, Extensión de navegador, Microservicios, Privacidad, Google Drive, peticiones HTTP, webRequest, API REST

# Abstract

The metadata, which is data about the documents we generate digitally, tell more about ourselves than what we could think, potentially being a threat to our privacy and even to the security of our systems. Our project has been the development of a web application which works as a browser extension which is easy to handle by the user and which requires very little interaction with it, offering the possibility to eliminate the metadata present in the documents, specifically in images, that are going to be uploaded to Google Drive.

For the achievement of this goal, we decided to organize the project following an architecture based on microservices, offering an easy way to introduce new formats to the system for the processing of the metadata. For the development of these Microservices and the extension, we used technologies widely used today, such as JavaScript, Spring Boot.

*keywords:* Metadatos, Browser Extension, Microservices, Privacy, Google Drive, HTTP Requests, webRequest, API REST

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Objective . . . . .	2
1.2. The work plan . . . . .	2
<b>2. Preliminaries</b>	<b>5</b>
2.1. What is Metadata . . . . .	5
2.2. Why is Metadata so Important . . . . .	6
2.3. What are Microservices? . . . . .	8
2.4. Benefits of Microservices . . . . .	9
2.5. Browser extensions and JavaScript . . . . .	9
2.6. Spring Boot . . . . .	11
2.7. Tools for metadata management . . . . .	11
<b>3. System Organization</b>	<b>13</b>
3.1. Extension . . . . .	14
3.2. Orchestrator . . . . .	15
3.3. Cleaning microservices . . . . .	15
<b>4. Extension</b>	<b>17</b>
4.1. Collecting requests . . . . .	17
4.2. Processing the request . . . . .	19
4.3. Data exchange to microservices . . . . .	20
4.3.1. Orchestrator . . . . .	20
4.3.2. Cleaning microservices . . . . .	21
4.4. File cleaned . . . . .	22
4.5. Limitations found during development . . . . .	22
<b>5. Orchestrator</b>	<b>25</b>
5.1. Performance . . . . .	25
5.2. Classes and Interfaces used to create the Orchestrator . . . . .	26
5.2.1. MSOrchestratorMain . . . . .	26
5.2.2. Format . . . . .	26
5.2.3. FormatsDAO . . . . .	27
5.2.4. FormatsREST . . . . .	27

5.2.5. SwaggerConfig . . . . .	29
5.3. Consistency . . . . .	29
5.4. Libraries employed . . . . .	30
5.5. Difficulties and considerations . . . . .	30
<b>6. Metadata Management</b>	<b>31</b>
6.1. Implementation details . . . . .	31
6.1.1. Viewing metadata from a proposed file . . . . .	31
6.1.2. Remove metadata from a provided file . . . . .	33
6.2. File formats . . . . .	34
6.3. Appendix . . . . .	34
<b>7. Contribution</b>	<b>37</b>
7.1. Diego's contributions . . . . .	37
7.2. Guillermo's contributions . . . . .	38
7.3. Rosa's contributions . . . . .	40
7.3.1. Stage 1: Realization of a test browser extension . . . . .	40
7.3.2. Stage 2: Browser extension study . . . . .	41
7.3.3. Stage 3: Feasibility of the proposed browser extension . . . . .	41
7.3.4. Stage 4: Memory report . . . . .	42
<b>8. Conclusions</b>	<b>43</b>
8.1. Future work . . . . .	43
8.1.1. Extend the functionality to more formats . . . . .	44
8.1.2. Extension . . . . .	44
8.1.3. Move to from local to online . . . . .	45
8.1.4. Extend this service to other platforms . . . . .	45
<b>Bibliography</b>	<b>47</b>





## Introduction

Metadata (11) is data about data. In other words, it's information that's used to describe the data that's contained in something like a web page, document, or file. Another way to think of metadata is as a short explanation or summary of what the data is.

A simple example of metadata for a document might include a collection of information like the author, file size, the date the document was created, and keywords to describe the document. Metadata for a music file might include the artist's name, the album, and the year it was released.

For computer files, metadata can be stored within the file itself or elsewhere, like is the case with some EPUB book files that keep metadata in an associated ANNOT file.

Metadata represents behind-the-scenes information that's used everywhere, by every industry, in multiple ways. It's ubiquitous in information systems, social media, websites, software, music services, and online retailing. Metadata can be created manually to pick and choose what's included, but it can also be generated automatically based on the data.

The following are the existing types of metadata:

- **Descriptive** metadata properties include title, subject, genre, author, and creation date, for example.
- **Rights** metadata might include copyright status, rights holder, or license terms.
- **Technical** metadata properties include file types, size, creation date and time, and type of compression. Technical metadata is often used for digital object management and interoperability.
- **Preservation** metadata is used in navigation. Example preservation metadata properties include an item's place in a hierarchy or sequence.
- **Markup languages** include metadata used for navigation and interoperability. Properties might include heading, name, date, list, and paragraph.

## 1.1. Objective

Our goal is to offer an online metadata removal service using Google Chrome and Firefox extension.

On a functional level, this tool offers the user the possibility to remove metadata from files that he wants to upload to the Internet.

The general objective consists of the following:

- To have a microservices architecture, offering scalability and flexibility to the project
- Create microservices for cleaning metadata from different file formats, so that the functionality of the tool is more extensive.
- Create an orchestrator, in order to offer the extension information related to the microservices being created.
- Create a browser extension that connects to the cleaning microservices and the orchestrator

## 1.2. The work plan

Initially, we made a study of the Chrome extension to see the viability of making requests to external services, different browsers for which we could develop the extension as well, etc.

After this study, the following tasks were proposed:

- **Extension**

Regarding the extension, a study and made a Proof Of Concept (POC) about the possibility to make requests from an extension, block the request and send it to an external service with a file attached to it.

- **microservice in charge of dealing with the metadata**

In this area, since metadata can be present in different types of files, such as images, PDF, office files... we had to decide with which we would deal first. By team consensus, it was proposed to start the project with a microservice in charge of processing image files.

For this purpose, we decided the development of a microservice, which will allow the consumption of two services:

- Consult the metadata of a file
- Remove metadata from a file

Although the project started with a microservice that manages the image files, this component can be scalable and replicated, and it would be possible to create as many microservices as formats you want to treat.

### ■ **Microservice Orchestrator**

This microservice offers to the extension the possibility to know which are the microservices which deals with each format. Making in this way easier to the extension to keep track of what are the URL where each service is hosted. Also, this microservice will be the one in charge of registering the new formats of files that we can be added.

The orchestrator will consult the address of the microservices in charge of both consulting and removing the metadata.

After clarifying the existing tasks, the division of tasks was as follows:

- Guillermo was in charge of the extension.
- Rosa was in charge of the microservice that manages the image metadata.
- Diego was in charge of the orchestrator microservice and the coexistence of microservices in the server.

The tasks are carried out in several phases, which are listed below:

#### **Phase 1:** Research and task sharing, *October 2019 - December 2019*

The research and feasibility research of the initial project proposal is carried out. This phase involves the following tasks:

- Study and viability of the project.
- Investigation different possible technologies.
- Distribution of tasks among team members.

#### **Phase 2:** POC Development, *December 2019 - February 2020*

The development of a proof of concept for all components is carried out. This phase involves the following tasks:

- Development of a POC of the different components.
- Study of limitations and changes in requirements.
- Integration of the different components.
- Unit tests.

#### **Phase 3:** Application development, *February 2020 - April 2020*

This phase includes the development of the extension and the two microservices. The following tasks are included:

- Application development with final requirements.

- Final integration of the components.
- Unit tests.

**Phase 4:** Testing, *April 2020 - May 2020*

During this phase, we started to integrate the different components, including improvements and corrections that were detected.

**Phase 5:** Memory, *May 2020 - June 2020*

The last phase of the project includes the distribution of the different sections of the memory, as well as one last revision of the code.

# Chapter 2

## Preliminaries

This chapter introduces and explains a series of different concepts that are important to know well before continue reading this memory. They go from concepts which will be continuously mentioned and are the very base of the project itself as the notion of metadata or browser extension to other more structurally related as Microservice or linked technologies as Spring Boot. At the end of this chapter in Section 2.7, you will find how is the metadata handled currently by different programs and websites which have a similar goal as our project.

### 2.1. What is Metadata

Metadata is data that tell us information about a certain digital resource. Every time we create a text file or take a picture with our mobile phone, send a voice message through our favourite chat application or we install a program on our computer, we are not just generating the resource itself; in addition to it, a little piece of information is as well built and attached to it. This information is what we call metadata and it will describe certain features about the resource such as its size, which title does it have, who is the user who has created it, and more sensible one such as the kind of software used, the coordinates and the date in which it was generated. Depending on the resource more or less metadata will be set up, also this metadata will be modified as the resource is used, enabling it to evolve among the life of the resource itself.

We could understand metadata as a label, almost any product you could find has some sort of label, from the clothes you wear to the food you buy on the market or even this memory has some kind of label telling us information about the memory itself in the first page. It can vary in shape or form, could be physical like a piece of paper, a barcode hide somewhere in the product or just some words printed on it. Metadata is the digital equivalent of this label.

In the case of metadata is much easier to find, you just have to go to your computer and inspect the properties on any document you wish. Easily you will find the main metadata of resource, such as its title, size, the type of resource or wherein the disk it is stored. It

is important to mention that most of the times, it requires a deeper search to gather the whole metadata and what we find with this fast search is just the tip of the iceberg.

## 2.2. Why is Metadata so Important

From a naive point of view, metadata is a great tool that computer systems have to help the user dealing with resource discovery and organization. Is in the metadata where the information we typically use to identify documents is stored, such as the title or the format. Metadata also stores a unique identifier that tags each resource, which is not that useful from a user point of view but is what helps the system to avoid duplicity of documents. As searches are most of the time made using words, for multimedia resources as music, photos, or video its metadata is the only text present, unlike text-based documents.

Metadata is particularly harmful when shared, as its own name says metadata, after all, is data that we have generated, so is our information. The distribution of personal data usually leads to privacy matters. This joined to the fact that metadata is something many people do not know about and even people who do rarely take care of checking it out, makes of the metadata a great leak of information and a potential threat to both our privacy and security.

If you use a photo storing service like Google Photos, Amazon Photos or even your phone gallery maybe you have detected that your photos are being classified not just by the day they were taken but also by the place they were taken. This is because photos taken by most of modern digital cameras and smartphones, in addition to the metadata that we are more familiarised with like the date and time when the photo was taken or the size of it, it is also stored the GPS coordinates and the camera model which was used. In other words, among all the pictures we have stock for years and years, it hides a map of all the places were when they were taken.

Not many people are aware of the fact that embedded with a picture from your home or your kid's school, there may be some GPS coordinates. They are letting anyone who has access to that picture and knows how to use the resources like the ones mentioned in Section 2.7, know where your home or your kid's school are located. Figure 2.1 is shown the GPS metadata from a mobile phone photo.

Another ground where metadata is telling about us more than what we could think of is when we exchange emails or text messages with other people using a digital platform. Most of these services available on the market like Whatsapp or Gmail claim to be secure and private by performing end-to-end encryption to all our texts avoiding them to be read even by anyone, including the service itself, but the addressee.

Nonetheless, if the whole message is ciphered (20) and not even the courier company can read any of it how is it supposed to deliver it? Yes by this point you have probably guessed it right, by the means of metadata. This is because is the content of the message itself what has been protected but not its metadata, which will be used to distribute the message, and once it has been delivered these metadata may not be just deleted but instead stored.

Thanks to this metadata the companies, even not knowing the messages itself, will get

---

```

"Details":{
  "Categories": [
    "GPS"
  ],
  "Name": "VersionOfGps"           , "Value": "2 2 0 0",
  "Name": "GpsLatitudeReference"  , "Value": "N",
  "Name": "Latitude"              , "Value": "40°25'1.54",
  "Name": "GpsLongitudeReference" , "Value": "W",
  "Name": "Longitude"            , "Value": "3°39'45.71",
  "Name": "ReferenceAltitude"     , "Value": "Sea level",
  "Name": "Altitude"             , "Value": "36675/50 metres",
  "Name": "GpsTimestamp"         , "Value": "18:37:17 UTC",
  "Name": "DegreeOfPrecisionGps" , "Value": "17399/1000",
  "Name": "GpsDatestamp"        , "Value": "2020:06:13"
}

```

---

Figure 2.1: Content of the metadata of a photo taken from a phone on the street, we have selected here all the fields with some information about the GPS coordinates, from all the original metadata.

a lot of information. They will know who are you texting with, the timestamp of those messages, if it is a one-way or a reciprocal communication and for how much time the conversation took place. Realizing the popularity of these services, the amount of personal information gathered is huge.

Finally, we will like to spotlight another thread which can be even more direct than the ones mentioned above. The insight of your metadata can be a great service for someone planning on a phishing (21) attack on you.

A phishing attack could be summarized as a way to get some of your personal information by pretending to be a trustworthy entity which you will not mistrust. Therefore you will obey their demands for your personal information like changing a password or directly telling it to them. But phishing by itself is not such a dangerous tool, after all, pretending to be your bank or your phone company just by making the mail look official is just useful to a certain point.

But now imagine that this attacker has been able to gather some of your files, knowing the number of files we upload every day to the internet is not unthinkable. Maybe they did not contain any sensitive data, but as we have said before the metadata on it could contain personal information like your name, the name of your company, mail addresses of the people you have been in contact with, your professional department, the kind of software you use, where in your computer the file was stored in, so if the folders of your computer have descriptive names they will become even more clues about you. With all this information in his power, the phishing attacker can look way convincing, otherwise, how would they know that much about you?.



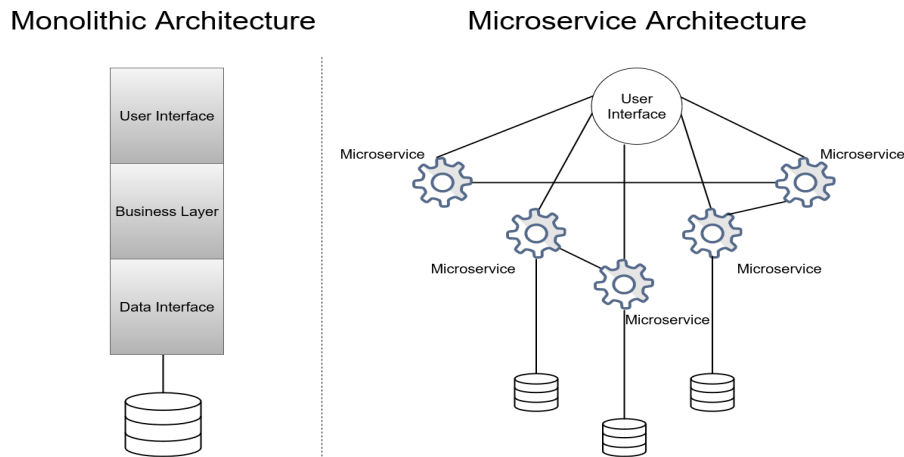


Figure 2.2: Monolithic architecture in opposition to the Microservices architecture.

### 2.3. What are Microservices?

Microservices (52) is an architectural and organizational way of building a software project by disengaging it in smaller and great independent services distributed in different machines which communicate among themselves through a well-defined API.

A Service in the Software Architecture vocabulary is each of the functionalities that compose an application and has a unique task. So imagine you are on a website where you can buy any kind of good. When you add any good to the cart you are using one specific service, if you search using the different filters that the site offers you are using a second service, and when finally you proceed to pay for your order, you will make use of a third service. So on one website and in a few seconds and without noticing you have employed three different services, all of them working together in the same frame.

Traditionally when Software companies had to develop a project followed what we say is a Monolithic architecture. In this approach, the entire system functionality is based on a single application. All the features will be executed on the same machine, communicating among them to offer different services to the user. As a consequence of this, all had to be coded on the same programming language, as there was just one machine there was just one database and one server. This model can be seen on the left side of Figure 2.2.

As the computing power of the systems evolved also did the Software applications, which became bigger, more complex and demanded a higher Maintainability. This was hard to accomplish in the frame of a monolithic architecture for obvious reasons: looking for a certain function in a project which has thousands of lines of code which were not written by you does not look optimal.

From this need to decentralise the projects into smaller parts, the Microservice Architecture arise. Here every feature which forms the project will be their application, running in their machine and having if needed their database and server. As said in the definition at the beginning of the section, each microservice will communicate with each other using an API.

## 2.4. Benefits of Microservices

The main advantages of adopting the microservice architecture are the following ones:  
(51)

- **Modularity:** As each Microservice is widely autonomous from the others, it can be individually developed and deployed. This helps to split the tasks among the team, as boundaries of each service are well-defined. As well in the case of any error, it will be just in the same Microservice where it was caused, enabling the rest of the system to work as usual.
- **Scalability:** With the split of the project, we can focus our effort on the Microservices which demand a more intensive workload. Once a microservice is already completed it can be left aside being fully functional and not disturbing the main effort on the others.
- **Versatility:** Microservices bring us the possibility of using different technologies and programming languages for each microservice. They will communicate with the other components in the system by the API, no matter the chosen technology. This allows the developers to select the technology used in each module to the one which adjusts the most to the required functionality.
- **Maintenability:** When maintaining any functionality, as they are well distinguished we could focus on those involved directly in the changes, leaving those not involved in the maintenance aside. This was harder to get in the monolithic architecture, were processes were tightly connected, and making a change could cause a domino effect.
- **Agility:** Many common functionalities such as the ones for authentication, the search of an item or its traceability have already been well developed as Microservices by third parties, so there is no need of starting from scratch as you can make use of them. Moreover, the greater facility to split the work between already established microservices makes it faster to start developing.

## 2.5. Browser extensions and JavaScript

Browser extensions are small applications that run on your Internet browser and offer additional functionality to the ones the browser has by default. They are typically available on their websites or in an extension store, as many are not free. They require the user to download and install them but as they are typically quite small it is done easily.

We could classify the most popular browser extensions (19) by the users by the purpose they have:

- **Advertisement blocking:** Here we could place the two most popular ones: *Adblock* and *Adblock plus*. Their task is to vanish the publicity, the first one is more focused on website popups while the second in the advertisement on the videos.

- **Security:** Here we can mention *Avast Online Security* and *Avira Browser Safety*. They try to avoid the download of malware and the tracking of your activities respectively.
- **Easiness:** Unlike the two last categories, this one has a broader variety of purposes but can be summarized saying that they try to save you time.
  - *Avast Safe Price:* Compare the prices of the same item across many sites to offer you the best deal.
  - *Quick start:* In a similar way that browser bookmarks, allow you to get to your favourite sites by just one click.
  - *Adobe Acrobat - Create PDF:* Downloads a whole web converting it to a PDF file.
  - *Hangouts:* Allows you to go to the famous Google chatting app without the need of entering and logging every time in their main site.

It is very encouraging to our project to have a similar character to some of the most popular extensions on the market, as are the ones previously grouped as security extensions.

Most of the extensions are coded using JavaScript (50) as is one of the most popular and widely used programming languages (18). JavaScript is an interpreted programming language, this means that does not require to be compiled as is designed to run on the browsers. It is mainly employed to execute actions on the client-side but it is also possible to use it as well in the server-side. Although JavaScript is an object-oriented language, it is based on prototypes, which means that it does not use classes, also it is an untyped programming language.

JavaScript indicates the action it wants to perform to a certain resource by HTTP petitions, JavaScript can do this in two different ways, using `fetch`(41) or `XMLHttpRequest`(45):

- `fetch` is a new native JavaScript API, supported by most of the browsers which provide a friendly interface. The `fetch` method ease making asynchronous requests and handle responses better than `XMLHttpRequest`.

The main difference between `fetch` and `XMLHttpRequest` is that the `fetch` API uses `Promises`(42), avoiding callbacks. When we call `fetch` a `Promise` is returned, whether it was successful or not. A `Promise` represents a value that may be available now, in the future, or never.

Once `fetch` has performed the request, a set of methods will response as follows:

- **`.then()`** (38) In case the request was successful, the `.then()` method will receive a `Response` object.
- **`.catch()`** If the request failed the `.catch()` method will receive an error object.
- **`.json()`** As the communication among Microservices is carried on by the use of JSON files, the response must be transformed into a JSON object with the `.json()` method. As this method is applied to a `Response` object, the return will be a new `Promise`, so it needs to chain on another `.then()`.

- `XMLHttpRequest` is a JavaScript object that provides an easy way to get information from a URL without having to reload the entire page. A web page can refresh only part of the page without interrupting whatever the user is doing. `XMLHttpRequest` can make both synchronous and asynchronous requests but is not as optimal as `fetch` (48) because, as said before `XMLHttpRequest` do not use Promises. But unfortunately `XMLHttpRequest` is the only way to make synchronous requests (36).

## 2.6. Spring Boot

Spring Boot (54) is an open-source framework, based on Java, to develop stand-alone Java web applications that can rapidly and easily be deployed.

Spring Boot evolves from the Spring (22) framework, but instead of all the configurations needed to finally get a functional application, Spring Boot runs with minimal or zero configurations. It owns an embedded HTTP server, and to avoid the major number of configurations, it prioritizes convention over configuration. As well Spring Boot does not require an XML configuration, replacing it by the annotations which structure the project using component-scanning.

All the above characteristics make Spring Boot an ideal tool for developing Java Microservices as it perfectly matches the benefits detailed in Section 2.4.

## 2.7. Tools for metadata management

From what we have researched there is not a service that offers what we have targeted in this project. The mix of both an extension that is easily handled and not requires from working with the files locally and the capability of cleaning the metadata from the browser itself is something that at least we could not find.

Considering extensions, there are some in the market which allows you to see the metadata of the website you are on, but there is no one which focuses on the cleaning of the metadata from uploaded files before sending them. Regarding metadata cleaning, there are services which labour as desktop programs requiring installation and working locally. There are also some online applications which require you to upload the files explicitly to their site, so you have to get them instead of the other way that is what we pretend. We have discarded the services which require to be bought to try them as *BatchPurifier* (30) and centered our attention on the ones we could use for free, as our extension is. Some of the ones we consider more interesting are:

- There is a desktop program called *PDF Metadata Editor* (29) which helps you to handle the metadata contained in a certain PDF. It works as follows: first you drag and drop the PDF you want to work within the program window. After that, all the metadata contained in the document is displayed in the different panels. The strong point of this application is how user-friendly it is and how little time it requires to modify the metadata. It also presents some interesting features like setting some

default values for future uses. The main differences with our application are the fact that this one just works locally and targets specifically PDF documents.

- Another program worth mentioning is *Exiv2* (28) which in a similar way as *PDF Metadata Editor* cleans files which are stored in the file system. This program instead focuses on image files, a total of 25 different image formats are supported (Among them the most common ones JPG, PNG, TIF and GIF). It is better than *PDF Metadata Editor* in terms of supporting a wider range of valid formats. Moreover, the use of this tool is by commands and is only available for Linux and MacOS making it less manageable to most of the users. We have in common with it the formats we can clean, but the fact that this tool just works locally once more makes it differ from our application.
- A tool that matches more with our project in the sense that is an online service is *Exif Viewer* (31). However, it is not available as a browser extension and it does not allow you to modify the metadata found, just to visualize it. Similarly to *Exiv2* it focuses on image formats and it has a complete platform where you can find metadata such as all the modifications that the image has had since it was created.
- A program we can not exclude when talking about dealing with metadata is *FOCA* (33) (Fingerprinting Organizations with Collected Archives). Its main purpose is to audit domains, and it is very used in the practice of penetration testing. FOCA searches in different search engines (Google, Bing, and Duck Duck Go) for information at the domain level downloading all the files related. Then, using the metadata attached to those files, it elaborates a map where you can see the different operating systems, applications, emails and the server addressing of the company you are investigating.

In addition to this powerful tool, FOCA also allows you to scan a set of metadata which are in your file system. It is better for this task than the programs enumerated above because offers you the possibility to scan many different formats (PNG, PDF, Open office...) and to do it in groups and not one by one. It extracts the metadata, treats it before presenting it, avoiding the repetitions and classifying the metadata by kind. FOCA is an open-source program which is easily downloaded and which complete code is on their Github repository (32).

- Eleven Paths, the company who owns FOCA, also came up with *Metashield* (23) a service-oriented program to protect the metadata of the companies. It is a paid service, but they have as well a free application which enables you to update just a file at a time and then it displays all the metadata on it. Then it allows you to clean it all. Out of all the free services we have checked this is probably the one which works the best, not just in terms of the number of file formats which accepts but also because how easy is it to use.

## System Organization

This chapter of the document is dedicated to the distribution of the different components that make up the application, where, as a metadata cleaner, it requires the processing of many different types of files. Each type of file requires a different microservice cleaner, forming a complex system that works together with the extension and the orchestrator microservice as shown in Figure 3.1.

Traditionally, applications were designed so that all the elements could be implemented in the same project where all processes are closely associated and run as a single service. But when the application requires improvements or added functionality all architecture must be improved. Adding or improving the features of a monolithic application becomes more complex as the code base grows. In contrast, if applications are designed with **microservices** (14), these problems are solved and development and responsiveness are promoted. With microservices, applications are divided into independent elements that work together to carry out the same tasks.

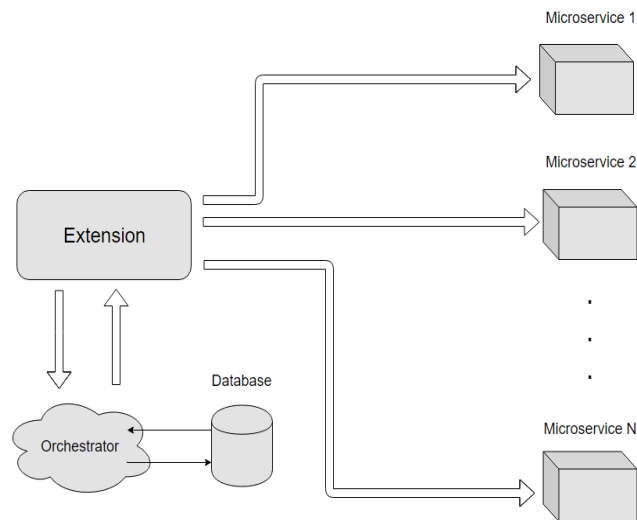


Figure 3.1: Graphical representation of the internal organization of the system

Each service is a separate codebase, which can be managed by a small development team or a single person. Services can be deployed independently without rebuilding and redeploying the entire application and they do not need to share the same technology stack, libraries, or frameworks. If developers add more code to a service over time and the service becomes complex, it can be broken down into smaller services.

To guarantee the correct operation of the microservices in the system, there must be an **orchestrator** component that is responsible for placing services in the right place ensuring that the correct microservices are being used in the right place.

### 3.1. Extension

The extension is a small piece of software that is attached to the browser and allows the user to remove all the metadata sent in a file while uploading it. It works as the main program directing each operation performed in the system. The extension is the component that makes most of the connections as shown in Figure 3.2, and those are the following:

- **Web browser:** The extension monitors all requests made by the browser until it finds a request with a file to be uploaded. This file is extracted from the request and processed it in the extension, where the file type is used to select the URL of the microservice that must be called to clean the file thanks to the orchestrator microservice.
- **Orchestrator:** The orchestrating microservice provides a mapping from file types to

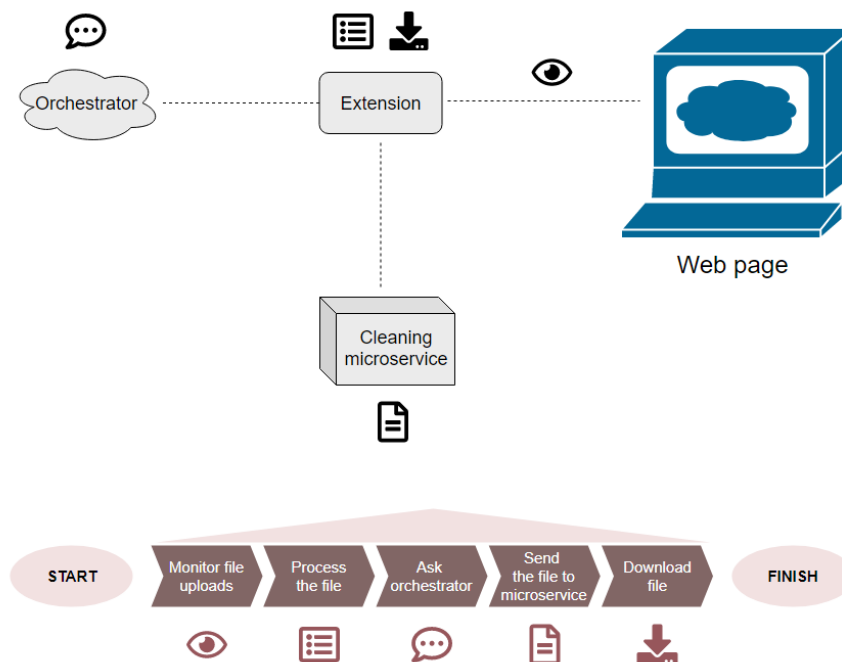


Figure 3.2: Extension tasks and its connections

microservice URLs that processes them. These connections are made asynchronously with `fetch` API (41), sending a GET request to the URL of the orchestrator.

- **Cleaning microservices:** The cleaning microservices receive the file sent by the extension and return a list with the metadata of the file to be removed. This process is done synchronously with the `XMLHttpRequest` API (45), which sends a POST request sending the file to the corresponding URL. It can also receive a file and delete every metadata found on it. These connections are made asynchronously with `fetch` API, sending a POST request to the URL given by the orchestrator.

The extension, as the central program of the system is the one in charge of carrying out data exchanges with other parts of the system. In addition to performing data exchanges, it also performs other tasks, such as processing requests to intercept outgoing files, or processing these files to break them down into valuable information for each microservice. Lastly, it is in charge of downloading the clean file once it has been received by the cleaning microservice into the user's computer. This process is explained in more detail in Chapter 4.

## 3.2. Orchestrator

The orchestrator provides the extension with information about the types of files that are supported by the system, that is, if they can be processed or not. Besides, it provides the extension with the URLs of the microservices for each file type, so that it organizes the sending of data to other services (this process is further detailed in Chapter 5). These are the connections made by the orchestrator:

- **Database:** The information that the orchestrator manages is stored in a database. The orchestrator accesses the database with **Spring Data JPA** technology (13), which is part of the large Spring Data family. This allows the orchestrator to query and add data to the database. This part is explained in detail in Chapter 5.
- **Extension:** As explained before, the extension sends a GET request to the orchestrator and receives a list with the names of the file types that are available in the system, in addition to the URLs where each of the associated microservices are located.

## 3.3. Cleaning microservices

The cleaning microservices remove the metadata from the files they receive. There is a microservice for each type of file that can be cleaned by the system. These microservices can be implemented in different ways, using different technologies, etc. New cleaning microservices can be added to the system by notifying the orchestrator that there is a new type of data that can be cleaned. The connection established by the microservices is always to the extension, which can be of two types:



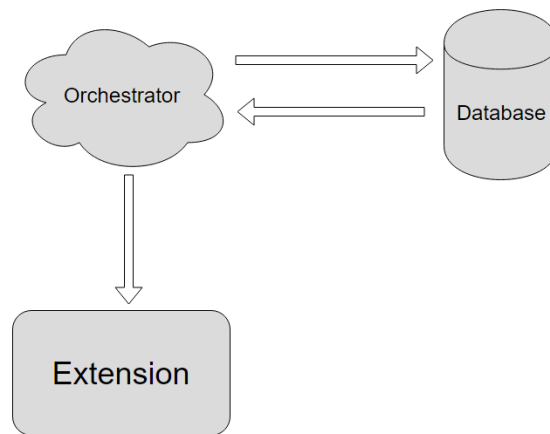


Figure 3.3: Orchestrator connections

- **Metadata query:** The extension sends a POST request with the file (explained in Section 3.1) and the microservice sends back a `string` with the information of all the metadata found in the file that can be removed.
- **Cleaning metadata:** The extension sends again the file (explained in Section 3.1) and the microservice removes metadata found in the metadata query.

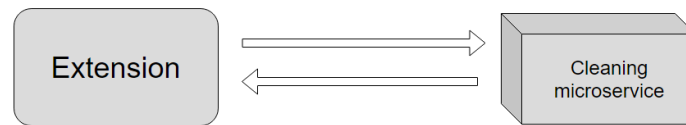


Figure 3.4: Cleaning microservices connections

# Chapter 4

## Extension

A **browser extension** is a small software program that customizes the browsing experience. The purpose of having this extension is to eliminate unnecessary information existing in the files, known as metadata. The extension itself has the main task of collecting outgoing requests (where the file uploaded is) and processing them so they can be sent to the microservices. Cleaning microservices wait for information to arrive and process the file, delete the metadata from it, and send back the file with no metadata left. This file is downloaded to the computer by the extension.

### 4.1. Collecting requests

The browser extension is responsible for collecting outgoing requests, to extract the file we want to process. This task is possible thanks to **webRequest** API (16). This API allows observing, analyzing traffic and intercepting, blocking, or modifying requests on the fly (before they reach the server).

To capture the requests, we are going to use **listeners**. These listeners are set with the `addListener()` function, that requires 3 parameters:

- **Callback:** Each `addListener()` call takes a mandatory callback function as the first parameter. This callback function receives a dictionary containing information about the current URL request. The information in this dictionary depends on the specific event type as well as the content of `opt_extraInfoSpec` and they can be: `requestId`, `url`, `method`, `frameId`, `type`, `requestHeaders`, `requestBody`, etc.
- **Filter:** They can be URLs for limiting the listeners to certain pages only, types for the request type, tab ID, or window ID.
- **`opt_extraInfoSpec`:** String array that has listener properties. We can use `blocking` if we want the callback function to be received synchronously, which means that the request will be blocked until this function ends. In this case, the callback can return a `webRequest.BlockingResponse` that determines the further life cycle of the

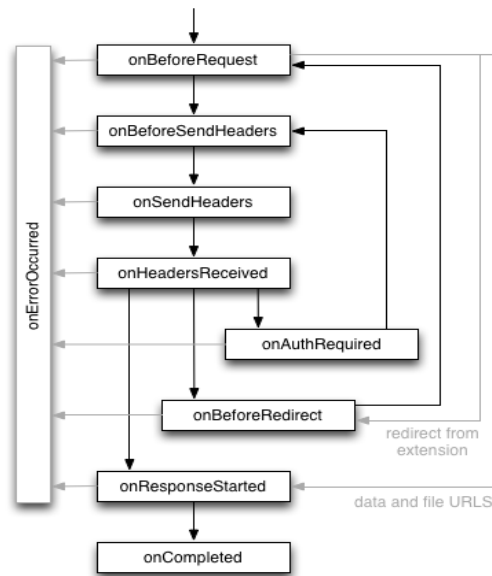


Figure 4.1: Illustration of an event life cycle for successful requests.  
 Source: '<https://developer.chrome.com/extensions/webRequest>'

request. Depending on the context, this response allows canceling or redirecting a request. For this extension this parameter is crucial since we want to freeze the request to the point that the user has made a decision, either to cancel the upload of files, or to clean them.

The `webRequest` API defines a set of events that follow the life cycle of a web request. You can use these events to observe and analyze traffic. Certain synchronous events will allow you to intercept, block, or modify a request.

As shown in Figure 4.1, **onBeforeRequest** are the events that are processed first and they are fired when a request is about to occur. This event is sent before any TCP connection is made and can be used to cancel or redirect requests and the body of the request can be accessed through this event. The request body stores the file as a binary array, which is saved in the extension in a global variable. This is due to how the listeners capture the events on that order (see Figure 4.1) starting by sending the body and then sending the headers.

Second, we have the **onBeforeSendHeaders** event that fires when a request is about to occur and the initial headers have been prepared. The event is intended to allow extensions to add, modify, and delete request headers. According to these headers, we can see if it is a file upload. For example, we can look at the method by which this header is being sent and if it is of the POST type, we are on the right track. Another important section to consider is the content-type, which tells you the type of content that this request contains.

The rest of the events in Figure 4.1 are not used because the request already reached the server and canceling or redirecting the request is not allowed at those stages.

While listeners are active, all requests are processed by the extension, so these requests

have to go through a filter. This filter is applied to `onBeforeSendHeaders` where the headers are. These headers have information about the request like the type of the request, the ID, the type of content inside of the body, etc. First, the filter selects POST requests to find just the upload request that it is needed.

Once the POST request has been located, the `Content-Type` has to be filtered to find the **multipart/related**. Each request is identified by a **request ID**. This ID is unique within a browser session and the context of an extension. It remains constant during the life cycle of a request and can be used to match events for the same request. Using the ID of the remaining request from `onBeforeSendHeaders`, request body can be found in `onBeforeRequest`, sharing the same ID. At this point, the body and headers of the request can be accessed and the information about the files can be obtained.

## 4.2. Processing the request

When the request has been already found, the body is on binary array format. *Raw* data has to be decoded. The extension uses **UTF-8** to decode the raw data array and get a readable format. UTF-8 (8-bit Unicode Transformation Format) is an ISO 10646 and Unicode character encoding format that uses symbols of variable length (60). The IETF (Internet Engineering Task Force) requires that all Internet protocols indicate which encoding they use for texts and that UTF-8 is one of the encodings contemplated. `TextDe-`

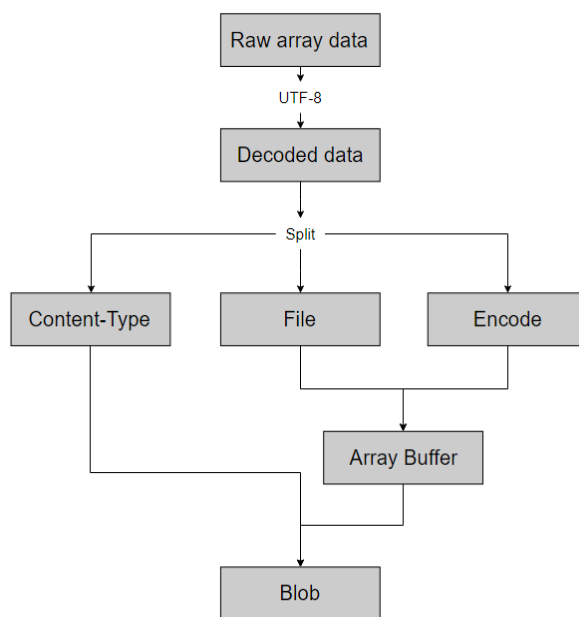


Figure 4.2: Data processing graph

coder is a JavaScript interface that represents a decoder for a specific text encoding, such as UTF-8, ISO-8859-2, KOI8-R, GBK, etc. A decoder takes a stream of bytes as input and emits a stream of code points (40). For Unicode, the particular sequence of bits is called a code point, which is normally assigned to abstract characters. An abstract character is not a graphical glyph but a unit of textual data. Many code points represent single characters

but they can also have other meanings, such as for formatting (35). When the data is already decoded, the result is a `String` with information about the file and a `String` with the content of the file expressed in the encoding indicated above.

In a request body containing file uploads, a "multipart" `Content-Type` field must appear in the entity's header. The body must then contain one or more "body parts," each one preceded by an encapsulation boundary, and the last one followed by a closing boundary. So the information obtained decoding the raw data has to be decomposed into a body part consisting of a header area, a blank line, and a body area, using the boundary as a guide to finding the correct information (15). The last step before sending the file to the cleaning microservices is converting the file into a **Blob** object. Blobs are immutable objects that represent raw data. It can be created from an **ArrayBuffer** object and the content type of the file. `ArrayBuffer` object is used to represent a generic buffer of raw binary data of a specific length. The extension converts Base64 strings of data into an `ArrayBuffer`. This is made with `atob()` method which decodes a data string that has been encoded using base-64 encoding (44).

### 4.3. Data exchange to microservices

This project is based on microservices, where the extension works as an intermediary, knowing exactly where the file should be sent. This structure is further explained in Chapter 2.7.

JavaScript can send network requests to the server and load new information whenever it's needed, and both asynchronous and synchronous requests are used.

The extension must be in contact with different servers, the **orchestrator**, and the **cleaning microservices**.

#### 4.3.1. Orchestrator

Using an orchestrator service for our system is critical because there are many types of data and it is necessary to implement a cleaning microservice for each one of them. The orchestrator is in charge of managing the URLs of the microservices where the data will be sent to be cleaned, according to their content type.

When the extension is turned on, his first task is to communicate with the orchestrator obtaining a list with all the available file formats to be processed by the extension, and their respective URLs. This process is made **asynchronously**. The response is stored in the extension on an array of objects where the key is the `Content-Type` and the value is a pair of the URLs of the different functionalities of the cleaning microservice:

- **Name:** This is the name of the `Content-Type`. It works as a key for searching the correct address for the corresponding file type.
- **Url:** The address of the cleaning microservice.
- **Metadata:** The address that returns the metadata that can be cleaned.

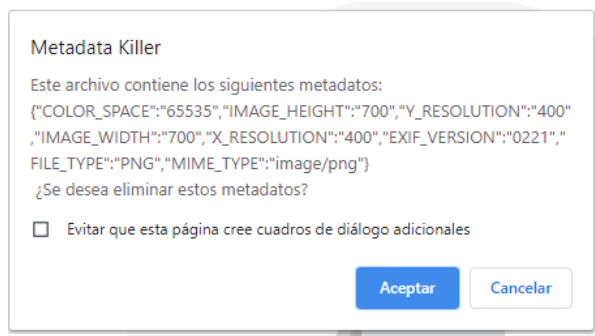


Figure 4.3: Example of notification showing metadata from the file

This information can be used every time the extension wants to clean metadata without the need to make extra requests to the orchestrator. This request must be made asynchronously sometimes to detect new file types that the extension accepts.

### 4.3.2. Cleaning microservices

With microservices, applications are divided into their smallest and most independent elements. The microservices that make up this network have in common the same task: metadata cleanup. Each microservice is responsible for cleaning one type of data. To send the file to the microservice we have to convert the `Blob` obtained previously into a **FormData** object (43). The `FormData` object is used in sending form data but can be used independently from forms to transmit keyed data. The transmitted data is in the same format that the form's `submit()` method that would be used to send the data if the form's encoding type were set to `multipart/form-data`

Cleaning microservices have 2 different functionalities:

- **Show metadata:** This function receives a file, searches for metadata that can be removed and sends it back to the extension as a `JSON`. This function is made synchronously because all the extension must block until the user can see the metadata and decide whether to delete it or not.
- **File metadata cleanup:** This function receives a file and replaces all the fields shown previously with a `"0"` therefore cleaning the metadata. This process is made asynchronously and a `Blob` object with the data of the new cleaned file is received.

Before cleaning metadata, the users are asked if they want to delete the metadata displayed on the screen. If the user decides not to delete them, then the file metadata cleanup is not done, and the file is uploaded. Otherwise, if the user decides to clean the file, it is sent to the microservice to be cleaned. The uploading request is canceled by returning a **BlockingResponse** with the fields `cancel: true` on the callback function from `onBeforeSendHeaders` listeners.

## 4.4. File cleaned

Once the `Blob` object is received from the cleaning microservice the last step is downloading the file. `Chrome.downloads` API (17) allows you to programmatically initiate, monitor, manipulate, and search for downloads. This method requires a URL to download. `Blob` objects can be converted into URL with `URL.createObjectURL()` static method. The extension only works for Chrome browser and Firefox (39), although there is another extension based on `chrome.downloads` that works for other browsers (46). This is further explained in section 4.5. The file is downloaded directly into the `downloads` folder and the user is notified with notification in the system tray as shown in Figure 4.4. These rich notifications are made with `chrome.notifications` API (12).

## 4.5. Limitations found during development

Initially, the idea of the project was to catch the outgoing upload requests, clean the file, and modify the initial request so that the process had been as transparent as possible. But this idea has been refined during the development of the project due to some limitations found in the `webRequest` API. The lack of precise documentation about the `webRequest` API and the small community using it has made it difficult to use the API. The main problems we have faced it were:

- **Modifying ongoing requests:** The idea of modifying ongoing requests made the extension very transparent and easy to use for the user because all he had to do is uploading the file and all the job was automatically done. This idea was not possible to make, because `webRequest` API allows you to modify headers of the request, but not the body of it, where the file is. Official documentation gives us information about modifying headers at `onBeforeSendHeaders` event, but not in `onBeforeRequest`, where the body is. This idea was discarded and instead, we decided to download the new file and canceling the request before it reaches the server.
- **Canceling upload request:** This task should have been easy because canceling requests just require to return a `BlockingResponse` on the callback listener. However, in some web applications like Google Drive, canceling this request produces that the server keeps waiting for the request to reach the server and the browser sending requests with no stop. This means that more requests are involved in the uploading process, adding complexity to the process of canceling the request because there is no information enough to know about all these requests. The solution we came up with was refreshing the page once the main request is canceled, and that worked, but

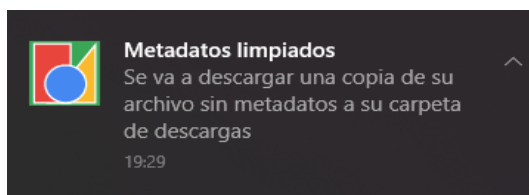


Figure 4.4: Example of notification in the system tray

it pops an alert for refreshing and not saving the changes like it is shown in Figure 4.5.

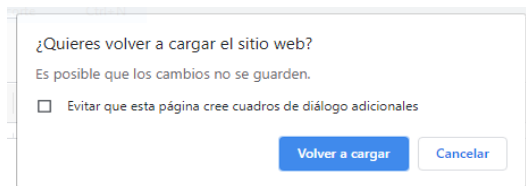


Figure 4.5: Popup when updating the page

This interruption is an automatic alert generated by the browser and cannot be accessed with code, only the user can close it. This makes the extension less transparent making the user interact with it more times than expected, and it could cause a worse experience with it.

- API limitations:** Webrequest API seems to be incomplete according to some posts found in chromium forums indicating that some functionalities are incomplete or bugged (34). The biggest problem was finding those requests that carry the file that is about to be uploaded. The Google platforms (Google Drive, Gmail, etc.) in theory have the same file upload protocols. They use `multipart/related` request types as it is shown in the documentation for developers, but analyzing traffic requests we have discovered that `multipart/related` requests are captured in Google Drive only. This increases the limitations imposed on the extension, just working in Google Drive.

Another request type we studied was `multipart/form-data`. These requests are used in HTML forms, the most simple POST request that can be done. The uploaded file was traveling in a **Request Payload** header and webrequest API does not support this type of headers and the information cannot be handled with an extension.

- Synchronous XMLHttpRequest deprecated:** Most of the requests to the microservices are done asynchronously, but when the metadata query is done, the extension has to pause its execution until the metadata reaches the extension. The use of synchronous methods triggers a warning on the console log like the one shown in Figure 4.6 indicating that synchronous methods are deprecated (36). As explained


 [Deprecation] Synchronous XMLHttpRequest on the main thread is deprecated because of its [background.js:138](#) detrimental effects to the end user's experience. For more help, check <https://xhr.spec.whatwg.org/>.

Figure 4.6: Alert due to the usage of synchronous methods on the main thread

in the XMLHttpRequest documentation, these types of requests will stop working in the near future:

"Synchronous XMLHttpRequest outside of workers is in the process of being removed from the web platform as it has detrimental effects on the end user's experience. (This is a long process that takes many years.) Developers must not pass false for the `async` argument when the current global object is a Window object. User agents are strongly encouraged to warn about such usage in developer tools and may experiment with throwing an "InvalidAccessError" DOMException when it occurs."



- **Extension compatibility:** The usage of APIs like `chrome.downloads` and `chrome.notifications` make the extension only compatible with Chrome browser and Firefox. Chrome implements the APIs using the `chrome.*` namespace with callbacks. Firefox implements the namespace `browser.*` with Promises for all APIs, and `chrome.*` (with callbacks) for almost all APIs (all that are cross-compatible with Chrome) (39). The other API is based on `chrome.downloads` but it is implemented using the `browser.*` namespace (46) and this cannot be used on Chrome.

# Orchestrator

We have already spoken before about the advantages of the Microservices Architecture (MSA) 2.4. But when you follow this architecture, you have to decide on how to integrate the microservices and therefore how are going to interact among themselves. There are two options which have are the most popular ones when using MSA; the *choreography* and *orchestration* techniques (56):

- **Choreography:** In this technique microservices directly interact with each other with no middle man. When a microservice executes a transaction, an event is published and it could be subscribed by one or more microservices to trigger their transactions.
- **Orchestration:** Here one service, the central coordinator, conducts the interactions between microservices. Based on an incoming event, it triggers the next local transactions just in the microservices that concerns.

Having considered both, we decided to follow the Orchestrator technique. This election lightens the work of the rest of the microservices of the project so they can focus completely on their tasks. Also, this approach makes the message flow among microservices more organized, as can be perceived in the representation of both techniques in Figure 5.1. This chapter details the Spring Boot microservice we have developed to act as Orchestrator, and how exactly it works.

## 5.1. Performance

We have developed the orchestrator as an API REST (26) to manage the formats supported by our system. So basically is a web service with a series of methods, which are on different URL waiting to provide certain functionality. Any client can use them by the expected HTTP protocol function, either to get information or to commit it, this information will always be represented as a JSON file.

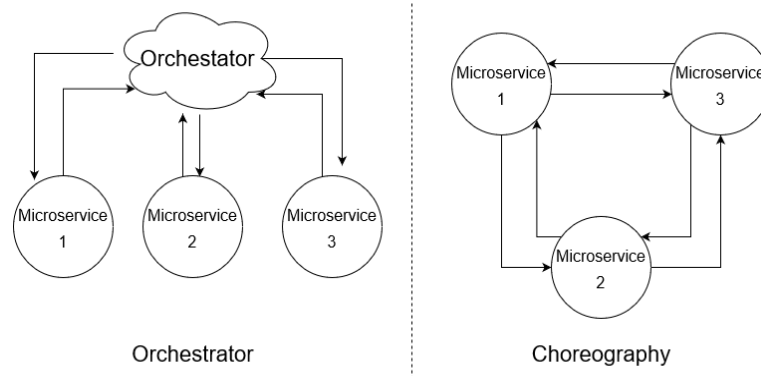


Figure 5.1: The two main techniques to coordinate the iteration among microservices

The performance will be as follows: When the extension receives a request from a user, it will ask the orchestrator microservice for the URL of the service which hosts either the metadata cleaning or the metadata displaying of that format. The orchestrator will consult the database where this information is stored, and tell the extension where the service is hosted.

As well if we want to register any new data format that now is supported, it can be included to the database just with an API call to the adequate function, as explained in section 5.2.4. Same can be applied for modification in case the URL in which a microservice operates changes or the elimination of a format that is not held anymore.

## 5.2. Classes and Interfaces used to create the Orchestrator

In this section, we will detail the five classes that make up this microservice. All of them are in the following part of the repository `Metadatos-2020/MSOrquestador/src/main/java/com/init/formatos/`.

### 5.2.1. MSOrchestratorMain

This is a very simple class, only contains the main function of the application. Spring Boot generates this class by default and it initializes the Tomcat server.

### 5.2.2. Format

This class will represent each of the formats that are supported by our application. It is a plain old Java object (POJO) so it extends neither implements any other class. It has three private class variables which are:

- `String name`: It represents the name of the format which is supported, for example, "PNG" or "JPEG".

- `String eliminationURL`: URL of the service which deals with the format name and which eliminates the metadata.
- `String consultURL`: URL of the same service but this time of the address which will just show the metadata instead of eliminating it.

In addition to these class variables it will have methods `get` and `set` for each one of them, so they could be consulted or modified by the methods in `FormatsREST`, that will be explained later in Section 5.2.4. It counts with a nullary constructor, and another constructor with three parameters, to create an object with all the three attributes already set.

To communicate with the database through JPA (27), we annotate this class as `@Entity`. JPA, will turn the objects of this class to rows so they will be able to be stored in the MySQL database of this microservice, also the transformation will as well be in the other way turning each row from the table to a different `Format` object.

To indicate to Spring Boot which table from the database corresponds to this class we use the annotation `@Table(name = formatsDB)`. As well to create the correspondence among the class attributes and table columns we have to annotate each of the attributes with `@Column(name="columnName", nullable = False)`, where each attribute "columnName" has to match with the column name of the table. The expression `nullable = False` forces to fill these variables, not be possible to leave them with a null value. We also have to indicate which attribute will act as identification, once again this is done by annotations, with `@Id` annotation, the ID in this class will be `name`.

### 5.2.3. **FormatsDAO**

This is the only interface of the microservice. This interface itself extends from another interface, `JpaRepository`, which has two generics. The first one tells from which entity does the DAO, in our case `Format`. And second interface configures which datatype is its ID, so for us, it will be `String`.

Only by doing this, we are implementing all the functionalities that the DAO pattern in Spring Boot has (2). These functions will us to make use of its functions, which will come as a great help in the REST class 5.2.4 to perform consults to the database.

Another reason to use the DAO pattern is to separate the functionality of the logic of this microservice from the storage platform we have chosen. So if for example, we decided to change our database from MySQL for another one, we tried and is detailed in Section 5.5, all the logic of the system will stay the same as nothing related to it has changed. The DAO acts as a runway between our database which ensure the persistence and the logic of our system, so if we commit the change of storing service, only this class will be affected.

### 5.2.4. **FormatsREST**

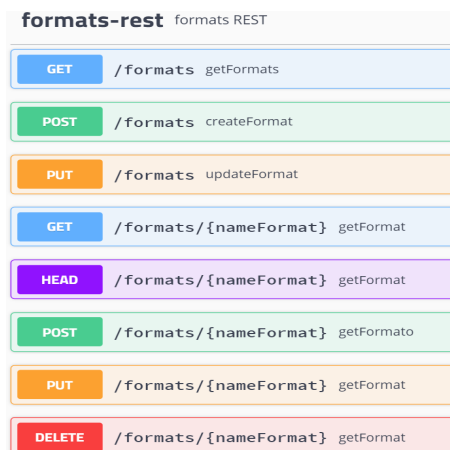
This class has all the REST services that this microservice provides. To indicate this to Spring Boot we have to use the annotation `@FormatsREST`. Also to indicate

the URL where this service will listen to connections we have to use another annotation `@RequestMapping("/formats")` which will make this microservice work from the address `localhost:8085/formats`, in Section 5.5 is explained why this port.

It has a `FormatsDAO` object, the one explained in Section 5.2.3, as the only attribute, to carry on the communication with the database. To use it we have to make the annotation `@Autowired` which will inject a real object. This is necessary because `FormatsDAO` is an interface, and interfaces can not be instantiated, but Spring Boot internally generates a proxy that when running the application will provide us with a valid `FormatsDAO` object.

The methods of this class has are:

- `getFormats`: This function will return a `List` of all the `Format` objects held by the application. Is going to be annotated as `@GetMapping` so it can response in the address `localhost:8085/formats` to the methods `GET`. It uses the `FormatsDAO` method `findAll()` to get them all.
- `getFormat`: This is a similar function of `getFormats`, but this will only return one certain `Format`. It recives as parameter a `String` `name` telling the name of the format we want to consult. As `getFormats` is already listening in the address `localhost:8085/formats` for `GET` methods we need a different address to listen it. To solve this we will use `@RequestMapping(value = "nameFormat")`, so this function will be listening to `@GET` methods in the address `localhost:8085/formats/nameFormat`, where `nameFormat` corresponds to each of the names of the formats present in the database. The search for that particular format this time will done by the `FormatsDAO` method `findById()`. As result of this fuction the `Format` object itself will be return if it was found and a `HTTP` Status code `no-content` otherwise.
- `createFormat` : Will insert a new `Format` that we are able to handle to the database. As parameter it recieves the object `Format` that we want to store. It will use the annotation `@PostMapping` so will be listen in the address `localhost:8085/formats` like `getFormats` but this time waiting for `POST` methods. This function will use the `FormatsDAO` method `save()` to store the `Format` in the database. It will return the `Format` object that we just inserted, in case the service which made the call wants to check that spellcheck went as expected.
- `deleteFormat` : This function is used to eliminate of formats which do not hold any more. Same as `getFormat` it listens in all the addresses corresponding to each of the `Format` present in our database. But unlike that, this service will wait for `DELETE` methods as is told by its annotation `@DeleteMapping(value = "nameFormat")` on the address `localhost:8085/formats/nameFormat`. It will have a parameter `String` `nameFormat` which indicates the `FormatsDAO` which `Format` to eliminate. The `FormatsDAO` object will perform the elimination by using `deleteById()` being this ID `nameFormat`. The answer from this method will be an empty body, but we will be able to know it was successful because also will produce the code `200`.
- `updateFormat` : Is in charge of changing the values of an object `Format` in case the URL where the corresponding service is hosted changes. It will work on the address `localhost:8085/formats` waiting for `PUT` methods, this is indicated



formats-rest formats REST	
GET	/formats getFormats
POST	/formats createFormat
PUT	/formats updateFormat
GET	/formats/{nameFormat} getFormat
HEAD	/formats/{nameFormat} getFormat
POST	/formats/{nameFormat} getFormato
PUT	/formats/{nameFormat} getFormat
DELETE	/formats/{nameFormat} getFormat

Figure 5.2: Documentation generated by Swagger, showing all the functions available by out API REST.

by the annotation `@PutMapping`. The calls to this service will be done attaching to them a JSON object, showing how we want that `Format` to look like after the update. First we will find that `Format` by using `findById()` and then with the `Format` setters we modify it to look as the JSON object received. It will return a `Format` showing how the update looked like.

### 5.2.5. SwaggerConfig

This class generates the documentation of the whole microservice. If someone wants to make use of our services, with no need to look at our code, will be able to know for each method, in which URL responses, which HTTP method is waiting for, which parameter is expecting and what will be the answer. Has just one method, `api()` which is annotated by `@EnableSwagger2` to enable Swagger to auto-document our services. All the documentation will be displayed in a very useful graphic interface where we can even try each of the methods, on the address `"/swagger-ui.html"`. An example of how this documentation looks is in Figure 5.2

## 5.3. Consistency

To keep the data consistency in this microservice and avoid losing the formats our application can work with, we decided to use a database to store them. As the amount of data we store is small, the decision of using MySQL is not optimal, as is a heavy platform designed to work with huge tables. But due to difficulties which we explain in Section 5.5 and that Spring Boot offers easy integration for MySQL (24), we decided to use it.

As detailed in the class `Format` in Section 5.2.2, JPA makes the interaction with the database quite easy, just needing the annotations to do the transformations between objects and SQL instructions and vice versa. In the MySQL environment, we just need to generate the table with the same name as the name used in the annotations. To let Spring Boot know the details of the database connection such as the username or the password we have

to modify the configuration file, specifying these properties. It is useful also to indicate in this file the property `ddl-auto: update` so the table will be filled in automatically from the entity class every time we launch the project. Another property which needs to be set is `dialect: org.hibernate.dialect.MySQL5Dialect`, which defines the language used to carry the previously mentioned transformations.

## 5.4. Libraries employed

Along with the classes and interfaces detailed in Section 5.2 to build this orchestrator microservice we have used some libraries to simplify certain actions. These libraries are:

- `Spring Boot starter web` (55): When we run the `main()` method, this library will start a Tomcat server as an embedded container so we do not have to use an external server. This in addition to simplifying a lot the architecture of the microservice, allows it to run in just a few seconds.
- `mysql connector java` (24): This one includes the MySQL driver which allow us to connect to that Database.
- `spring boot starter data jpa` (25): Java Persistence API (JPA) let us convert objects to SQL instructions and the other way, turn the result of a SQL consult to an object.
- `sprinfox swagger ui` (8): This library give us the possibility to auto-document our services employing the metadata on the annotations used on the functions.

## 5.5. Difficulties and considerations

- We tried to use a lighter database more lined with our needs, namely SQLite (9) instead of MySQL. Unfortunately, there was not much documentation on how to integrate it in Spring Boot and we were not able to modify the properties file to get it. So finally we stayed with MySQL.
- We have decided to run the whole web application in the port `8085`, therefore, avoiding the default port, `8080`, and the possible port conflicts that usually it causes. As well the communication with the MySQL database is carried on through the port `3306`.

# Chapter 6

## Metadata Management

This chapter provides the implementation detail of the microservice in charge of managing the metadata of the provided file. As discussed in previous chapters, you can develop as many microservices as you want to manage

### 6.1. Implementation details

The microservice has been developed in Java (7) with the main technologies mentioned above. It provides two resources that will be consumed from the outside, in this case from the browser extension.

The resources developed in the application are as follows:

- Viewing metadata from a proposed file
- Remove metadata from a provided file

#### 6.1.1. Viewing metadata from a proposed file

POST service with the following characteristics:

**Name:** /metadatos/mostrar

**Content-Type:** multipart/form-data

**Produces:** application/json

The service receives the file as a parameter, and returns the corresponding metadata.

```
@RequestMapping(value="/metadatos/mostrar",method = RequestMethod.POST,  
consumes = "multipart/form-data", produces = "application/json")
```

```
public ResponseEntity<String>
```



```

leerMetadatos(@RequestParam("file") MultipartFile file) throws
    Exception
{
    return new ResponseEntity<>(mostrarMetadatos(file),
        HttpStatus.OK);
}

private String mostrarMetadatos(MultipartFile file) throws Exception
{
    File multToFile =convertToFile(file);
    Map<Tag, String> meta =process(multToFile, null);
    Map<Tag, String> metaNuevo = new HashMap<>();
    for (Map.Entry<Tag, String> entry : meta.entrySet()) {
        if (entry.getValue() != null && !entry.getValue().isEmpty()
            || !entry.getValue().equals("0"))
        {
            System.out.println("clave=" + entry.getKey() + ",
                valor=" + entry.getValue());
            metaNuevo.put(entry.getKey(), entry.getValue());
        }
    }
    ObjectMapper objectMapper = new ObjectMapper();
    String metadatos = objectMapper.writeValueAsString(metaNuevo);
    return metadatos;
}

```

---

*Line 6:* The `mostrarMetadatos` method, after converting the file using the `convertToFile` method, the `process` method is invoked which queries the metadata. Finally, only the metadata tags that have a value other than null will be shown.

*Line 9:* The `convertToFile` method receive as parameter the file in `Multipart/form-data` format and returns the file in `File` format for its later treatment in the subsequent processes.

The `process` method is the transversal process, both for the consultation service and for the deletion service. In this method, the `Exiftool` library is used, providing the interface for method management.

```

public static Map<Tag, String> process(File image, Map<Tag, String>
    meta)
throws Exception {

    try {
        ExifTool exifTool = new ExifToolBuilder().build();
        if (meta != null) {
            exifTool.setImageMeta(image, meta);
        }

        Map<Tag, String> metadatos = new HashMap<Tag, String>();
        metadatos =exifTool.getImageMeta(image, asList(
            StandardTag.ISO,
            StandardTag.APERTURE,
            StandardTag.WHITE_BALANCE,
            StandardTag.CONTRAST,

```

---

```

StandardTag.SATURATION,
StandardTag.SHARPNESS,
StandardTag.SHUTTER_SPEED,
StandardTag.DIGITAL_ZOOM_RATIO,
StandardTag.IMAGE_WIDTH,
StandardTag.IMAGE_HEIGHT)

```

---

*Line 1:* This process receives as parameter the file and the metadata, providing two functionalities, a reading and a writing functionality.

*Line 9:* On the one hand, if it receives null value in the metadata field, it returns all the associated tags, corresponding to metadata of the file it receives as parameter.

*Line 5:* On the other hand, if it receives a value other than null in the metadata field, it updates the metadata of the received file with the received metadata.

### 6.1.2. Remove metadata from a provided file

POST service with the following characteristics:

**Name:** /metadatos/eliminar

**Content-Type:** multipart/form-data

**Produces:** ResponseEntity<byte[]>

*Line 3* The service leerMetadatos receives the file as a parameter, and returns the file with the deleted metadata. The service queries the tags related to the metadata that have value, removing only metadata that have non-null value.

*Line 15:* The removeMetada method, receive the file and returns the file with the deleted metadata. Initially, the convertFile method is invoked to obtain the file in the necessary format for its later treatment.

*Line 18:* Finally the process method is invoked, which will be the method that removes the metadata from the file.

```

@RequestMapping(value="/metadatos/eliminar",method = RequestMethod.POST
,
consumes = "multipart/form-data")

```

```

public ResponseEntity<byte[]> eliminarMetadatos(@RequestParam("file")
    MultipartFile file) throws Exception
{
    File fileNuevo =removeMetadata(file);
    byte[] fileContent = Files.readAllBytes(fileNuevo.toPath());
    if (fileContent!=null)
        return new ResponseEntity<byte[]>(fileContent ,
            HttpStatus.OK);
    else
        return new ResponseEntity<>(null ,
            HttpStatus.INTERNAL_SERVER_ERROR);
}

```

```

}

public File removeMetadata(MultipartFile file)
    throws Exception {
    File fileMetadataRemoved =convertToFile(file);
    Map<Tag, String> metalectura =process(fileMetadataRemoved, null);
    Map<Tag, String> meta = new HashMap<>();

    for (Map.Entry<Tag, String> entry : metalectura.entrySet()) {
        if( entry.getValue()!=null    &&
            !entry.getValue().isEmpty() ||
            !entry.getValue().equals("0")
        )
        System.out.println("clave=" + entry.getKey() + ",
            valor=" + entry.getValue());
        meta.put(entry.getKey(), "0");
    }
}

```

---

## 6.2. File formats

Specifically, a microservice has been developed that manages the following formats:

AFCP, AIFF, APE, APP0, ASF, Composite, DICOM, DNG, DV, DjVu, Ducky, EXE, EXIF, ExifTool, FITS, FLAC, FLIR, File, Flash, FlashPix, Font, GIF, GIMP, GeoTiff, GoPro, H264, HTML, ISO, ITC, JFIF, JPEG, JSON, Jpeg2000, LNK, Leaf, Lytro, M2TS, MIE, MPEG, MPF, MXF, MakerNotes, Meta, Ogg, OpenEXR, Opus, PDF, PICT, PLIST, PNG, PSP, Palm, Parro, PhotoCD, PhotoMechanic, Photoshop, PostScript, QuickTime, RAF, RSRC, RTF, Rad Real, Re, SVG, SigmaRaw, WTV,XML, XMP, ZIP

---

## 6.3. Appendix

The following is a list of the libraries that have been used in the development of the microservice

### Maven

Maven (6) is installed as follows:

- **Step 1:** Download Maven

To do so, access the url <https://maven.apache.org/download.cgi>. In the upper part there is the suggested mirror server to download the files.

- **Step 2:** Unzip the file

Unzip the file on the hard disk, in the path you find most convenient. In my case I unzipped it in C:\Program Files\ApacheSoftwareFoundation\Maven, since the folder

Apache Software Foundation already existed in my system previously when I installed Apache Tomcat.

- **Step 3:** Create the environment variables for Maven

```
M2_HOME=C:\Program Files\ApacheSoftwareFoundation\Maven\apache-maven-3.2.2
```

```
M2=%M2_HOME%\bin
```

- **Step 4:** Add the M2 path to the environment variable PATH:

```
PATH=...;%M2%
```

- **Step 5:** Finally, we open a command line, and we run

```
mvn -version
```

## Exiftool

ExifTool (4) is a Java (7) component for reading and writing metadata in a wide variety of files, including the manufacturer's note information of many digital cameras from various manufacturers such as Canon, Casio, DJI, FLIR, FujiFilm, GE, HP, JVC / Victor, Kodak, Leaf, Minolta / Konica-Minolta, Nikon, Nintendo, Olympus / Epson, Panasonic / Leica, Pentax / Asahi, Phase One, Reconyx, Ricoh, Samsung, Sanyo, Sigma / Foveon and Sony

The installation of the library is done as follows:

- **Step 1:** Install Exiftool

Install Exiftool, because when you run the project with the library, it will look for the \*.exe of the application, if this is already included in the system PATH the application will run without a problem, otherwise, you must add the \*.exe in the base folder of the project as seen in the image below. Referenciado en Figura 6.1

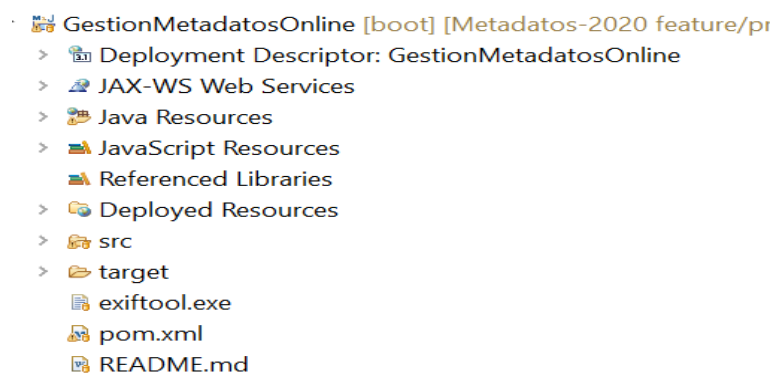


Figure 6.1: Exiftool.exe library integrated in the project

- **Step 2:** Dependence

Add the following dependency:

```
<dependency>
  <groupId>com.github.mjeanroy</groupId>
```

```
    <artifactId>exiftool-lib</artifactId>
    <version>2.1.0</version>
</dependency>
```

---

## Spring boot

The installation of Spring boot (5) in eclipse is done as indicated:

Spring Boot (5) dependencies use the `org.springframework.boot` groupId. Typically your Maven POM file will inherit from the `Spring-boot-starter-parent` project and declare dependencies to one or more “Starter POMs”.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.3.RELEASE</version>
</parent>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>
</dependencyManagement>
```

---

# Contribution

## 7.1. Diego's contributions

Diego was involved for the first weeks after the first meeting with the rest of the team in the investigation of the feasibility of the initial idea of the project. As his knowledge in this topic was not as good as it came to be, he did not find some limitations that later we got, especially related to the extension and the petition handle. Also during those first weeks of research, he got more familiar with what metadata was. He could find a lot of bibliography and valuable resources like the books *Metadata* (53) and *Introduction to metadata* (47). Also, he made some hands-on research, proving on different file formats how and how much metadata was generated with their creation and modification. As well he proved to share different documents to check if those messenger services cleaned the metadata or not. After those first weeks of research, with the rest of the components of the team, he agreed that the initial idea of an online service to eliminate the metadata form files though a plug-in was something that could be accomplished.

After the second meeting, the idea of organizing the project in independent microservices was accepted. He was assigned to develop the orchestrator microservice among with looking for applications which target as well the treatment of metadata. For the microservice first of all, he had to investigate what microservices were, as he had just heard of the concept but never really work with it. Diego enjoyed this part of the project as he learned a lot, not just about the microservice architecture but also about the development of web services. All this time of documentation and research came as a great help to him when writing the memory.

Once Diego had a clear idea of what a microservice was and how did they work, he looked for what technology employ to carry it out. For the language, there were almost as many possibilities as programming languages exist. He focussed his quest on those which he had used before so he could use that background, those were Python (61), PHP (57), C (49) and Java (58). After this research, Diego thought that the best would be either Python or Java, as there were the ones with the greatest amount of documentation and with a wider and most active community of developers, which always is a great help when looking for common errors. Finally, Diego chose to do it in Java for the next reasons: it was

the language he was more familiarized with, so the learning curve would be smaller. The website "Baeldung" (1) offered free high-quality courses on how to develop microservices, so it was a great start point. The existence of frameworks which helps a lot at the time of developing application such as Spring Boot.

The next step was starting to code, the integrated development environment (IDE) chosen by Diego to start to code was Eclipse, as it was the IDE he had used the most. He employed a tool that Eclipse offers to incorporate all the Spring boot features to Eclipse (10). Nonetheless, Diego switched to Spring Tool Suite (STS) in a later phase of the development process. This change was motivated by the difficulty he found in integrating the database to the microservice using the above-mentioned tool. As STS is completely focused on developing Spring Boot services, aspects as the configuration of the database or the use of external libraries were easier to accomplish.

Appart of that complication the development of the orchestrator microservice which took Diego the next month and a half was placid. He got familiarised with the HTTP methods which once again was something he had the motion of but never before had the chance to work with. He programmed some functionalities that he thought could be useful but when talking with the other components of the team realised that they were not. These functionalities were: the possibility of not just deleting a certain format but also being able to delete all the formats stored at once, acting as a reset function. The other discarded functionality was thought to register a service with only one of the two URL available, this is having a service which only eliminates or consults the metadata but not both.

Once the orchestrator microservice was finished and Diego had tested it, he tried to help Guillermo with the extension as it turned to be the part of the project where the most difficulties were found. In that particular stage of his development, Guillermo was trying to make the extension work as well in Gmail, as it was already functional in another Google service as Drive. Unfortunately, there was not much documentation that could not help them, and the few knowledge Diego had about Javascript, in addition to how to advance the project was in that point, made Diego not very useful. On the other hand, the integration of the orchestrator with the extension went very well, being completed in just some hours.

On the last part of the development process of the project, Diego tried to create a new microservice to handle the metadata of the PDF formats, as this format is one of the most commonly used. Following what Rosa accomplished with the images, he looked for a library which enabled him to work with the metadata of PDF, discovering iText (3). This library provides a series of functions to work with PDF files, including some oriented to metadata. Diego was able to clean metadata from files he had stored locally, but he could not extend this to the files he received online, therefore this service was not included to the project. However, we believe that is feasible to get this service completely functional with some extra work in that part.

## 7.2. Guillermo's contributions

Guillermo has been in charge of researching and developing everything related to the extension. In the first weeks of the project, he investigated the development of the extension due to ignorance of the development of these. In this time he took an online course on

YouTube called "Programming from A to Z" by Daniel Shiffman (known as "The coding train" on YouTube) (59) where he learned how to make his first browser extensions.

To practice, he developed several varied extensions to assimilate the concepts seen in the online course. The first test with extensions was a Chrome browser extension that pops up with the phrase "Hello World!" when you click on the extension icon. Next, he developed an extension that changes all the images on a website into kitten photos. Finally, he made an extension that had a window that an image would appear when activating the icon of the extension.

Once the basic concepts of extensions were assimilated, he began with the research on `webrequest` API (16) which will be used during the development of the entire project. The objective at the beginning was to print in the console all the requests of every event available in the API (more information available in the Chapter 4) and compare them with the requests of the browsers developer mode (F12 mode). The first attempt working with these outgoing requests was to make an HTML server where only one file could be uploaded. At the same time, the same tests would be done on Google Drive, since it is a more realistic scenario for the use of this extension. Most of the time invested in this part of the project was spent searching for information on requests and information on the `webrequest` API. No information was found on how to extract the files uploaded from requests in HTML forms, so from that moment, the investigation was done entirely on Google Drive since there was information about the requests that contained uploaded files.

The following month was dedicated to the processing of the data from the request. This process was slower since on those dates more time was needed to carry out tasks from other subjects. To validate that the image was being processed correctly, the `base64` code obtained was tested on a page (37) where it converts the `base64` code into a visible image. Many tests were necessary since the information on these requests was not found until several tests later, failing some of those attempts. The processing of this data involved many modifications regarding the information obtained from the request, such as dividing it into information sections to discard unnecessary information, decode, encode, etc. Each of these operations requires a few days to obtain information and prove that they work individually.

The next milestone corresponding to Guillermo's work was to connect all the components of the project, that is, the orchestrator and a cleaning microservice. To learn about making connections with other services, a test extension was first created and then it was integrated into the project's extension code. This test consisted of sending information to the orchestrator that was written with `XMLHttpRequest` and seeing if the database was modified. This option at the beginning was not viable because working with `XMLHttpRequest` (45) was confusing and was changed by the `fetch` API (41) which simplified the code and the connection with the orchestrator was a success from the first attempt. This first connection to the orchestrator took about two days of work. The first connection to the cleaning microservice took approximately one week of work. This is due to the format incompatibilities that were sent by the extension since they did not coincide with the microservice. Guillermo also had to know the technologies used for each component of the project to be able to make the necessary adjustments so that everything could fit together. To do this, he had to watch videos on the internet about the basic knowledge of Springboot since it was unknown to him. Regarding the database, he downloaded and learned to use My Workbench SQL, which is connected to the orchestrator.



Once all the components are working at the same time, the next step for the development of the application would be to cancel the request where the processed file is found. It took several weeks of parallel work with other tasks of the extension to investigate on canceling this type of requests since the first tests carried out were all unsuccessful. This functionality would not be perfectly implemented because there is no information about it in official manuals and an alternative solution had to be chosen.

Another of the tasks that he has tried to solve is being able to use the extension on every web page. This took 2-3 weeks without any results, as the webrequest API did not offer enough information. It was not possible to know all the sections where there can be a file. This was a failure, so, together with the project development team, they decided to limit the functionality of the project to Google services (such as Google Drive, Gmail, etc.). This option was still not feasible given that no clear information about file upload requests could be found in the other Google services, so the team decided to limit the project to Google Drive only, and then, as future work, it could be implemented in different platforms or web pages.

The last two weeks of development were spent on optimizing certain parts of the code and correcting errors. Not many errors were found but enough tests were carried out to verify its correct operation. Changes were also necessary for the part of the code that processes the request information for its proper functioning.

### 7.3. Rosa's contributions

This chapter describes the contributions that Rosa Olivia Zumaeta Sánchez made to the project. Those contributions are classified in the different stages that the development of the process went through.

#### 7.3.1. Stage 1: Realization of a test browser extension

The initial objective we set was to obtain enough knowledge about the development of browser extension and how do they work. Since we did not have this knowledge, we dedicated the first weeks to learn about browser extension, doing some tests like displaying an icon in the browser. At this point, the first problem arose from the version of the manifest. This was because the latest version of Chrome had changed the field `manifest-version`, that now must be included with value two. After finding the correct value, the browser extension icon could be displayed and visualized in Chrome.

After that initial test, the next objective was to offer more functionalities to the browser extension. We worked to display a file selector, so the user could select a file, and then the properties or data of the file would be displayed. At this point, it was a matter of assigning JS and HTML files to the initial browser extension. After several investigations of different portals, the changes described were made.

Initially, the code was in Javascript and HTML, although the final idea was to use jQuery or AngularJS. To end with this stage, we made a small POC, using a FileReader to display the data of a file selected from a Chrome extension.

### 7.3.2. Stage 2: Browser extension study

During this stage, our goal was to check for which browsers we could develop the project. For practical purposes, we thought that Chrome and Mozilla were great candidates since they use the same procedures to implement the browser extensions.

### 7.3.3. Stage 3: Feasibility of the proposed browser extension

For this stage, each member of the team was assigned a task. Rosa, in particular, was in charge of developing a microservice that could manage the metadata of a file.

When she started with the development she found difficulties with the installation. Using Eclipse as the IDE to work with Spring Boot, generated lots of errors until she was able to finally started Eclipse correctly. She was able to accomplish that after configuring the libraries and properties needed for the inclusion of the dependency with Spring Boot.

During this research process, she observed that a specific IDE, for example, Spring Tool Suite (STS), could have been a better choice for this task. STS is an Eclipse-based IDE with Spring Boot integrated on it, offering the developer to get away from configuration issues, including the configurations needed to develop microservices.

After having an environment working on a *Hello world*, the next she faced was to start with the development of the metadata consultation service. Initially, the metadata query was done completely by Rosa, without using any library. This initial program was the one she uses for the proof-of-concept she had with the rest of the team. But when she tried to develop a service to remove the metadata from the file (a task that involved several tests) the possibility of using a library was considered due to the difficulty of the task.

To continue with the removal of the metadata, she decided to focus on finding a library that provided image metadata management, finding the library she finally decided to use, Exiftool. The installation of the library did not work properly the first time since she did not pay attention to the executable file required the installation, using only the previously commented dependency.

Once the library was properly installed, and after developing and testing the functionality to remove metadata, the last step was to invoke from a client the services of consulting and deleting metadata. She found that the content-type for sending images is multipart/form-data, and therefore was needed to implement a test client, based on an HTML that allowed the selection of an image file.

The action was modified to invoke the metadata query or metadata deletion service as appropriate. During the testing phase, she noted that the service queried all the metadata fields present on the file, even if those fields did not have any information, returning most of the time a lot of empty data because of those fields. To solve this she had to recode the program to make it return only those fields which had information.

Something similarly happened for the metadata erased. The query deleted all metadata tags, even those tags that were not present in that type of file, so she modified it to only delete those metadata tags which had value.

During the integration tests, the team observed that the file that was returned by the microservice when the metadata was deleted was not supported by the browser extension developed by Guillermo because the browser extension expected the file as an array of bytes. To solve this Rosa made a change in the way the resource was being sent so it matched the expected format.

After the development of the microservice and the fixing, the complications emerged during the integration, the memory is written.

#### **7.3.4. Stage 4: Memory report**

In this part, Rosa's contribution focused on the following: The introduction of the report, the chapter in which Rosa's contributions to the team were detailed, the chapter corresponding to the implementation of the microservice that manages the metadata of a file, and along with the other members of the team the conclusions she obtained while working on this project.

## Conclusions

In general terms we are quite happy with the results of the project, as is quite similar to those we planned at the beginning of the development. But we should mention the great amount of complications we faced during the evolution of it. Most of them occasioned by the extension, as explained in Chapter 4.

What we have accomplished is a completely functional application which detects the Google Drive requests, offering the user the possibility of cleaning the metadata of the uploaded files if these are among the formats we support.

At the moment the application supports image formats such as JPEG, PNG, GIF, etc. But being designed as a microservices-based architecture, adding new formats would not be a problem, since there is no need to modify code in the different components of the system. It is extensible to all the desired formats and for any available platform, being able to create each microservice in a different technology or programming languages.

Another thing we value very positively is the chance we have had to use modern technologies, which are widely used in the industry but not so in the academic world. Examples of these are Spring Boot, JavaScript, microservices architecture, or the use of external libraries such as JPA or Spring Boot starter web.

A considerable part of the time we invested in this project was directed to investigation and discover if different ideas were feasible or not. Having our work in this project as a base, we believe there is a great potential to include new functionalities making the "browser plug-in for metadata removal" more useful.

### 8.1. Future work

The main areas where this project could be extended, with not much worked needed and making it a more complete product would be:

### 8.1.1. Extend the functionality to more formats

What we regret the most was not being able to extend the functionality we built to work with other metadata formats but images. As we were three components in the group, one was focused on the orchestrator, other on the extension and just one person was dealing with the formats. Our initial idea at the early stages of the project development was to be able to treat the metadata in formats like:

- *PDF*.
- *OpenDocument* mainly *.odt*, *.ods* and *.odp*.
- The most popular Microsoft Office formats like *.docx*, *.xlsx* and *.ppt*.
- Some video formats like *.mp4* or *.avi*.

This was a very optimistic approach, because each of them would require from a different microservice, as each format will have different functions or libraries to deal with the corresponding metadata. Nonetheless taking advantage that the extension and the orchestrator are completely developed, future work will be only required on this side of the project. Using frameworks like the ones we have already used as Springboot will speed up this development. As well as having already a working microservice like the one we developed to deal with images, is a great reference. Therefore, finding the the proper library to deal with the metadata of the new format is the key of any future expansion of the project.

All those suggestions were proved while the development of the microservice we made to deal with the *PDF*. This feature is not included in the project as was not completed, but we were close being able to modify the metadata from PDF documents on the file system, as is explained in detail in Chapter 7.3.4.

### 8.1.2. Extension

As detailed in the Chapter 4 the extension has had numerous errors that could not be solved due to problems with the API or insufficient information. The task that should be improved the most in the future is to extend the extension for every possible web page that has to file uploads. This can be implemented as the `webRequest` API improves, offering more information on requests so that files that are not in the body of the request can be accessed by the extension.

Another possible improvement to the project is the cancellation of requests when the user decides not to upload the file to be cleaned. In the current state of the project it has not been implemented in an optimal way, so the user has to do some more interaction with the extension than expected. This can be improved as the API improves or the upload requests are standardized so that it is the same for all web pages.

Finally, the extension is only available for the Chrome browser and Firefox, so it would be necessary to rewrite some functions to make them compatible with other browsers or offer the same extension in other browsers separately.

### 8.1.3. Move to from local to online

The project is currently working locally, that is, the orchestrator and cleaning microservices are installed on the user's computer and working at the same time. Having the entire project locally is necessary for its proper development since it allows testing with the other components of the project, but it is not a solution for its use with real users. Therefore, the services that make up the extension (see Chapter 2.7) must be uploaded to Internet servers. This allows you to have as many cleaning microservices as you want at different URLs, notifying the orchestrator where those microservices are located. The user does not need to start the cleaning microservices for the entire system to function, allowing only the user to interact with the extension. In addition to the CPU load that these microservices could be using, it saves having to download and execute these components.

### 8.1.4. Extend this service to other platforms

We approach this project with the idea of making the project working with the least interaction from the user as possible. That was the reason we used the browser extension as the component which looks for outgoing HTTP connections. In that sense, an extension is an ideal tool, but the microservices we developed could work in very different frames. One of the advantages of the microservice architecture is the independence they have from other components of the system, as is explained in Chapter 1.2.

This independence makes them greatly reusable, as to how they communicate is through API calls, they could be called from very different platforms. For example, if we develop a desktop application, the only code that would be required will be the one necessary to develop a user interface and the calls to our microservices. Same could be applied to a mobile app, as most of the communications people have nowadays is through mobile phones, this is a very interesting future work. Once again all the part which deals with both the user and the leaving petitions with a file attached will need to be code, but once that is done, the treatment of the metadata could be done in our microservices

One last platform is to develop a complete website instead of an extension. This could be useful if we wanted to have greater interaction with the user, as in the extension this is limited to the small window it uses. Once again all the microservices and the advances achieved in this project could be reused for it.



# Bibliography

- [1] Baeldung website. <https://www.baeldung.com/>.
- [2] A controller, service and dao. <https://www.baeldung.com/jsf-spring-boot-controller-service-dao>.
- [3] Discover itext pdf. <https://itextpdf.com/>.
- [4] Exiftool by phil harvey. <https://exiftool.org/>.
- [5] Installing spring boot. "https://docs.spring.io/autorepo/docs/spring-boot/1.0.0.RC5/reference/html/getting-started-installing-spring-boot.html".
- [6] Learn apache maven build automation tool. [https://www.tutorialspoint.com/maven/maven\\_overview.htm](https://www.tutorialspoint.com/maven/maven_overview.htm).
- [7] Oficial web for java. <https://www.java.com/es/download/faq/java8.xml>.
- [8] Setting up swagger 2 with a spring rest api. <https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api>.
- [9] Spring boot with sqlite. <https://www.baeldung.com/spring-boot-sqlite>.
- [10] Spring tools for eclipse ide. [https://www.eclipse.org/community/eclipse\\_newsletter/2018/february/springboot.php](https://www.eclipse.org/community/eclipse_newsletter/2018/february/springboot.php).
- [11] What is metadata? <https://www.lifewire.com/metadata-definition-and-examples-1019177>.
- [12] Chrome notifications developer guide. <https://developer.chrome.com/apps/notifications>, (accessed June 12, 2020).
- [13] Spring data jpa. <https://spring.io/projects/spring-data-jpa>, (accessed June 12, 2020).
- [14] What are microservices? <https://aws.amazon.com/es/microservices/>, (accessed June 12, 2020).
- [15] Rfc1341(mime) : 7 the multipart content type. [https://www.w3.org/Protocols/rfc1341/7\\_2\\_Multipart.html](https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html), (accessed June 5, 2020).



- 
- [16] Webrequest api documentation website. <https://developer.chrome.com/extensions/webRequest>, (accessed June 5, 2020).
- [17] Chrome api for downloading files. <https://developer.chrome.com/extensions/downloads>, (accessed June 9, 2020).
- [18] Github 2.0 a small place to discover languages in github. [https://madnight.github.io/github/#/pull\\_requests/2020/1](https://madnight.github.io/github/#/pull_requests/2020/1), April 12, 2019 (accessed June 10, 2020).
- [19] Most popular google chrome browser extensions ever. <http://www.skipser.com/p/2/p/most-popular-chrome-extensions.html>, April 12, 2019 (accessed June 10, 2020).
- [20] Why whatsapp (and telegram) messages are not really private? <https://www.andreafortuna.org/2019/08/12/why-whatsapp-and-telegram-messages-are-not-really-private/>, April 12, 2019 (accessed June 10, 2020).
- [21] Phishing attack. <https://www.adarsus.com/phishing-que-es/>, April 12, 2019 (accessed June 8, 2020).
- [22] Spring framework - spring. <https://spring.io/projects/spring-framework>, April 12, 2019 (accessed June 8, 2020).
- [23] Metadata analyzer - elevenpaths. <https://www.elevenpaths.com/es/labstools/metashield-analyzer/index.html>, April 2, 2019 (accessed June 12, 2020).
- [24] Accessing data with mysql. <https://spring.io/guides/gs/accessing-data-mysql/>, April 22, 2019 (accessed May 10, 2020).
- [25] Accessing data with mysql. <https://spring.io/guides/gs/accessing-data-jpa/>, April 22, 2019 (accessed May 12, 2020).
- [26] Rest api tutorial. <https://restfulapi.net/>, April 22, 2019 (accessed May 12, 2020).
- [27] What is jpa? introduction to the java persistence api. <https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html>, April 22, 2019 (accessed May 12, 2020).
- [28] Exiv2. c++ metadata library and tools. <https://www.exiv2.org/index.html>, April 3, 2019 (accessed June 10, 2020).
- [29] Pdf metadata editor. <http://broken-by.me/pdf-metadata-editor/>, April 3, 2019 (accessed June 10, 2020).
- [30] Batchpurifier. batch multi-format hidden data & metadata removal software tool for windows. <http://www.digitalconfidence.com/batchpurifier.html>, April 3, 2019 (accessed June 13, 2020).
- [31] Jeffrey's image metadata viewer. <http://exif.regex.info/exif.cgi>, April 3, 2019 (accessed June 13, 2020).

- 
- [32] Foca github repository. <https://github.com/ElevenPaths/FOCA>, April 5, 2019 (accessed June 1, 2020).
- [33] Foca website - elevenpaths. <https://www.elevenpaths.com/es/labstools/foca-2/index.html#>, April 5, 2019 (accessed June 1, 2020).
- [34] Discussion in chromium forums about asynchronous methods not implemented or supported. <https://bugs.chromium.org/p/chromium/issues/detail?id=806283>, January, 2018 (accessed June 7, 2020).
- [35] Code point information. <http://tutorials.jenkov.com/unicode/index.html>, June 2, 2020 (accessed June 9, 2020).
- [36] Xmlhttprequest reference. <https://xhr.spec.whatwg.org/#synchronous-flag>, June 4, 2020 (accessed June 8, 2020).
- [37] Base64 to png converter. <https://onlinepngtools.com/convert-base64-to-png>, June 6, 2020 (accessed June 9, 2020).
- [38] Promise.prototype.then() - javascript | mdn. [https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Promise/then](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise/then), June 6, 2020 (accessed June 9, 2020).
- [39] Referenceerror: browser is not defined. <https://github.com/mdn/webextensions-examples/issues/194>, March 13, 2017 (accessed June 9, 2020).
- [40] Textdecoder api documentation. <https://developer.mozilla.org/en-US/docs/Web/API/TextDecoder>, March 20, 2020 (accessed June 9, 2020).
- [41] Fetch api - javascript | mdn. [https://developer.mozilla.org/es/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/es/docs/Web/API/Fetch_API), March 23, 2019 (accessed June 9, 2020).
- [42] Promise - javascript | mdn. [https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Promise](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise), March 23, 2019 (accessed June 9, 2020).
- [43] Using formdata objects. [https://developer.mozilla.org/es/docs/Web/Guide/Usando\\_Objeto\\_FormData](https://developer.mozilla.org/es/docs/Web/Guide/Usando_Objeto_FormData), March 23, 2019 (accessed June 9, 2020).
- [44] Windowbase64.atob() function information. <https://developer.mozilla.org/es/docs/Web/API/WindowBase64/atob>, March 23, 2019 (accessed June 9, 2020).
- [45] Xmlhttprequest - javascript | mdn. <https://developer.mozilla.org/es/docs/Web/API/XMLHttpRequest>, March 23, 2019 (accessed June 9, 2020).
- [46] Api for downloading files with other browsers. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/downloads/download>, October 25, 2019 (accessed June 9, 2020).
- [47] Murtha Baca. *Introduction to metadata*. Getty Publications, 2016.
- [48] Swapnil Bangare. The fetch api, a modern replacement for xmlhttprequest. <https://medium.com/beginners-guide-to-mobile-web-development/the-fetch-api-2c962591f5c>, March 26, 2018 (accessed June 9, 2020).

- 
- [49] Matt R. Cole. *Hands-On Microservices with C*. Packt Publishing Ltd, 2016.
- [50] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.
- [51] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.
- [52] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2019.
- [53] Jeffrey Pomerantz. *Metadata*. MIT Press, 2015.
- [54] Dinesh Rajput. *Mastering Spring Boot 2.0: Build modern, cloud-native, and distributed systems using Spring Boot*. Packt Publishing Ltd, 2018.
- [55] K Siva Prasad Reddy. *Beginning Spring Boot 2: Applications and Microservices with the Spring Framework*. Apress, 2017.
- [56] Chaitanya K Rudrabhatla. Comparison of event choreography and orchestration techniques in microservice architecture. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, 9(8):18–22, 2018.
- [57] Carlos Perez Sanchez and Pablo Solar Vilariño. *PHP Microservices*. Packt Publishing Ltd, 2017.
- [58] Sourabh Sharma. *Mastering Microservices with Java: Build Enterprise Microservices with Spring Boot 2.0, Spring Cloud, and Angular*. Packt Publishing Ltd, 2019.
- [59] Daniel Shiffman. Programming from a to z. <https://shiffman.net/a2z/chrome-ext/>.
- [60] F. Yergeau. Utf-8, a transformation format of iso 10646. <https://tools.ietf.org/html/rfc3629>, November, 2003 (accessed June 8, 2020).
- [61] Tarek Ziadé. *Python Microservices Development*. Packt Publishing Ltd, 2017.