

# LINKERO

A Brand Monitoring Platform



This report is submitted in partial fulfilment of the requirements  
for the  
B.Sc. (Honours) Information Systems and Information Technology (DT249)  
to the  
School of Computing  
College of Sciences & Health  
Dublin Institute of Technology

**Author:** Stefano Richiardi

**Student ID:** D12124118

**Supervisor:** Arthur Sloan

**Date:** January 2019



# Declaration

I Stefano Richiardi hereby confirm that the project I now submit for Honours Project Module titled *Project Linkero* is my own research work and was written by me following my research as cited therein. The work is new and has not been submitted for any previous award.

I also confirm that the slides I use/will use during the presentation that I made/will make are also my own work and are based on the content of my proposal.

I further confirm that the work has not been taken from the work of others save and to the extent that such work has been identified, cited and acknowledged within the text of my proposal and presentation.

Date

---

Stefano Richiardi

---



# Abstract

In this project I aim to build a data aggregator service for investigations and monitoring of counterfeit products online. The service will be implemented with a web interface for clients' access, an http server and specialized databases depending on data and performance requirements.

The core mission of this project is to try to simplify daily tasks of counterfeit investigations, and leverage server technologies to allow investigators to see the bigger picture painted when data from one case connect to one or multiple other cases previously unconnected.

Equally important will be the ability to deliver a product that is ready for production environment, that is: running on a virtual server accessible from the internet, protected from unauthorized intrusions and easily scalable should the user population grow significantly.

In this project I will also discuss pros and cons of data collection from online sources: from open API offered by online services to the community of developers, to unstructured data that can be scraped and schematized, and which legal implications need to be taken into account in light of the new European General Data Protection Regulation (aka GDPR).



# Acknowledgements

I would like to thank professor Arthur Sloan for his valuable feedback and invaluable encouragement throughout this project: receiving constructive directions when I was off track, and praise when I was on target made a huge difference.

I would also like to thank my wonderful partner Kasia for keeping our young family thriving, especially during the last sprint, when I needed maximum focus to complete the project, and my marvelous two-year-old daughter Clara for being part of the motivation that helped me to bring this experience to conclusion, although she is too young to know this.

A kind thank you to Claudio, Marianna and Alberto, friends and colleagues that were there to offer a shoulder to cry on, the times I felt I was getting nowhere, when I was trying every possible combination of a configuration file, and none would work, so much so that even the most skeptical person would concede machines too can be irrationally spiteful sometimes.

I am equally grateful to my boss, Kevin, with whose blessing I was able to learn and test at work (for work related tasks, I should clarify) some of the technologies required for this project. Also, it is his voice I hear in my head when wrestling with my non-native English, telling me: “clarify, simplify, summarize”.

Finally, I would like to thank those DIT lecturers that, between 2012 and 2016, really inspired me to take their teaching as a starting platform, an appetizer, and keep exploring on my own.





# Contents

<b>Declaration</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Project overview . . . . .	1
1.2 Background . . . . .	2
1.3 Objectives . . . . .	3
1.4 Scope . . . . .	3
1.5 Challenges and learning requirements . . . . .	3
1.6 Deliverables . . . . .	4
<b>2 Literature review</b>	<b>5</b>
2.1 Introduction . . . . .	5
2.2 Open Source Intelligence . . . . .	5
2.3 GDPR . . . . .	7
2.4 Server security standards . . . . .	10
2.4.1 Prevention . . . . .	12
2.4.2 Detection . . . . .	15
2.4.3 Recovery . . . . .	17
2.5 Design Engineering . . . . .	17
2.5.1 Component-level design . . . . .	18
2.5.2 User interface design . . . . .	19
2.5.3 Software architecture . . . . .	20

2.6	Existing brand monitoring services . . . . .	22
<b>3</b>	<b>System implementation</b>	<b>25</b>
3.1	Development challenge . . . . .	25
3.2	System architecture . . . . .	27
3.3	Technologies . . . . .	28
3.3.1	Server operating system . . . . .	28
3.3.2	Web application framework . . . . .	29
3.3.3	Databases . . . . .	30
3.3.4	Celery . . . . .	32
3.3.5	Email server . . . . .	32
3.3.6	Web server . . . . .	32
3.3.7	Firewall . . . . .	33
3.4	Development methodology . . . . .	34
3.5	Use cases . . . . .	35
3.6	Requirements . . . . .	38
3.6.1	Functional requirements . . . . .	38
3.6.2	Non-functional requirements . . . . .	39
3.6.3	User requirements . . . . .	39
3.7	Class diagram . . . . .	40
3.8	Sequence diagram . . . . .	43
3.9	Code samples . . . . .	45
3.10	Front end screenshots . . . . .	48
<b>4</b>	<b>Testing and deployment</b>	<b>51</b>
4.1	Testing strategy . . . . .	51
4.2	Unit testing . . . . .	52
4.3	Security testing . . . . .	55
4.4	Deployment . . . . .	57
<b>5</b>	<b>Evaluation</b>	<b>59</b>
5.1	Lessons learned . . . . .	59
5.2	Future work . . . . .	60
5.3	Conclusions . . . . .	60
	<b>Bibliography</b>	<b>61</b>

# List of Figures

3.1	Linkero system architecture . . . . .	27
3.2	Use cases diagram . . . . .	35
3.3	Class diagram . . . . .	42
3.4	Launch report sequence diagram . . . . .	44
3.5	Login page . . . . .	48
3.6	Home page . . . . .	48
3.7	eBay input modal form . . . . .	49
3.8	Case details modal . . . . .	49
3.9	Password change form . . . . .	50



# List of Tables

3.1	Launch new report use case . . . . .	36
3.2	Delete records use case . . . . .	36
3.3	Create user profile use case . . . . .	36
3.4	Download report use case . . . . .	37
3.5	Change password use case . . . . .	37
3.6	Search use case . . . . .	37
3.7	Delete user profile use case . . . . .	38



# Chapter 1

## Introduction

### 1.1 Project overview

The European Union Intellectual Property Office, in collaboration with the OECD, estimated that in 2013 the value of imported counterfeit good was €338 billion, which corresponded to 2.5% of the total imports in the world trade [OEC16]. The report also calculates that 5% of imported good in EU are counterfeit worth up to €85 billion. Counterfeit is an unfair commercial practice which takes advantage of the brand identity and presence on the market built by the brand owner, without incurring in the same costs of brand development (product design, quality standards and marketing, etc...). Other than a financial damage for the original brands, counterfeit products may pose safety risks for end consumers, as much as for the people that work to produce them, since they evade the strict quality and safety standards set by national and international agencies around the world. From counterfeit iPhone batteries that explode, to counterfeit air-bags that do not trigger, they all can cause physical damage to consumers.

E-commerce has been rapidly expanding since its early days in mid '90 when online marketplaces like eBay and Amazon were launched. Although it is hard to make a global estimate of market share for e-commerce companies, if we take the US market, the Census Bureau estimated that in 2009 online sales accounted for 4% of all retail sales, whereas in 2017 the e-commerce market share went up to 9% [oC18]. As legitimate retailers increase their presence online, shops selling counterfeit follow suit.

It is not surprising that many companies started to invest in fighting this trend, and that a good portion of the battleground is online. The tool developed for this project is designed to help users to extract data related to a specific brands from online platforms such as eBay, MercadoLibre, Allegro, and present it in a tabular format.

Online marketplaces offer API open to developers in order to enable automated interaction with their platforms. This tool will focus on extracting sales and business registration data. The data will be stored in a database so that users can keep historical records of all their queries. At the same time, users will be able to leverage the growing dataset to link new investigations to old cases across all platforms whose data has already been stored in the database. For instance, searching the email address of a shop that deals counterfeit Diesel jeans may return details of multiple businesses registered on eBay and MercadoLibre at different times, as well as information about the administrator of a Facebook page about counterfeit Ray-ban glasses. Users can use this tool to estimate brand damage caused based on the volume of sales, as well as a forensics platform to facilitate identity attribution of potential counterfeiters.

## 1.2 Background

Identifying online counterfeit items starts with researches of a brand or product online presence. The aim is to identify sellers that offer a branded product sold at a price point below average retail price. An investigator would ideally prioritize sellers with higher business turnover, and ideally located in jurisdictions where legal action is a dependable and impartial option. For instance countries like Russia and China, to name few, do not always offer adequate protection for European and North American companies, making it more difficult to pursue compensation from actors located withing their borders.

There are already different online companies that offer brand monitoring services: they range from keyword web-crawlers, to consumer sentiment analytic based on social platforms, to anti-counterfeit detection. I will review those companies and their services in details in the next chapter.

GDPR is the latest European Union legislation about processing of personal identifiable, privacy protected data belonging to European citizens. GDPR affects the implementation of this project in two ways: first, and most obvious, because users will submit limited personal data in order to be able to access Linkero services. Something as simple as an email address required to get notification for password recovery or data collection completed confirmation, is considered a PII (personal identifiable information) and thus is protected under GDPR. But GDPR affects also the collection of public personal details available online. This is the compliance requirement that is specific to companies providing *web scraping* services. When we deal with open source intelligence and attribution, PII are the most valuable data for anti-counterfeit investigations, therefore I will talk about how GDPR affects services like Linkero and what



are the requirements to guarantee compliance.

### 1.3 Objectives

The objectives set for this project fall in two main categories: business and personal.

In relation to the first type, the goal is to build a business-ready web platform for brand-monitoring investigations. Being *business ready* means presenting a final application that runs on a virtual server directly connected to the public internet; that implements all industry standard security features to guarantee that only authorized users can access its features and data; and is scalable, designed to accommodate an exponentially growing number of users. The other aspect of the first goal refers to the ability to provide basic case management, data extraction and keyword searches capabilities tailored for brand monitoring and anti-counterfeit investigations.

On a personal level, I aim at gaining a hands-on understanding of specific web technologies (e.g. JQuery, Django framework, NGINX, uWsgi), NoSQL databases (MongoDB, Redis), asynchronous and non-blocking programming techniques (multi-threading, AJAX, Python Celery), development methodologies and testing practices.

### 1.4 Scope

This project will consist of a ready to deploy and use system for case management and web scraping data using eBay public API.

Regular users will be able to login, change password, set their preferred email address for general notifications and report delivery, launch queries, download data from completed reports, delete old queries, and perform keyword searches within all data collected and stored by any user in the database. Also the system will be configured to provide security protection against external unauthorized access and use of the system. Site administrators will manage the registration, password reset and deletion of users profiles.

### 1.5 Challenges and learning requirements

There are technical and personal challenges that have some kind of impact on the delivery of this project. To start with I am new to most of the technology used: from the basic configuration and administration of a Unix server, NoSql databases, client side web technology, SSL and DKIM protocols, etc.... This means that part of the time dedicated to the project will be spent bringing my knowledge and familiarity to

a level that suits the project's requirements. Another element of technical challenge is determined by the use of a virtual server with only 1GB of memory and 1 single CPU. These limits are mainly dictated by financial reasons: this is the cheapest available VPS, which comes at \$5.00 USD a month. Considering that I needed a VPS for the entire development and testing of the project, which took almost 2 years, anything more expensive would have impacted my family finances. The other problem is that limited memory and processing power started to show their impact and the code had to be optimized accordingly.

On a personal level I will have to take into account all the extra pressure that comes from being a father in a young family and the professional responsibilities of a full time job.

## 1.6 Deliverables

The completed project will include:

- full documentation of the system design, development and functions;
- a working implementation of the system;
- a PowerPoint presentation of the project.

## Chapter 2

# Literature review

### 2.1 Introduction

This chapter will provide an overview of different topics that informed design decisions made during the initial stages of the project. I will start looking at the discipline of Open Source Intelligence, which refers to all those protocols and techniques used by government and private agencies to piece together intelligence reports using publicly available sources, since Linkero is a tool that facilitates the structured collection and analysis of a selected portion of open source data. The legal aspect of data collection and analysis will be explored with a reasoned summary of what is GDPR and how it impacts similar online services. Three sections will be dedicated to industry level best practices in relation to the security of servers facing the public internet, the usability of software interfaces and current guidelines to build reusable, maintainable and expandable code. Finally I will review briefly what are some of the current services already offering brand monitoring tools and how they differ from one another.

### 2.2 Open Source Intelligence

Open Source Intelligence (more commonly referred to as OSINT) is a relatively young discipline, that is concerned with the art of piecing together strategic intelligence from public sources of information. Michael Bazzel, a leading OSINT expert, defines it as:

any intelligence produced from publicly available information that is collected, exploited, and disseminated in a timely manner to an appropriate audience for the purpose of addressing a specific intelligence requirement. [...] For the CIA, it may mean information obtained from foreign news broadcasts. For an attorney, it may mean data obtained from official gov-

ernment documents that are available to the public. For most people, it is publicly available content obtained from the internet [Baz15].

As Bezzell explains, OSINT is not necessarily based on online sources, at least in its most broad definition. Journalistic style, old fashion dossiers filled with newspaper clippings are a form of OSINT. However it is fair to say that whenever OSINT is mentioned today, it will automatically produce the expectation that a large proportion of content is sourced through the internet. Another important author in this field, Stewart K. Bertram, explains:

older OSINT research was limited by both the coverage of its information and the ability of the researcher to focus the capability on a specific subject, be it a person, location or topic. [...] What has changed this status-quo is the arrival of the Internet, and particularly the explosion in the use of social media technology circa 2000. The rise of these two technologies created a multilingual, geographically distributed, completely unregulated publishing platform to which any user could also become an author and a publisher. [...] By increasing the coverage and focus of OSINT the Internet effectively promoted OSINT from a supporting role to finally sit alongside other more clandestine and less accessible investigative capabilities [Ber15].

This explosion of sources of information causes another problem: reliability. Unless we are sourcing information from a scientific magazine which follows the rigorous fact checking protocol of most scientific researches, then we are facing a vast landscape of information with various degrees of veracity: reliable fact-checked sources on one side and *fake news* at the other end of the spectrum. The work of the OSINT investigator is to move in this virtually infinite universe of news, pick only relevant information and establish how reliable they are. Again this is not new, it is called *intelligence analysis*: “[...] the application of individual and collective cognitive methods to weigh data and test hypothesis within a secret socio-cultural context” [Hay07]. Information are scored from A (reliable) down to E (unreliable), plus F (reliability unknown).

Within the domain of counterfeit investigations, OSINT can have a number of roles to play. First off and foremost, should be used to establish if the items being sold are genuine or counterfeits. Secondly, it can be used to establish the extent of profits made (and therefore loss of income for the original brand) by the merchant selling them. And finally, OSINT can provide vital help in identifying the identity of the merchant as well as that of the manufacturer.

## 2.3 GDPR

The General Data Protection Regulation (EU) 2016/679 (a.k.a. GDPR) is the latest legislative effort of the European Union, that regulates the collection and use of European citizens' data. The two main goals that guided this new legislation were already set in 2015 by the Council of the European Union, which stated:

The twofold aim of the Regulation is to enhance data protection rights of individuals and to improve business opportunities by facilitating the free flow of personal data in the digital single market [otC15].

Essentially the GDPR gives more control to European citizens over the use of their data, thus restricting the ability of private and public organizations to process Personally Identifiable Information (a.k.a. PII) data, and at the same time it becomes a legislative framework that standardizes personal data processing in every country member of the European Union [oECtECohRE18]. Let's see in more detail what this entails and what are the consequences for a product like Linkero.

First off it is important to clarify some of the terminology used in the legislation. *Personally identifiable information*, or PII, is defined in privacy laws as one single piece of data that alone can identify a single person. For instance: the first and last name of a person, an email address, a physical address, a phone number, passport serial number, credit card number, a picture portraying an individual, a combination of username and password. GDPR applies to the collection, processing and storage of PII. A *Data Controller* is any organization that collects and manages PII belonging to European citizens. That includes online private companies (e.g. Google, Facebook the most obvious), as well as public institutions like primary and secondary schools and hospitals to name few. Another party that is involved in the handling of private data is the *Data Processor*, which in some cases coincides with the the Data Controller itself, when data is analyzed internally, or with any other organization (e.g. contractors, vendors, suppliers, etc.) that receives PII from the Data Controller and processes it in its behalf. Finally the *Data Protection Officer* is a public office that is appointed to overlook and audit every Data Controller in its area of control [CR17].

What GDPR means for European citizens is that as of May 25<sup>th</sup> 2018, when it officially came into force, it grants much more rights over their own personal data, compared to previous legislation. These rights include *data portability*, which is to say that any Data Controller has to structure the collection and storage of PII in such a way that those information can be removed, transferred and re-allocated to another Data Controller without any further modification. In other words, this is a way to guarantee interoperability between Data Controllers, preventing single Controllers to

lock-in their users. GDPR also states that EU citizens have the *right to information and transparency*, so that at any moment they can request their PII to be disclosed as they have been collected up to that point in time. Another right that generated a lot of discussion, is the *right to be forgotten*, which states that users can have their information permanently erased from a Data Controller, if they no longer require their services [oECTECohRE18].

GDPR also sets more stringent constraints to the operation of organizations that fall within its scope. GDPR is self-declared *transnational in scope*, meaning that it applies to every company, anywhere in the world, as long as they hold EU citizens' PII data. Obviously, enforcement is an issue in cases such as companies that are based somewhere in the Cayman Islands. However this aspect is mainly meant at large multinational corporations, that could simply transfer their databases to another jurisdiction outside the EU in an attempt to circumvent its reach. Under the current definition of GDPR, those multinationals would still be subjected to EU law and would still be accountable as long as they have legal presence in any EU member state, regardless of where the PII data actually sits. GDPR also establishes a stronger *Data Subject consent*. This means that Data Controllers can no longer assume that their users' data can be processed and used for whatever purpose, just because PII were given to them during the registration process. Nor they can pre-tick the data consent acknowledgment box, or ask the user to tick a box only if they do not consent to the treatment of their data. On the contrary, Data Controllers have to explicitly ask and receive users' consent, and consent has to be renewed over time. Finally, GDPR requires organization to *report data breaches* within 72 hours after the breach was discovered. Data Controllers not only have to notify the supervisory authority and their users, but the notification has to carry adequate details to explain how the incident occurred, how many users are affected, and how is the organization going to respond [CR17].

With all these new rights and obligations in place, the last part of the legislation concerns to the *sanctions* for organizations that fail to comply. There are three levels of penalties that can be imposed:

1. a warning letter for the first time an organization is found in breach of GDPR, and if the breach is non-intentional;
2. a more serious breach could result in the order to commit to regular periodic data protection audits;
3. and finally, if the breach is deemed to be serious enough, GDPR prescribes two sets of financial fines depending of which obligations were missed by the Data Controller, the harsher of which could amount to up to €20 millions or up to

4% of the annual turnover estimated from the prior financial year, depending on which is higher [oECtECohRE18].

The question now is: how does GDPR affect online services that for their nature collect and process bits of PII that are publicly available? First and foremost, GDPR applies only to PII belonging to EU citizens, therefore we can say that an online service that collects PII can keep on providing its functions without any further thought as long as it stays clear of EU citizens. But let's say some users will be pointing the service to PII of European users, will that be sufficient to be found in breach of GDPR? The answer to this latter question is not a clear-cut *no*, there are actually use-cases where collection can still be an option. Obviously those cases do not include Data Subjects who expressed consent to their PII to be collected and stored by anyone online, that is not how data scraping for intelligence purpose works, especially in the context of open source investigations. Although it is too early for similar cases to have been tried, GDPR accepts that organizations may have a *legitimate interest* in scraping, storing and processing publicly available PII. The protection of intellectual property is clearly mentioned as one of the rights that fall within the boundaries of legitimate interest [oECtECohRE18]. In the specific context in which Linkero is designed, we are talking about an Intellectual Property owner, or an agent in their behalf, that has a vested interest in gathering information about an online shop that sells their products, under market value, or without being an officially approved re-seller. Having said that, GDPR definitely limits and restricts, for better or worse, the ability to extract, store and process public PII.

To conclude this section, it is interesting to mention how ensuring adequate privacy protection is a difficult balance and has its own trade-offs. This is the case of the WHOIS protocol maintained by ICANN (i.e. Internet Corporation for Assigned Names and Numbers), where internet domain registrants' PII are kept and made public. With the introduction of GDPR, personal details of the person who registered a domain will have to be hidden. Details of the domain registrant are valuable information in order to identify the person, or the signature of the person or group behind specific internet domains. This is particularly useful when security researchers try to attribute the ownership of phishing or spamming domains to a particular actor or group. Likewise, in counterfeit investigations, being able to tell how many and which domains were registered by the same actor is particularly useful, especially if you are trying to monitor the actor's activities. Information do not need to be accurate, in fact domain registrars do not verify the real identity of the person registering a domain. There are however details that need to be valid, like contact details (e.g. email address, phone number, ...) because on them depends the correct functioning of the

domain, which is in the registrant's interest. The disappearance of those PII from WHOIS caused a great deal of frustration in the security community, and ended up with ICANN filing an official litigation against the European Union asking for some sort of exemption in order to keep offering relevant WHOIS data.

GDPR was created and voted by EU member states in a period when information is being considered the new oil of the global economy: access to personal data and consumer behaviour generates huge amounts of revenues for corporations like Facebook and Google. And the reason for that is simple: targeted ads campaigns are extremely effective and efficient, and marketing departments are willing to pay for that. The commercial application of petabyte of consumer data is well known. But there is another aspect to an unregulated access to data belonging to billions of people, which is *political influence*. As we have seen during the 2016 US election, bad actors managed to use personal data of millions of US citizens to both target specifically voters in swing states, and to analyze general voters' opinion in order to design very effective slogans. The same actors fabricated made up stories—literally, fake news—that would appeal to the most visceral human fears in order to influence undecided voters. That happened also during the British referendum of Brexit, and was attempted during the French presidential elections in 2017, and very likely in many other countries that did not receive as much media attention. At the same time we saw government bodies abusing their ability to tap into and collect indiscriminately data, just as Edward Snowden exposed how the US National Security Agency. GDPR may not be the best solution yet, but it is a much needed regulation at a time when personal information can have serious impact on the society.

## 2.4 Server security standards

A chain is as strong as its weakest link, and a system is as secure as its most vulnerable element, as the saying goes. In today's reality, any system facing the open and public internet has to be secured against people trying to misuse it. It is not just big banks and multinational corporations that have to put in place sci-fi grade defenses, but the same private wifi routers and connected thermostats have to be protected against bad actors that would gladly take them over and deploy them to perform fraudulent or criminal activities. It is well known that professional computer criminals constantly scan the internet to detect new connected devices and try to add them to their personal collection of bots for different purposes. Internet security is therefore a must regardless of the complexity of the system. Information Security literature identifies three main goals of a good system security strategy, also referred to as the *CIA triad* [Pipool]:



Confidentiality, Integrity and Availability. That means that a system security strategy aims at guaranteeing that data stored in the system will be accessed only by legitimate and relevant parties (confidentiality), that the same data will be kept in its original state away from sources that may alter, intercept or erase it (integrity), and that despite of the safety measures in place, if an attack succeeds, the data will still be accessible, and business will continue with minimal to no disruption (availability). In addition to the CIA triad, some experts also include *accountability*, which essentially means that if an incident occurs, the remediation team should be able to identify exactly where the breach happened and the location if not the identity of the person who executed the attack.

The industry identifies two main types of threat: internal and external. The external threat is probably the first and most popular to come to mind, often depicted like the hooded boy wearing a Guy Fawkes mask and typing away on his laptop: the hacker. Put simply, it is a person or organization that attacks a system from the outside. The other threat that does not get as much attention, but can be equally damaging, is the internal threat. The internal threat is represented by an internal user that intentionally or by mistake causes damage to the system. An attacker can be further classified based on his/her level of activity: a *passive* attacker tend to sit and intercept data exchanges, without affecting the system resources, whereas an *active* attacker directly alters system resources while the operation is underway [WS15].

Any attack normally impacts one or more of system assets: hardware, software, data or networks. There are five types of security measures that an organization can put in place to protect its assets: deterrent, prevention, detection, response and recovery. Deterrent measures are meant to scare attackers off before they even consider to start their exploits. The criminal law with its system of penalties is definitely the first deterrent that comes to mind. But media articles detailing how sophisticated and complex is a detection system can offer the same purpose. Preventive measures on the other end are technologies that anticipate potential attacks and stop them before any damage occurs. Firewalls are a great example. Since absolute prevention is rarely feasible, detection technologies can cover those situation in which an attack is found while still ongoing. Detection immediately triggers response measures, such as blacklisting an IP associated with a number of failed login attempts. Finally, recovery measures are the ones that are deployed after an attack occurred, and try to assess the damage and remediate where possible [Lam04].

Information security is a very vast and complex discipline, which directly reflects the complexity of modern interconnected systems. For the purpose of this project however I will only consider the basic security configuration required to protect a standalone Linux server.

In the next part of this section I will review in greater details some of the prevention technologies and solutions available, particularly those that were considered for this project, focusing on prevention, detection and recovery measures.

### **2.4.1 Prevention**

#### **Regular software updates**

Applications running on the system are rarely bug-free. It is always good practice to keep every system application up-to-date with latest patches, since every time a new vulnerability is discovered, application developers tend to issue a new version or patch to close it [WS15]. This system is set to look for security updates issued by the Linux distribution Debian, every 24 hours.

#### **Environment isolation**

Rather than a technology, *isolation* is a principle and a policy that recommends every resource should be isolated from the rest, and access between them should be regulated and monitored. This is particularly critical in the context of systems that have public interfaces. Isolation applies to file management as well, requiring users files and folders to be isolated from one another, preventing one user's compromised credentials to be used to access everyone else's files. Finally, isolation applies to the same security mechanisms, since they should be isolated from the rest of the system, so that they cannot be compromised by the attacker [WS15]. For this project, isolation was implemented by giving each application (e.g. web server, email server, databases, etc...) its individual user profile and access, so that if one is compromised, the intruder will still be limited to the set of tasks granted to that application.

#### **Virtual Private Networks**

VPN are a good solution to connect parts of the system that are not located on the same machine, or in the same network. Since communication between resources has to go through the public network, VPN offer a way to encrypt and protect data exchange based on asymmetric encryption technology [WS15]. Currently there are no VPN deployed in this project, since every element is installed on the same single server. However VPN would be the ideal solution, if the system grows and specific applications, like databases, would have their dedicated machines.

### SSL and TLS encryption

Similarly to VPN, SSL and TLS are two technologies that create an encrypted communication channel between a client and a server. The most common use case for these solutions is when login forms are required, so that a client can transfer login credentials to the server using a HTTPS connection, thus avoiding to transfer username and password in clear over a non-secure public network [WS15]. For this project, an SSL certificate was issued for free by *Let's encrypt*<sup>TM</sup>(<https://letsencrypt.org>).

### SSH

The Secure Shell, SSH, is the default application for remote server management. Like VPN and SSL, it establishes an encrypted connection between a client and a server, relying on asymmetric encryption. It allows system administrators to login remotely to the server command line shell. It can be configured to either accept a standard password login, or to accept automatic logins from approved clients only [WS15]. Every admin interaction on this system happens through SSH tunnel.

### Firewalls

Firewalls can be described as *intrusion prevention systems*. A firewall is an application that filters all incoming and outgoing network traffic between the internal network and the public internet, and responds to a set of rules to decide what is allowed and what is not. It provides a single choke point for network traffic, where the system can act as detection mechanism for suspicious connections, as well as traffic logger for audits. There are four different types and corresponding level of complexity: packet filtering, stateful inspection, application-level gateway and circuit-level gateway. The last two are the most sophisticated and require also a lot of CPU and memory resources, for this reasons are not suitable for this project [Cheo3].

Packet filtering is the most basic capability of a firewall, and bases its decision on whether to allow or reject a connection on the information contained in an IP packet: source and destination IP, source and destination transport-level address, IP protocol and interface. Packet filter firewalls however are incapable of establishing when a connection to a port number above 1024 is legitimate or not. This is where a stateful inspection firewall comes into play. Stateful packet inspection works by keeping track of ongoing connections, session numbers, and originating party in order to prevent attacks like session hijacking [WS15]. I will discuss the details of the system firewall in the next chapter.

**Users and groups least privileges**

Just like environment isolation, assigning users and groups with the right amount of privileges and permission is more a policy than a technology. This policy states that users should be given the minimum amount necessary of privileges in order to perform their tasks. This way if an account is compromised, there is a chance that damages can be controlled and limited. Permissions include the ability to read, write, execute, delete or share a specific resource. More advanced resource and permission management policies include a temporary aspect, in that permissions are granted for a limited amount of time, and are active only during working hours, or whenever a specific task needs to be performed [WS15].

**Password policy**

A password policy is design to make sure that all users comply with the industry best known standards of password security. A password policy should establish that:

- the strength of the password by insuring and checking that a user generated password uses the most varied combination of lower and upper case letters, numbers and special symbols in a string of a minimum length of eight characters, in order to guarantee that a brute-force attack (attempting to guess the password trying every possible combination) would require a significant amount of time to succeed;
- the lifetime of a password, which should be long enough not to cause user friction, but short enough to guarantee that should the password be compromised, there is a chance it will be too late to use it before expiration;
- a new password cannot be the same as any previously used;
- a set of alternative contact details and security questions that should be used to recover a lost or compromised password;
- a maximum numbers of failed login attempts, after which the account is locked and a password reset is required;
- recommendations about the right way of storing the password: making sure the password is not written down or saved in an unsecured file.

**Encryption of data at rest**

Systems that store data in databases may be compromised, when that happens if the data is stored in clear, the attacker can make a copy of the database, and access all the

contents. Data that is not loaded in memory can be encrypted, so that simply making a copy and moving it outside of the current system, will not allow the attacker to access it, lacking the necessary decryption keys [WS15].

### Physical security

Physical and infrastructure security is sometimes an overlooked aspect, when the focus is all on hardening the software and data, when it is actually as important. There are three main categories of threats to physical infrastructure: environmental, technical and human caused. Environmental threats includes both exceptional events like tornadoes, floods or earthquakes, as well as less traumatic events like the level of humidity, dust, external temperature, gas and other chemical elements, factors that in the long run can actually cause serious damage. Technical threats include power surges or drops, and electromagnetic interference. Finally the most common human-caused threats are: unauthorized physical access, vandalism, theft and misuse.

For the purpose of this project however, no countermeasures were taken to mitigate those threats, since the system is composed of a single virtual server, run by the Virtual Private Server provider Linode Ltd. I assume that safety and security measures are being adopted by the service provider [WS15].

### 2.4.2 Detection

#### Intrusion detection systems

Intrusion detection system are the second level of defence, in case all of the previous preventive measure fail to stop an external intruder. The basic idea behind an intrusion detection system (IDS) is that the way an intruder interacts with the system is substantially different from any other authorized user, thus his/her behaviours can be described in quantitative terms (the intruder's signature) and this description can then be implemented in a system whose purpose is to ring an alarm when such behaviour is detected and block the intruder before any damage occurs [WS15].

These are the main types of IDS currently available:

**Host-based IDS** : application that runs on a specific host, and only monitors activity on that host.

**Network-based IDS** : application that runs on a host part of a larger network, that monitors all the activities within that network.

**Distributed or hybrid IDS** : application that gathers data from a cluster of sensors distributed across a network [Shioo].

There are essentially two ways that an IDS can be programmed to detect intruders: we can either decide that a range of behaviours is legitimate, therefore everything else that deviate from that norm is an intruder, or we can list bad behaviours, and tell the IDS to trigger the alarm when any of the behaviours listed is detected. The first strategy is called *anomaly detection* the latter *signature detection*. Without going in too many details, anomaly detection is clearly the most effective of the two, and would be anyone's first choice if not for the fact that is very difficult to define a norm. So difficult that it requires very sophisticate approaches like statistical analysis and machine learning techniques [GT09]. Signature detection on the other hand is a bit easier to implement, and guarantees a lower number of false positives, given that its database is stored with good quality intruders' signatures. The problem is that any new, unexpected and undefined intrusion behaviour will not be detected, leaving the system vulnerable to zero-day attacks [WS15].

### **Regular file, service and vulnerability audits**

A security auditing system is a very complex set of hardware and software tools, policy, processes and old fashion human labour. Such a system is beyond the scope of this project, but it is worth briefly mentioning it, for completeness sake, within the discussion on information security.

A well designed security auditing system has the following requirements:

- being able to establish at any time whether a system's security is intact or has been breached;
- capture by default all data that is necessary to establish if an intrusion occurred, when, how and possibly by who;
- store relevant data for future analysis and forensics;
- feed anomaly signatures back to an intrusion detection system signature database [WS15].

An audit can be triggered right after a potential attack is detected as well as can be repeated periodically. An audit can be triggered in different ways:

**basic alerting** : is the simplest for of trigger, which only requires a single rule to trigger an alarm if a specific event occurs.

**Baselining** : is the practice of defining a normal versus unusual event or pattern. This feature overlaps with an IDS signature-based trigger.

**Correlation** : is taken from statistical analysis and consists on triggering an alarm or notification when an event or set of events is used to infer the presence of a potential breach [WS15].

### 2.4.3 Recovery

#### Business continuity plan

A business continuity plan is a set of measures, mechanisms that ensure an organization can go on or quickly restore basic and critical operations with minimal downtime [Woo10]. In a business continuity plan a series of *disaster* scenarios are identified, for instance: loss of workspace, loss of personnel, loss of network infrastructure. Every scenario is coupled with a list of function critical resources that can be impacted and relative countermeasures to be executed along with roles and responsibilities assigned to specific people. Having a backup of all the data on the system is a preventive measure, using it to restore data that was lost, encrypted or corrupted during an attack is what recovery looks like.

## 2.5 Design Engineering

Goal of a good software project is to produce software that not only works, but that is easy to understand, to test, maintain and expand if needed. Hewlett-Packard developed five attributes [Gra87] that define the quality of a software product:

**Functionality** concerns the set of functions and capabilities of the program, as well as its security.

**Usability** relates to how well the intended users are able to operate the program.

**Reliability** is the measure of frequency and severity of the program failures.

**Performance** indicates the program speed of response, resources consumption, efficiency and throughput.

**Maintainability** is concerned with how the software is easy to troubleshoot, adapt, configure and extend.

Design engineering takes care of the quality of a software, and is the last stage of software modelling before code development can actually start [Pre05]. Design engineering is divided in four elements: data design, component-level design, interface design and architectural design. I will briefly explore each one here.

### 2.5.1 Component-level design

Component design refers to the procedural aspect of a software program: functions, processes and algorithms are the object of this element of design. Design engineering developed over the years a set of rules and guidelines that allow developers to write code that is maintainable and extendable:

**Modularity** states that the final software should be the result of separate nuclear components, called modules, that will be integrated and interact with each other in order to produce the desired functionalities [Mey78].

**Information hiding** suggests that, as much as possible, the data used within one module for its operations, should be hidden from other modules. Only the bare minimum amount of information should be exchanged between modules to ensure software operations [Par72].

**Functional independence** is a concept directly derived by the previous two, and it says that a module should focus on a specific functionality (measured in level of cohesion) and it should have as little interaction with other modules as possible (expressed in terms of coupling) [Par72].

**Refinement** rather than a prescription this is a top-down process of defining each element of the software starting from a high level of abstraction, and progressively narrowing down to more detailed procedural levels [Wir71].

**Refactoring** is another technique that aims at simplifying the design of a program without changing its functionality. The code is examined for unused elements, redundancy, inefficient or unnecessary algorithms [Pre05].

Robert Martin took those rules and techniques and elaborated further into his five S.O.L.I.D. principles [Mar03]:

**Single responsibility** states that each module or class should be responsible for one function, and that function should be well encapsulated within the module or class. This principle combines the concept of cohesion and information hiding.

**Open-Closed** dictates that a module or a class should be built in such a way to prevent modification, but still offer an interface that allows it to be inherited and expanded. The reason is that, assuming a new requirement arise calling for a new functionality to be implemented, adding that functionality to an established class could alter its behaviour and cause unintended consequences in other parts of the code where that class is in use. Open-closed principle suggests instead



a new subclass should be created in order to fulfill the new requirement, thus guaranteeing the stability of an application, while allowing for functionality extension.

**Liskov substitution** is an extension of the *open-closed* principle. It requires that any child or grand-child class can take its parent's class place in the application without altering the general functionality of the application. This principle is intended to guarantee consistency of interface within class types of the same family.

**Interface segregation** recommends to develop simple client-specific interfaces, rather than one general purpose interface for every known client. One general purpose interface would include all functionalities that are relevant to any client, thus forcing every client to implement functionalities that are not necessarily relevant. The other problem is that if a new requirement calls for a new interface to be added, all pre-existing clients should update their implementation to fit the new interface, whether they support it or not. To solve this issue, single functionalities should have their own interface, and clients can implement selectively just those that are relevant.

**Dependency inversion** finally integrates the concept of module de-coupling, stating that higher level modules should not depend on lower level ones. The problem with modules dependencies is that a change in one could force changes to its dependees, the assembly of modules could result in expensive operations to guarantee that every dependency is respected. This goes against module re-usability. The solution is to add a standard interface between two modules. A good example to explain this principle is imagining the relation between a lamp and the home electrical supply: making them directly dependent equates to soldering the lamp wires to the electrical system, instead the electrical socket was developed and became the standard interface between any electrical tool and the electrical power supply [Pre05].

### 2.5.2 User interface design

User interface design is concerned with making sure the user will be able to actually use the software, by designing an interface that is meaningful and intuitive. Theo Mandel developed three *golden rules* [Man97]:

**Place the user in control** means that the interface has to be designed with the user's needs in mind, not the developer. And these are some of the most common needs

from a user perspective: a user needs to be able to perform the same action in different ways (e.g. via console commands, mouse action, keyboard short cuts, etc.); a user sometime needs to suspend a task and move to a new one, before resuming the old one, as well as being able to *undo* an action made by mistake; a user needs adjust its interaction according to his/her level of expertise (e.g. initial point and click can be replaced by macro recording of a sequence of actions); a user does not need to see nor interact with low level functions; a user needs to manipulate objects of a visual interface as he/she would do if those objects were a physical thing.

**Reduce the user's memory load** requires for interfaces to be intuitive, without much study a user should be able to know what each part does just by looking at it. Mandel [Man97] further specify this rule, by saying that the visual layout should be modelled based on a real-wold metaphor using well established visual cues; the initial default configuration should make sense for the average user, allowing the expert user to change it to his/her liking; information should be presented in a progressive way, starting from general task, down to more detailed functions.

**Make interface consistent** demands that a standard layout should be defined and kept consistent throughout the rest of the program. Context layers are very important, allowing the user to know at a glance where his/her current task lays within the general process (e.g. window tiles, graphical icons, color coding, etc.), how the user was able to access the current interface and what are the available next steps. Interface consistency also requires that design takes interactive sequences that have become standard (e.g. *Cntrl-C* is the standard for copying pieces of text) and makes them available in the application.

### 2.5.3 Software architecture

Software architecture encompasses two elements of software design: data design and architectural design. Here I will touch on the main concepts that are used to describe data and architectural design.

#### Data design

Just like this project, most recent software applications and services have been developed around the idea that accessing, storing and computing large amounts of data can offer innovative solutions. Which is a way of saying that data today is the new black gold, it is not a surprise than that designing what kind of data will be analyzed and how it will be stored and structured is extremely crucial [Pre05].

At architectural level, data design is mainly concerned with the structures that will contain the data and how those integrate within the overall architecture. We are talking about using external databases to store the business data. Depending on the size and complexity of data requirements, we may see a small amount of centralized databases that store production data that is needed to provide live core business services. With bigger corporations and a growing number of diversity of database structures, it may become more difficult for data analysts to access information that is spread across multiple databases with multiple diverse data structures. *Data warehouses* come to the rescue. Data warehouses essentially store copy of production business data in a separate environment, and make sure that all the data is kept using the same structure, thus allowing data analysts to perform data mining on current and historical data, in order to offer strategic insight for business executive decisions [Mat96].

### Architectural design

Architectural design aims at building a blueprint of the entire system, with special emphasis of how each component interacts with the others. There are two aspects that describe an architecture: style and pattern [Pre05].

Style refers to the shape of the overall system structure, it is a static representation of the system: which components interface directly to each other, which are peripheral, which are central, and so on. Examples of architectural styles include:

**Data-centered** architecture refers to a system that is designed around its database, where all other components are clients that exchange information only with the central server, with little or no interaction with each other.

**Data-flow** architecture instead is applied to a system where data moves mainly in one direction, first as input, leaving the system as output, after all the modules located in-between these two points have modified somehow the original input.

**Call and return** is a style in which modularity is central to the overall operation, where each function is broken down in a hierarchy of modules. This style has the advantage of being relatively easy to modify and scale.

**Layered** architecture, finally, describes a system in which components are grouped from an outer layer, normally the user interface, through an application and utility layer, down to the core layer which performs machine levels operations.

These are just a sample of the most common architectural styles [Pre05].

An architectural pattern instead describes the most dynamic aspect of the system: how each component interacts with the others, how they behave. Architectural patterns are also referred to as a solution to a standard software development problem. These are some examples of patterns and the problem they solve:

**Concurrency** is the ability to perform multiple, different and independent tasks at the same time. One problem developer face is when the system has limited resources, like one single CPU, so that real multitasking cannot be achieved. An *operating system process management* pattern provides that concurrency can be simulated by assigning a turn to each task for a limited amount of runtime. This solution offers a real advantage for tasks that depend on many read and write operations with external slow components.

**Persistence** is the ability to store data, after the process that created it is finished. *Application level persistence* pattern for instance provides that applications show generate and manage their own file format where to store user-generated data or software configurations. Another pattern that offers a solution is the *database management system*.

**Distribution** addresses the issue of components communication in a distributed environment, like clients exchanging emails: each client cannot assume that the recipient will be online in order to receive the communication. The *broker* pattern's solution consists in building an agent to delegate to a communication, and the agent will guarantee the message delivery by waiting for the recipient to be ready to receive the message [Bos00].

## 2.6 Existing brand monitoring services

There are several online companies offering what they call *brand monitoring* services. This type of service has two major definitions:

- on the one end, brand monitoring refers to activities whose purpose is to analyze the online presence of a brand to gain insight about its the market reception and placement [Tec18].
- on the other end, it refers to activities whose aim is to protect the intellectual property of a brand by collecting information on online activities that may infringe it [Bra18].

Of the first category there are different platforms offering marketing analytics, promising to let you know what people think instantly: Brands Eye<sup>TM1</sup>, Brandwatch<sup>TM2</sup>, among the most popular. These services tend to monitor mostly social media platforms, where consumer sentiment can be gauged. On the other end of the spectrum Watchdog<sup>TM3</sup> is an example of platform that is much closer to this project idea, whose tag line summarizes its purpose: “Watchdog is an online surveillance tool designed from the ground up by investigators”[Wat18]. They enable searches across the major 50 e-commerce platforms, including Amazon and eBay. Unfortunately it is hard to get a better grasp of the mechanics of the platform, or even the UI, since the service is offered on subscription only.

---

<sup>1</sup><https://www.brandseye.com>

<sup>2</sup><https://www.brandwatch.com>

<sup>3</sup><https://watchdogapp.com>



## Chapter 3

# System implementation

In this chapter I will discuss the actual implementation of the original idea behind Linkero: a brand monitoring case management application. I will start from the architecture in order to show a general snapshot of the system, before talking in details about every single component, the technology that was chosen and reasons behind. I will follow with use cases and system requirements, to conclude with class diagram, database structure and some code samples.

### 3.1 Development challenge

The system architecture described in the next session *System architecture* needs a little explanation, because it deviates from the standard three-tier architecture of many websites.

Before this project I already developed a stand-alone, single thread, application that would generate reports of eBay items and sellers. The application was written in Python and included a simple user interface, corresponding to what is now the eBay input form (see figure 3.7). The application would take the user input, retrieve the data using eBay public API and would create and save in the local hard-drive a csv file. It was all that was needed, and worked well for one single user that needed access only to eBay data.

More users and new features immediately showed the limits of the stand-alone application approach:

- the installation process was not straight forward for non-technical users, since a Python version needed to be installed along with some other non-standard libraries and dependencies;

- some sort of version control system needed to be established and maintained in order to guarantee that all users would run the latest version.

The idea of a web-application seemed the perfect solution for those two issues:

- there would be no need to install the application locally, since every user could just use the tool by simply logging in onto the website;
- every user would be aligned on the latest version, since any new feature would be implemented on the server and immediately available to everyone.

However a centralized server calling external data-sources on behalf of different, potentially concurrent, clients created a different set of challenges. Online services who open up their database to external users via API, normally limit the number of requests a single user can make in two ways:

- total number of requests per user (e.g. 10,000 API calls each user);
- total number of requests per period (e.g. 100 API calls each minute).

API users need to register to a *developer program* and they need to include a API key in their GET or POST requests parameters, this way the data controller (who owns the data) can control the number of requests submitted by each member. A work-around solution could have been to request each user to register an API key for each service they wished to access and save their API key in Linkero's profile settings, so that each user was directly responsible for his/her own API calls quota. But it seemed a step-back from the original idea of insulating the user from lower-level inner workings of the application.

The other issue emerging from the web-application approach, related to the time the system takes to complete a report. eBay API results from a text search are paginated, and each page returns a maximum of 100 items, over a maximum of 100 pages. There are a total of 22 country sites where the same search can be repeated. After the search is completed we still need to retrieve the item details, which is done with a different API. Long story short: a report returning the maximum amount of results on all 22 eBay sites can take as long as 15 hours to complete, if requests are sequential and the transmission lag between request and response is half a second long. While a stand alone application could run for as long as needed, while the user runs other programs, you could not ask the user of a web application to keep the tab open until the request is completed.

The solution I came up with is inspired by the *broker pattern*: the broker is a component of the system that coordinates communications between different decoupled



components [Bosoo]. In Linkero, the broker is the part of the application that takes data requests coming from different clients, and returns a *report submitted* response to the client, so that the client knows his/her report has been scheduled and will receive a notification when it is ready, meanwhile the user can close the browser windows or request other reports. The broker on the other end, will queue the request and a single unit will then process every report in the queue sequentially, thus avoiding to exceed API call request limits with concurrent requests.

In the next section I will discuss how this architecture was implemented in practice.

## 3.2 System architecture

The system architecture represented in figure 3.1 provides an overview of all different applications that work together to perform the main system functionalities.

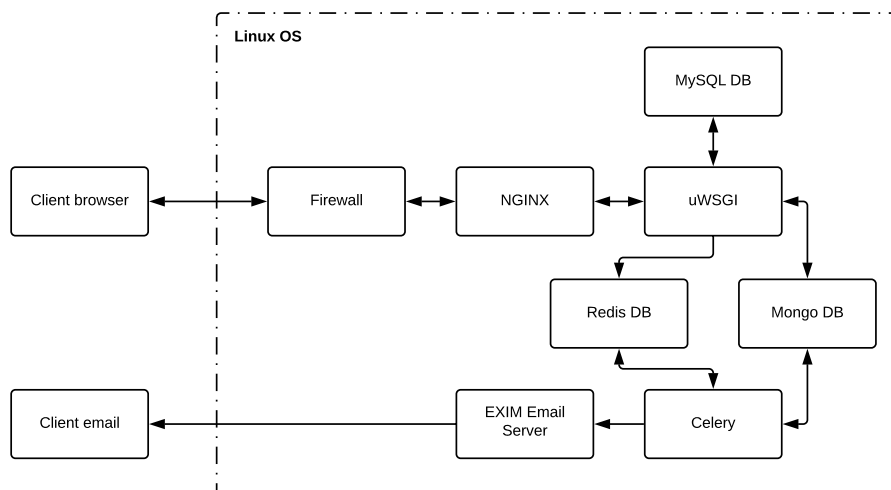


Figure 3.1: Linkero system architecture

As shown in the figure, client browser and client email are considered part of the system: the browser is included because some functionalities are passed to the client to run locally in the form of JQuery functions, and the client email is considered part of the system, because it represents the final destination of data files generated by the system.

Linux is the operating system running on a Virtual Private Server, that manages all other applications. The first point of entry is the Firewall, which monitors all requests knocking on ports 22, 80 and 443.

NGINX is the web server, reverse proxy server [Wik18a], which receives HTTP requests and assigns them to the correct internal web server.

uWSGI is the web engine that runs Python scripts, in the words of its developers: “uWSGI itself is a vast project with many components, aiming to provide a full software stack for building hosting services” [Ubi18].

Python scripts from within uWSGI can make calls to three databases. MySQL keeps information about users registration details, passwords, and access logs. Mongo DB instead is used to store all the data generated by users with their queries. Finally Redis DB simply stores information about tasks that have been scheduled and that will run independently from uWSGI’s scripts.

Celery is another server running Python scripts scheduled by the main web server. It does so by regularly checking in Redis if any new task has been logged, and if that’s the case it will execute each task sequentially.

Finally the email server is only called for outgoing emails, to deliver a copy of all the data collected in a .csv file attached to the email.

### 3.3 Technologies

The overall system was not written from scratch, there are many free and open-source applications that were assembled in order for the system to work.

#### 3.3.1 Server operating system

Linkero is hosted on a Virtual Private Server provided by the company Linode <sup>TM</sup>. I chose the smallest package offering 1 GB Ram, 1 CPU Core and 25 GB of SSD storage at \$5.00 USD a month.

The plan offers different operating systems to chose from, and Linux Debian is the OS that was picked for this project. There are several advantages to use a Linux distribution:

- Linux is a very lightweight system, which can run on hardware with even smaller resources than the ones available for this project;
- Linux is maintained by a global community who prioritize reliability and security over innovative/experimental functionalities;
- Linux is an open-source system, which means that its code can be read by anyone, increasing the chances of finding bugs and vulnerabilities, which is to say that Linux is very secure;

- Linux is also one of the most popular server technologies, guaranteeing that a system designed for Linux can access and be transferred to a vast public already using Linux;
- the Debian distribution development cycle emphasizes stability over innovation, offering products that are well tested before being published;
- installation and use are free and allowed under the GNU General Public License, making it ideal for a small budget project;
- the Debian distribution is one of the most rich in extra applications ready to be installed from their official directories [Deb18].

All the reasons above make Linux Debian the ideal candidate for this project.

### 3.3.2 Web application framework

The main business logic in this project is written in Python and runs on a uWSGI web engine, implementing Python's WSGI convention. However the actual set of objects and functions that constitute the web application have been developed using Django. Django is a "web framework for perfectionists with deadlines", quoting from Django developers' tag-line [Fun18b]. There are several reasons that make Django a good candidate for this project:

- the web development framework is freely distributed under the BSD Licence;
- Django offers object-relational mapping classes designed to interact directly with a relational database using Python APIs, simplifying interactions and data retrieval/manipulation operations with a SQL database;
- Django has already built-in user authentication, authorization and session management support, along with a pre-installed admin console with basic functionalities. This allows the development process to focus on core functionalities, rather than having to code standard session management and authentication procedures;
- Django comes with several security features to prevent web-specific attack, such as: cross-site scripting, cross-site request forgery, SQL-injection, click-jacking;
- Django is designed with scalability in mind: it enforces strict separation between application layer and database layer, so that new hardware can be added with no costly impact due to reconfiguration;

- successive Django framework versions include backward compatibility and try as much as possible to maintain consistent interfaces and API.

Choosing Django also means being able to access all the Python libraries available for data analysis, data mining and system administration.

### 3.3.3 Databases

There are three different databases that support all the data needs of this project.

#### **MariaDB**

MariaDB is a version of MySQL that forked from the original development in order to remain free under the GNU GPL licence. There is no special reason to prefer this relational database over the parent MySQL, other than it is the default version available in the original Debian repositories, since all Debian software is protected by GNU GPL.

MariaDB is used as default server for user authentication and session management, and to keep logs of user log-in, user IPs and browsing activity.

#### **MongoDB**

MongoDB™ is a NoSQL, schemaless database that stores data in JSON-like record structures called documents. MongoDB Community Edition is made available as Debian package under the AGPL licence.

MongoDB is used to store all the data generated while querying external sources, and is a vital component of the project. The reason for this choice is simple: MongoDB natural data format is JSON, and does not require the data schema to be declared in advance. This feature relieves much burden from the development stage because there is no need to commit to rigid data-schema based on the information that are being extracted from external sources, there is also no need to map a JSON response to a SQL INSERT query. The final implementation will result in less hard-coded configuration, less maintenance, more flexibility to accept modification to existing sources and to add new sources. This does not mean that some of the data points extracted should not be indexed, it actually is recommended to index some crucial data-points to improve search performance, and MongoDB provides this feature.

There are other features that make MongoDB attractive for this project:

- field, range and regular expressions queries are supported;
- results can include user-defined JavaScript functions;

- high availability is guaranteed thanks to built-in replication functions that make copies of selected data. This feature is relevant if MongoDB is deployed across a cluster of multiple machines;
- high scalability is also a feature by design;
- aggregation can be performed in three different ways, depending on the requirement: aggregation pipeline, map-reduce and single-purpose aggregation methods;
- finally, as of June 2018, the last version of MongoDB supports full ACID transactions.

MongoDB does not support SQL style queries, but its query language is very intuitive, since it reproduces an object-oriented method calling style, for instance:

```
1 db.inventory.find( {  
    status: "A",  
3    $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]  
} )
```

in SQL equates to:

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p  
%")
```

## Redis

Finally, the last database implemented in this project is Redis™. The choice of this database was dictated by the need of running an independent parallel Python thread that would receive requests for specific operations, and execute them asynchronously from the main web application. A graphical representation of this asynchronous execution is represented in the sequence diagram in figure 3.4. Redis is short for Remote Dictionary Server, is a database storing key-value data structures in the running memory, and is distributed under the BSD licence [Sal09]. Redis offers the option to transfer some data to permanent storage, but that is not its strength. Redis shines in a system with multiple parallel processes that need to exchange data between each other. Keeping the data in the running memory allows very quick I/O operations. Also Redis does not support complex queries, and data aggregate operations, but simple index lookups.

### 3.3.4 Celery

Celery is an “asynchronous task queue” [Cel09], developed in Python and distributed under the BSD Licence. Celery regularly checks if one or more tasks are being logged in the Redis database, and executes them in order, while the main program regains control immediately after submitting a task, so that its execution is not dependent on the task.

This choice is dictated by two factors:

- gathering all the data for one single report can take hours, depending on the volume of data requested and the speed of connection. In a web application it would result in a bad user experience if the page used to submit a report would hang for a long period of time waiting for data to be returned.
- Services offering API calls to access their data often limit the call frequency. Since Linkero is designed to sustain a growing user population, the system would reach a stage where multiple users could submit concurrent reports, and that would likely exceed the API rate limit.

In order to solve those two issues, Celery is the central server that receives and queues all user generated tasks, while executing them one at a time. From the user perspective, after submitting a report, a feedback message would acknowledge the fact that the report was submitted successfully, and that a notification will be sent via email once the report is ready. From the API call provider perspective, one single agent would make multiple API calls, making sure not to exceed the limit.

In this configuration, Redis is a message broker that mediates between the web engine uWSGI and Celery, when tasks are scheduled.

### 3.3.5 Email server

The email server installed on this system is Exim [Haz95] which is a very well established application, first released in 1995 under GNU licence. The choice of this server is dictated by the fact that under the current version of the project a simple send-only email server is sufficient to allow the system to send out email notifications to Linkero users once a report is completed.

### 3.3.6 Web server

NGINX was chosen to serve the pages of this application. NGINX is an http proxy and web server [Ngio4], released under the BSD licence.

The main reasons for choosing Nginx over other popular web servers is that its lightweight even under significant load (2.5MB for 10k simultaneous connections [Wik18b]), and offer native support of WSGI.

There are other reasons that make NGINX a good candidate for the project:

- acting as reverse proxy, NGINX adds a layer after the firewall, obfuscating the location of web servers, thus improving the system security.
- Performs system load balancing when multiple web servers are deployed.
- Manages centrally TLS encryption on behalf of every web server under its control.
- It can host multiple web servers from a single IP.
- Finally, it can cache static content, reducing the load to the origin servers.

### 3.3.7 Firewall

The firewall application IPtables used in this project comes as default with the Debian operating system and is distributed freely under the GPL licence [Net98]. There are no particular reasons why this application was preferred over others other than:

- it is easy to configure;
- it is an established application, well documented;
- offers filtering at network and transport layers, to provide stateful connections inspection.

These are the rules implemented for this project:

- the default policy on INPUT, FORWARD and OUTPUT chains is to reject unless otherwise specified;
- block every new incoming request for ports other than 22 (SSH), 25(SMTP), 80 (HTTP) and 443 (HTTPS);
- DROP incoming connections from reserved addresses like (but not limited to) 10.0.0.0/8, 169.254.0.0/16, 127.0.0.0/8 or 192.168.0.0/24;
- DROP all packets whose state is INVALID;
- block for 24 hours IPs that perform portscan;

- allow loopback connections;
- allow output traffic to ports 22 (SSH), 25 (SMTP), 53 (DNS), 80 (HTTP) and 443 (HTTPS).

Working along with IPtables, Fail2Ban scans IPtables' log files and dynamically adds IPtables rules to ban those IPs that behave suspiciously. In this system installation Fail2Ban is configured to look at SSH login attempts that fail three times within 24 hours, and bans them for a full day. This setting targets mainly bots that try to brute-force their way into a server connected to the public internet, by attempting to login into SSH using a long list of combinations of username and password. An unusual username and strong password normally are sufficient to prevent unwanted intrusions, however leaving the system open to accept and evaluate any volume of login could leave it exposed to DDoS attacks. Banning activity coming from a suspicious IP is a way to relieve the pressure from the SSH application, leaving it available to legitimate users [Ja04]. As of November 2018, there are currently 603 IPs being banned for a full day, and 8,845 that have been banned at some point in the last four months of activity.

### 3.4 Development methodology

For this project, considering its relative simplicity and limited headcount (one person working on it), Extreme Programming seemed to be the right development methodology. The methodology is subdivided into four main principles:

**listening** : listen to users' requirements and feedback, constantly throughout the entire development cycle.

**simplicity** : keep the design simple and clear.

**test-driven development** : write the test before the actual code.

**iterative releases** : divide the system in versions that can be developed over short plan and release cycles [Pre05].

In the next three sections we will look at the list of requirements and use cases created for this project, the object oriented class diagrams and some examples of the resulting code. The next chapter will be dedicated to the testing performed and results.

Extreme programming encourages to assign priority to each use-case and estimate what is called *project velocity*, otherwise known as the amount of weeks required to complete each use-case. With the current level of knowledge, each and every use-case



could be realistically be developed under a week of full time work. However that did not reflects the reality of time dedicated to each use-case.

Also, extreme programming focuses on the coding part of the development, which fits the business logic of the project. It is worth to mention that a lot of time was actually spent configuring off-the-shelf application that constitute vital parts of the system: enabling NGINX to interact with uWSGI, setting MongoDB user profiles and credentials, creating an SSL certificate and registering it in NGINX, and more. All these activities are vital for the successful implementation and working of the system, but do not feature in the XP paradigm.

### 3.5 Use cases

The figure 3.2 represents how the two main end-users types of interactions with the system.

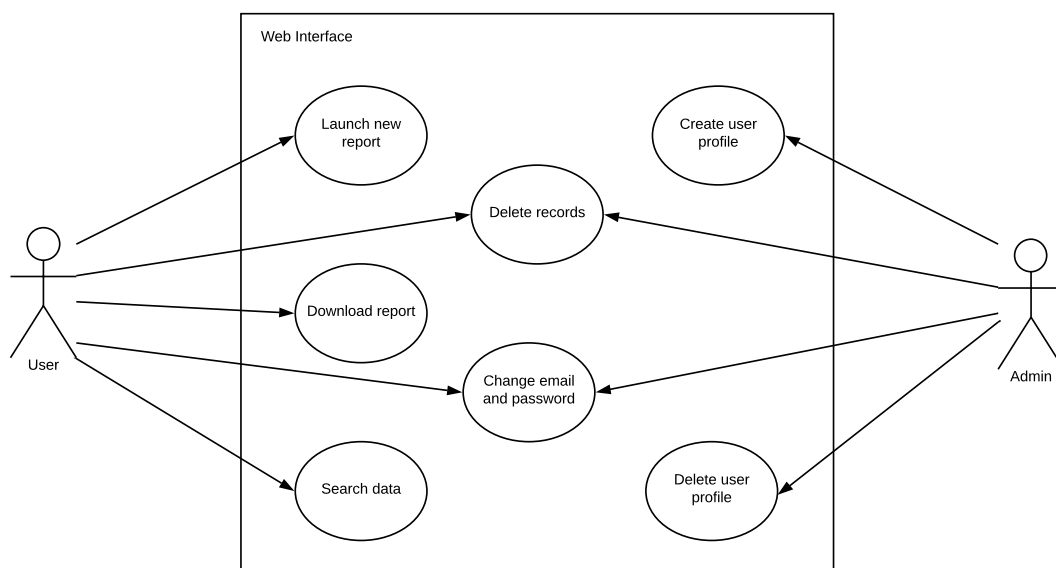


Figure 3.2: Use cases diagram

The figure provides a simplified set of possible actions, with no regard to the exact order in which they may be presented to the user under realistic circumstances. Use-cases are a schematic way to visualize the main functions of a system from the users' perspective.

The following set of tables will flash out what each use-case requires and what provides:

<b>Use case name</b>	Launch new report
<b>Actor</b>	Investigator
<b>Description</b>	The user selects the type of report he/she wants to generate, fills in the necessary input and submits the request.
<b>Preconditions</b>	The user has to be logged into his/her profile.
<b>Trigger</b>	Click on <i>Submit</i> button.

Table 3.1: Launch new report use case

<b>Use case name</b>	Delete records
<b>Actor</b>	Investigator, Admin
<b>Description</b>	The user deletes records associated with a case.
<b>Preconditions</b>	The user has to be logged into his/her profile.
<b>Trigger</b>	Click on <i>Delete</i> icon.

Table 3.2: Delete records use case

<b>Use case name</b>	Create user profile
<b>Actor</b>	Admin
<b>Description</b>	The user create a new profile with a username, email and password that will allow a new user to access the system functionalities.
<b>Preconditions</b>	The user has to be logged into his/her profile.
<b>Trigger</b>	Click on <i>Save</i> button.

Table 3.3: Create user profile use case

<b>Use case name</b>	Download report
<b>Actor</b>	Investigator
<b>Description</b>	The user selects a report that has already been generated, and downloads a csv file with the data.
<b>Preconditions</b>	The user has to be logged into his/her profile. Data from a report has to be already available and saved in the database.
<b>Trigger</b>	Click on <i>Download</i> icon.

Table 3.4: Download report use case

<b>Use case name</b>	Change email and password
<b>Actor</b>	Investigator, Admin
<b>Description</b>	The user can change the current password and email address.
<b>Preconditions</b>	The user has to be logged into his/her profile.
<b>Trigger</b>	Click on <i>Save</i> button.

Table 3.5: Change password use case

<b>Use case name</b>	Search data
<b>Actor</b>	Investigator
<b>Description</b>	The user can submit a set of keywords and find out how many reports are available in the database associated with that term.
<b>Preconditions</b>	The user has to be logged into his/her profile.
<b>Trigger</b>	Click on <i>Search</i> button.

Table 3.6: Search use case

<b>Use case name</b>	Delete user profile
<b>Actor</b>	Admin
<b>Description</b>	The user can delete the profile of a user that no longer needs access to the system.
<b>Preconditions</b>	The user has to be logged into his/her profile.
<b>Trigger</b>	Click on <i>Delete</i> button.

Table 3.7: Delete user profile use case

## 3.6 Requirements

I already mentioned some of the requirements and system constraints in the *system architecture* section. In this section I will discuss in more details all the requirements considered for the current implementation. Requirements are further divided in functional, non-functional and user requirements.

### 3.6.1 Functional requirements

Functional requirements concern the core business logic of the system:

- F1** the system should connect successfully to external sources, public API, to retrieve relevant data.
- F2** The system should collect data extracted from external sources and store it in a local database.
- F3** The system should avoid storing duplicates of the same record within one report request.
- F4** The system should assemble all the data gathered for one user's request and print it in a csv file format.
- F5** The system should send a copy of the data, to the requesting user, in a csv file attached to a notification email.
- F6** The system should limit the amount of API calls to each external data source, in order not to breach API user policy.

### 3.6.2 Non-functional requirements

- N1 The system should restrict access to authorized users only.
- N2 The system should store users' passwords encrypted.
- N3 The system should store data gathered from external sources encrypted while the data is at rest.
- N4 The system should be hardened against most common web attacks.
- N5 The system should be prepared to accommodate larger data sets;
- N6 The system should be prepared to accommodate more web servers to increase performance, redundancy and fault-tolerance.
- N7 The database should allow flexible storage of heterogeneous data-schemes from different sources.

### 3.6.3 User requirements

- U1 The application should allow access to the authenticated user.
- U2 The user should be able to change his/her password and email.
- U3 The application should present the list of all queries run by the user in the last two weeks.
- U4 The application should allow the user to search and filter through his/her cases based on case creation date range and platform (e.g. eBay, MercadoLibre, Allegro, Facebook, etc.).
- U5 The application should provide input-forms to allow data extraction from all the external data sources available.
- U6 The application should perform data gathering operations without stopping the user from performing other operations.
- U7 When the data is ready to download the user should be notified.
- U8 The user should be able to access both data and input provided for old cases.
- U9 The user should be able to delete old cases and data associated with them.

### 3.7 Class diagram

The class diagram in figure 3.3 represents the core modules of the web application.

Django provides three parent classes that represent web-forms, html page generators and database tables called respectively: Forms, Views and Models. These classes can be inherited to produce specific objects to fit the needs of the project.

The Forms class for instance offers default fields that represent the values of its fields (e.g. text fields, date fields, numeric fields...), and methods that allow data sanitization, html layout, data validity. Linkero inherits this class to produce the `CaseFilerForm`, which is a simple form that enables users to limit the amount of cases to be displayed on the main page, by selecting a time range and a platform (e.g. show only cases from Jan to Feb 2018, that pulled eBay data). The `CaseFilerForm` object was created by declaring the list of fields that should be part of the form, all other methods were inherited from the parent class.

The Models class is the core of Django's ORM (Object Relational Mapping) philosophy: it is a Python object that maps its fields to a database table (or document in MongoDB's case), and offers as many methods as SQL or NoSQL functions to read and write data. Like for Forms, the Models parent class provides all the methods that are needed, while the developer is left with declaring the field types for each object. In this project, the `CaseDetails` model represents information about a specific case: the case identification number, the creation date, the case owner, the platform which sourced the data, the inputs provided for the data search. Similarly the model `EbayItem` stores all the information about an eBay item that was queried and stored in MongoDB.

The Views class represents the objects that receives a http request and returns a response. Each child class will have their own implementation of one or both GET and POST methods. The class `CaseView` for instance represents the home page being displayed after the user logs in: a table that summarizes all the cases generated in the last two weeks, and a menu to bring up modal forms to submit new data requests. The GET method detects when it is receiving a request to load the full page, or an AJAX GET request to load just a new filtered set of cases in the cases table. The POST method of the `CaseView` class receives requests to generate new reports.

The last class in the diagram is `EbayAPI`. This class does not inherit from any other built-in Django object. Instead it was created from scratch to provide querying facility with eBay public APIs. Its fields represent authentication details for the local database and the remote server, whereas its methods are designed to extract data based on user's input, leveraging eBay APIs like general search by keyword, get details by item number or get seller details by seller id.

The diagram indicates also classes' level of coupling, or how much a modules shares data and interacts with other modules. Forms and Views are in one-to-one relations most of the time, since forms are unique, and each view incorporates one form for each type. Models and Views instead are in many-to-one relations, since each view can interact with multiple instances (i.e. database records) from the same class, but there is always only one view object at any given time.

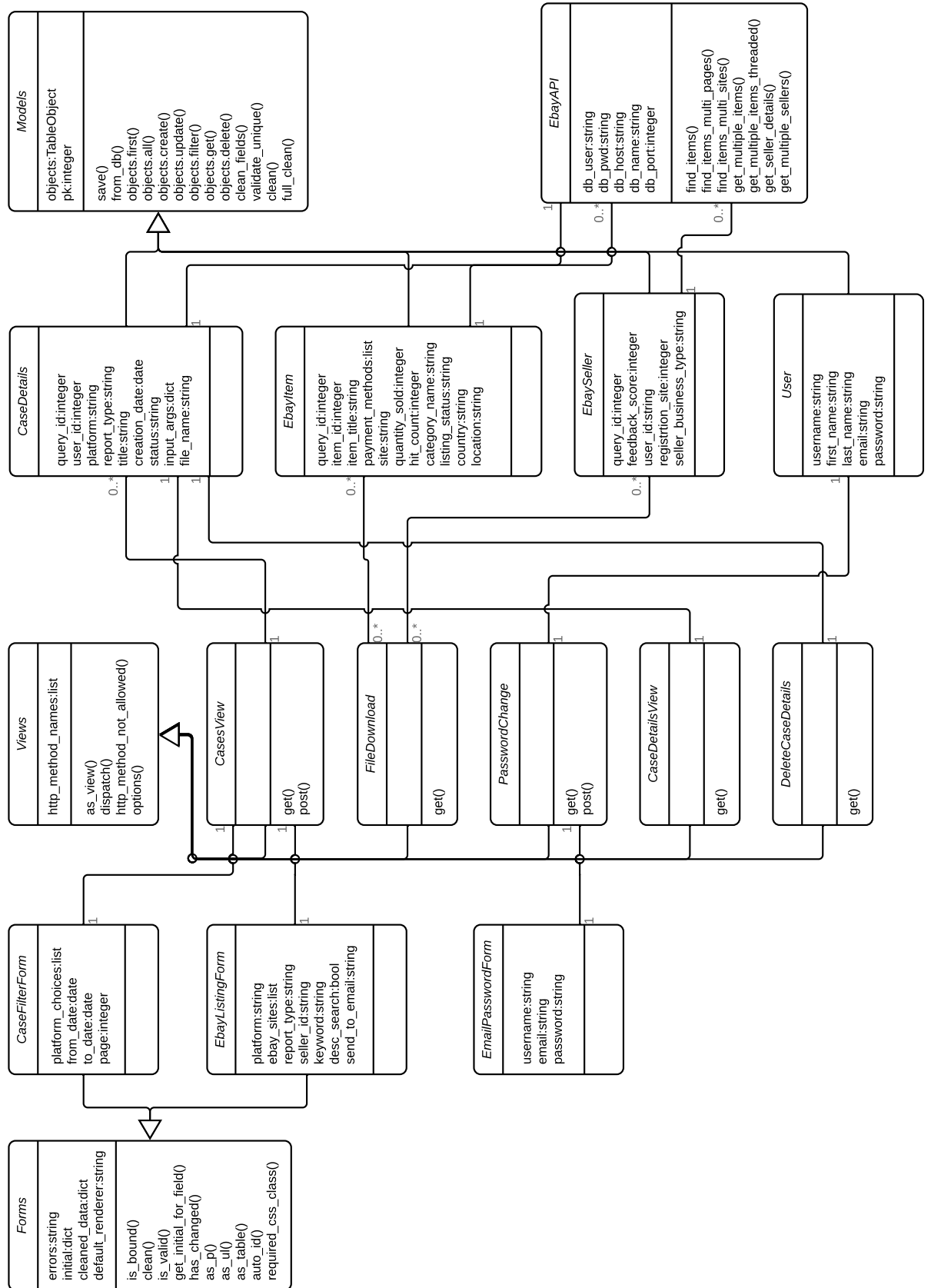


Figure 3.3: Class diagram



## 3.8 Sequence diagram

The sequence diagram represented in figure 3.4 is probably the most elaborate process performed by the core business logic in Linkero. It describes the sequence of operations that the system performs after receiving a request to generate a new report.

The entire sequence can be broken down into the following steps:

1. The process is triggered by the user submitting a POST request through a web-form designed to gather all necessary input to pull data from external resources.
2. The `CaseView` object receives the request and immediately saves the details of the new case in the database by making a call to the `CaseDetails` object and using the `save()` method. The case details are saved and nothing is returned, assuming there are no errors or exceptions raised.
3. Next, the `CaseView` instance submits a report request to the Task controller, the controller queues the request and returns the call to `CaseView` instance that in turn can send a response back to the client browser that initiated the process, confirming that the request for a new report was successfully submitted.
4. The Task controller script continues the process asynchronously by requesting the `EbayAPI` to pull a list of items.
5. The Task receives the results and saves them with the `save()` method of the `EbayItems` object.
6. The Task controller proceeds to request details of the sellers extracted from the previews search by calling the `search_sellers()` method of the `EbayAPI` object.
7. The results are then saved by the `EbaySellers` object.
8. Once the data necessary to generate a new report is ready, the Task script creates a new `.csv` file and parses the results there.
9. The Task controller sends the data to the `EmailServer` objects that is responsible for sending an email notification to the user who requested the report.
10. Finally, the Task controller changes the status of the request from *running* to *completed* by using the `update_status()` method of the `CaseDetails` instance.
11. While all other objects remains ready to answer calls, the Task script ends its cycle.

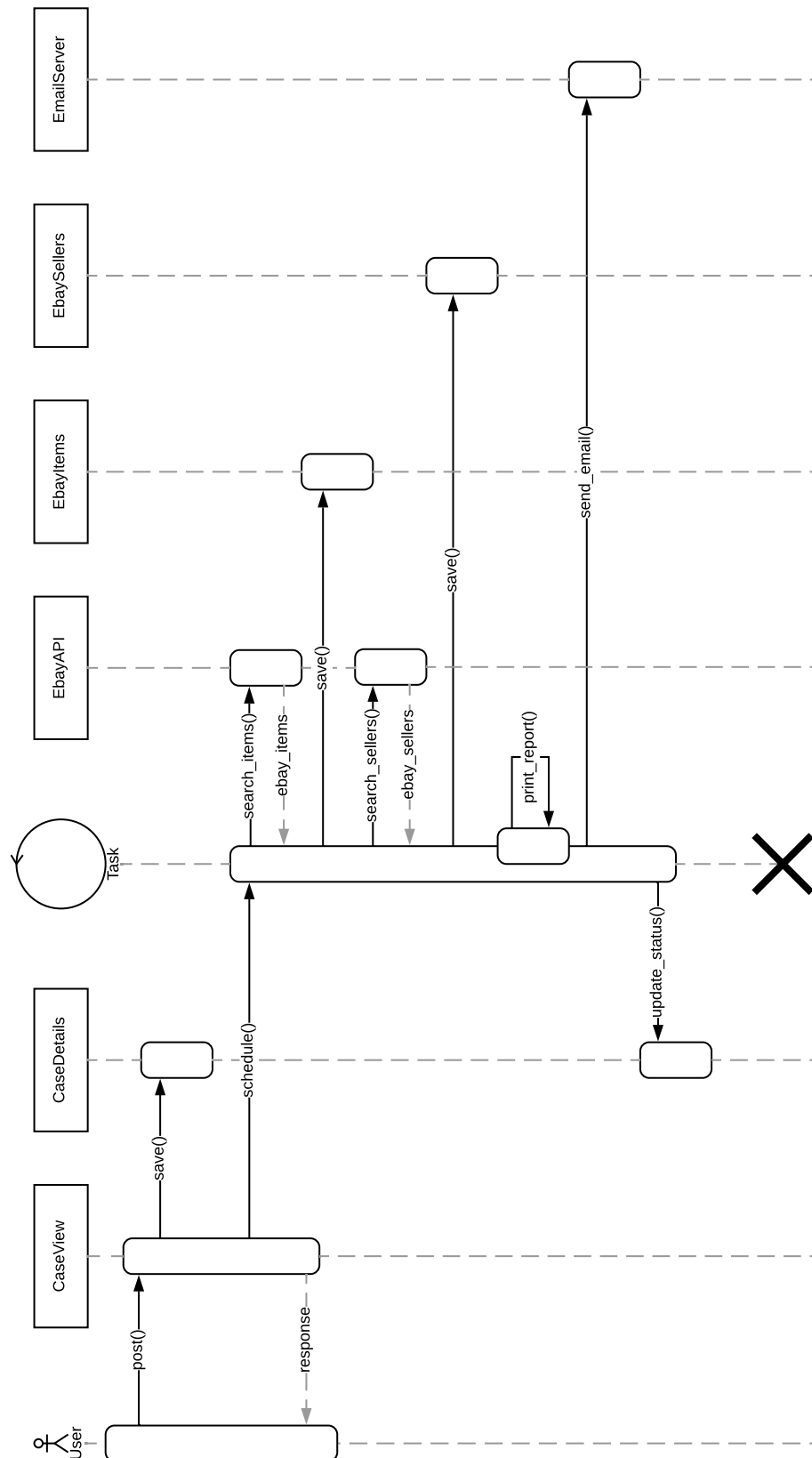


Figure 3.4: Launch report sequence diagram

## 3.9 Code samples

In this section I will look at some code snippets that make up the Task script just described in the sequence diagram.

The first part of the script is dedicated to load necessary libraries of functions that support the script execution, including the EbayApi object, and the models connected with the MongoDB database:

```
1 import os
import time
3 import json
from csv import DictWriter
5 from io import StringIO
from celery import task
7 from pandas import merge
from pandas.io.json import json_normalize
9 from mongoengine import connect
from django.contrib.auth.models import User
11 from django.core.mail import EmailMessage
from cases.ebayapi import EbayApi
13 from cases.models import EbayItem, EbaySellerDetails, ApiErrorLog,
    CaseDetails
```

Immediately after the task process is declared:

```
1 def send_ebay_listing_report(to_email, user_id=None, query_id=None,
seller_id=None, keywords=None, ebay_sites=['US'], search_desc=False):
```

In the following lines I create a connection to MongoDB, so that instances representing its collections (read tables) will be able to retrieve and store data in them, and I initiate an EbayApi instance:

```
    # connect to Mongo
2    mongo_client = connect('linkerodb', username='linkero-user',
password='password123')
4
    ea = EbayApi()
```

The script then uses the `find_items_multi_sites()` method of the `EbayApi` instance to search and pull all items that match the criteria input by the user. If the search returns results, the script continues to generate the report, otherwise it jumps to a last step sending a notification to the user that no items were found:

```

1 items_dict, slr_list, find_error_list = ea.find_items_multi_sites(
    e_sites=ebay_sites, kwd=keywords, s_id=seller_id, s_desc=search_desc)

3 # check if find items search returned results
  if items_dict:

```

Assuming that the first exploratory search was successful, the script proceeds to pull the details for each item, and subsequently the business details for each seller:

```

# pull item descriptions for each item
2 ebay_item_list = ea.get_multi_items_threaded(items_dict, q_id=
  query_id)

4 # pull seller details
  seller_list, seller_err_list = ea.get_multiple_sellers(
seller_list=slr_list)
6 seller_collection_list = []
  for slr in seller_list:
8     slr['lnkr_query_id'] = query_id
    seller_collection_list.append(EbaySellerDetails(**slr))
10 # insert in bulk
    EbaySellerDetails.objects.insert(seller_collection_list)

```

Probably not the most consistent class design implementation, but in the case of the EbayApi instance method `get_multi_items_threaded()`, items are saved in the corresponding MongoDB collection by the EbayApi instance itself; while seller details are retrieved by the EbayApi and passed back to the Task script in memory, and it is the Task script that calls the database object `EbaySellerDetails` to save the details retrieved. This *double standard* was implemented after a series of tests, when I found that leaving the data in the Task script memory would cause memory overflow and the script would crash. It is worth to remember that the system that is currently running this project only has 1GB of memory.

Continuing, the data extracted is then assembled into a csv file:

```

1 # save the results in a CSV file and send it attached
  e_items = EbayItem.objects(lnkr_query_id=query_id)
3 items_df = json_normalize(json.loads(e_items.to_json()))

5 sellers_df = json_normalize(seller_list)

7 df = merge(items_df, sellers_df, left_on='Seller.UserID',
  right_on='UserID')

```

```
9     filename = "linkero_ebay-listings_{}.csv".format(time.strftime("%Y%m%d-%H%M"))
```

In order to save system hard-disk space, the script creates a file-like object `StringIO` that is compatible with Python file manipulation API, but is never committed to permanent storage:

```
1     file_attachment = StringIO()
2     writer = DictWriter(file_attachment, fieldnames=headers)
3     writer.writeheader()
4     writer.writerow(df.to_dict('records'))
5     file_attachment.seek(0)
6     email.attach(filename, file_attachment.read(), 'text/csv')
7     email.send(fail_silently=False)
8     file_attachment.close()
9
10    # update the case execution status
11    query_status = 'completed'
12    CaseDetails.objects(lnkr_query_id=query_id).update(set__file_name
    =filename)
```

The last line calls the MongoDB instance dedicated to case details, `CaseDetails`, and changes the status of the report from *running* to *completed*.

The full code of the Django application is available at <https://github.com/srichiardi/linkero>.

### 3.10 Front end screenshots

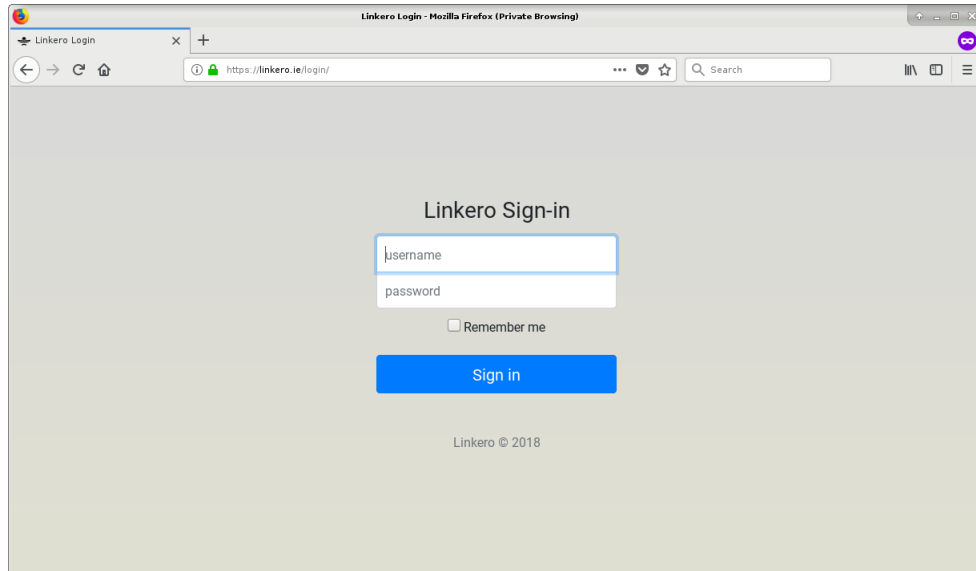


Figure 3.5: Login page

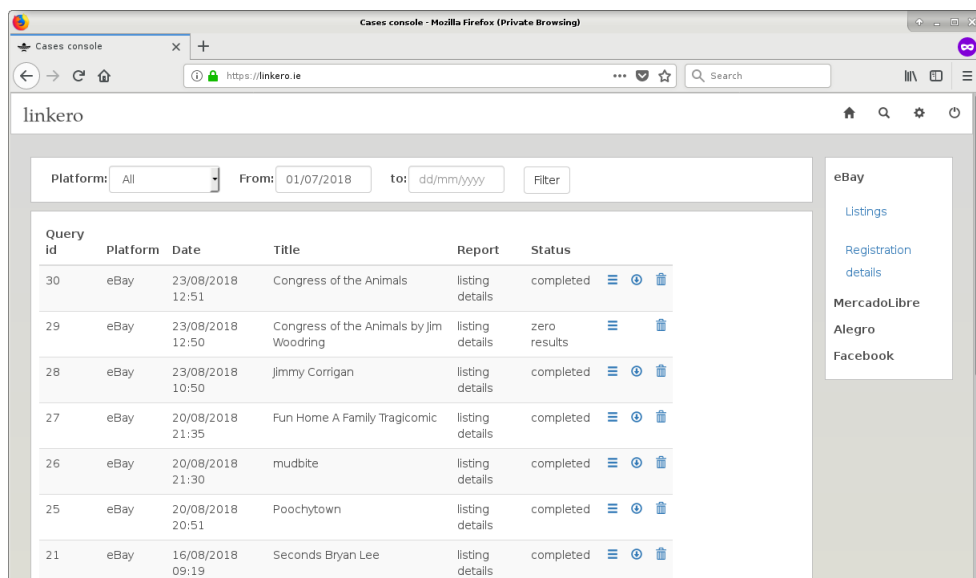


Figure 3.6: Home page

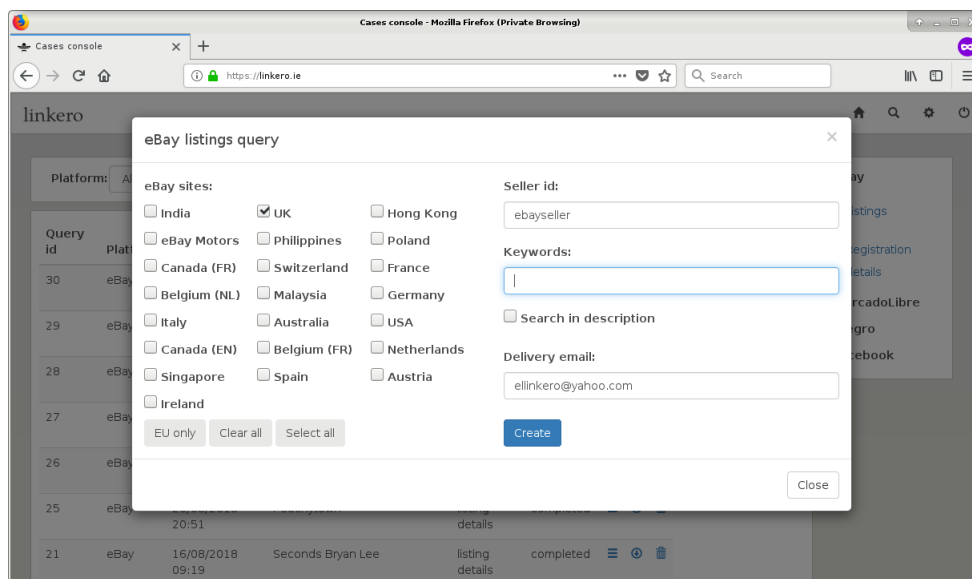


Figure 3.7: eBay input modal form

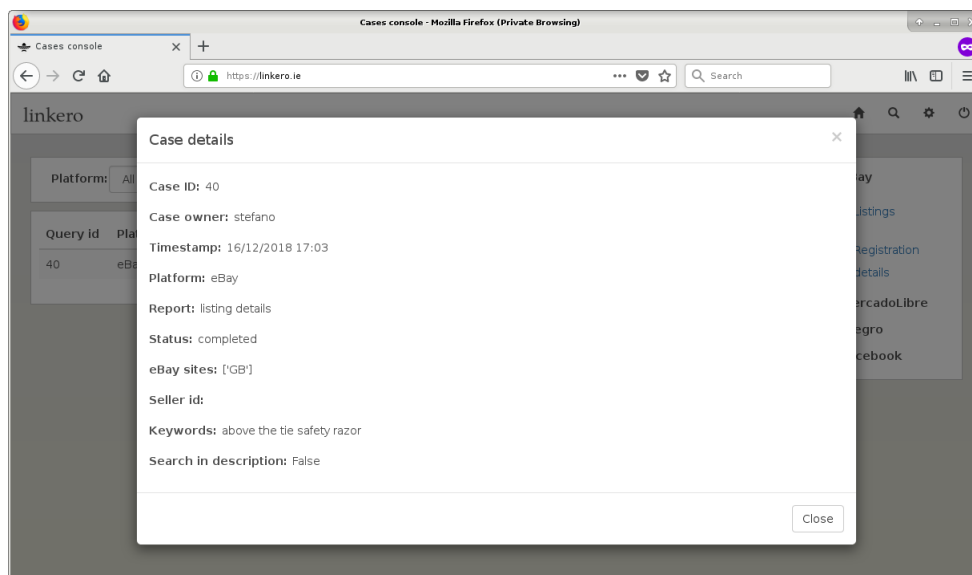
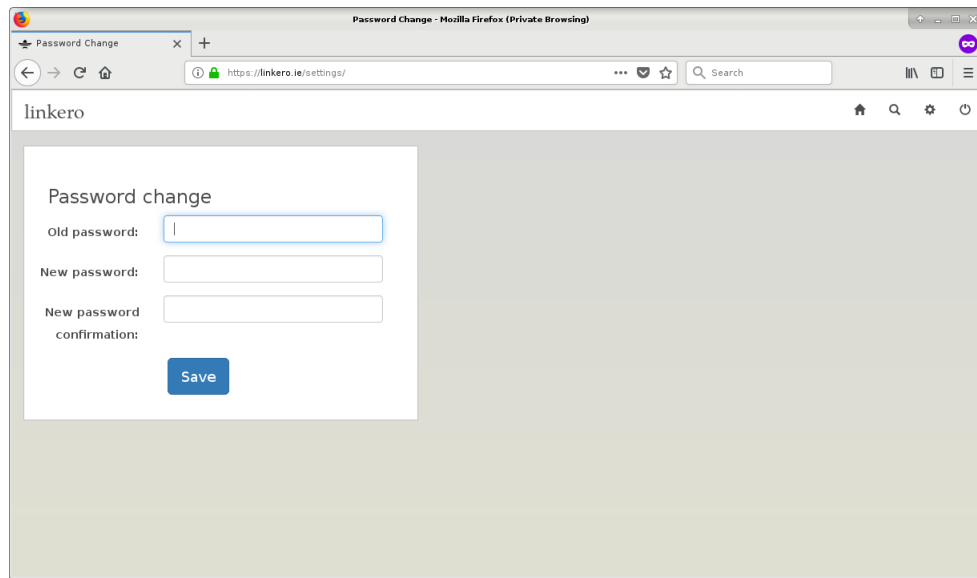


Figure 3.8: Case details modal



The screenshot shows a web browser window titled "Password Change - Mozilla Firefox (Private Browsing)". The address bar displays "https://linkero.ie/settings/". The page header includes the "linkero" logo and navigation icons. The main content area features a "Password change" form with the following fields:

- Old password:
- New password:
- New password confirmation:

A blue "Save" button is located below the confirmation field.

Figure 3.9: Password change form



## Chapter 4

# Testing and deployment

### 4.1 Testing strategy

Testing a system has two meaning: we need to find out if the system build works as intended (verification testing), but we also want to know if we built the right system (validation testing) [Vlio8]. For this project I will assume I did build the right system, and I will focus on verifying that what I built performs correctly and complies with its original requirements.

Testing is a tricky business, since its ability to detect issues is only as good as the set of test cases that are developed. Testing exhaustively every outcome of a system component is not a feasible solution, since even a very simple program consisting of a single for loop running 100 iterations, testing a condition and returning a binary output, could generate  $2^{100}$  different outcomes, which would mean that a brute-force testing with a machine able to execute 10 million instructions per second, would take  $3 \times 10^{14}$  years [Vlio8].

The alternative is to chose a limited set of test cases. The question then becomes, how do we chose an *adequate* set of test cases? The answer is a matter of priority based on the original agreement between the final user and the system developer:

- the testing could prioritize making sure that every requirement is met precisely as defined, which is considered **coverage-based testing**, but this approach assumes that requirements are defined in quantitative terms;
- or testing could focus on fault detection, **fault-based testing**, which means that a lot of effort is first put into creating a testing mechanism which needs to be calibrated by seeding intentional faults withing the system so that we can calculate the test's detection rate;

- finally, test cases can be designed based on known errors that are normally found in any system, such as *off-by-one* errors. The testing therefore will focus on exploring the boundary points [Vlio8].

The above classification of test techniques applies to the component-level, so individual class and object methods are tested independently from the rest of the system, this practice is called **unit-testing**. Linkero was developed following the Extreme Programming methodology, which requires component tests to be written before the module, the test is meant for, is actually written. It is a test-driven approach that forces the developer to think in advance and anticipate the potential cases of failure as much as the expected outcome [Vlio8].

In a traditional testing approach, unit-testing is followed by **integration-testing**, when individual components are then assembled and tested during their interaction with one another. Eduard Berard points out that integration testing is a practice that originates from a procedural software development culture and does not apply as well to object oriented languages. A solution is *use-based* testing, which proceeds by first testing classes that are generally independent, and then proceeding by including dependent classes and continuing to the next layer [Ber93]. Linkero, being based on Python, is built following the object-oriented style, and does not display that many layers, therefore integration testing will not be performed.

*System testing* is the last stage of testing before deployment, in which the system is tested as a whole. There are different types of system testing:

**Recovery testing** is designed to cause failure status to verify how the system is able to restore its original state.

**Security testing** verifies that protection mechanisms are in place against the most common attacks.

**Stress testing** is a test that executes instructions that demand a very high level of resources in order to assess the system's ability to sustain a significant workload [Pre05].

In this project I will consider only security testing, which is one of the main requirements identified on the outset.

## 4.2 Unit testing

Python has its own official testing framework: the `unittest` module. This framework allows to write test cases and, obviously enough, test if the output returns as expected

or not. Django integrates unittest in its libraries, and provides by default a `test.py` file for each web-app created, so that projects' tests can be stored and reused repeatedly.

This is an example of the approach that I took for this project. Before creating a method that would search for eBay items based on keywords and a list of regional eBay sites (the type of results of an items search on eBay are dependant on the region selected: "skateboards" on ebay.com will return different results compared to ebay.co.uk). In this test case I wanted to check the ability of the method to return a meaningful error message with the wrong eBay site was provided. I prepared a test in `test.py`, feeding a wrong eBay site to the `find_items()` method of the `EbayApi` class:

```

1 from django.test import TestCase
2 from cases.ebayapi import EbayApi, EbayBadSite
3
4 # Create your tests here.
5 class FindItemsTest(TestCase):
6     def setUp(self):
7         self.ea = EbayApi()
8
9     # test if the method accepts an eBay site not correctly formatted
10    def test_bad_ebay_site(self):
11        wrong_site = 'PK'
12        with self.assertRaises(EbayBadSite):
13            self.ea.find_items(ebay_site=wrong_site, keywords="baby oil")

```

The first test fails as expected, because the exception class `EbayBadSite` is not defined:

```

1 (Projects) stefano@attila:~/Projects/linkero$ python manage.py test
2 Creating test database for alias 'default'...
3 System check identified no issues (0 silenced).
4 E
5 =====
6 ERROR: test_bad_ebay_site (cases.tests.FindItemsTest)
7 -----
8 Traceback (most recent call last):
9   File "/home/stefano/Projects/linkero/cases/tests.py", line 12, in
10     test_bad_ebay_site
11     with self.assertRaises(EbayBadSite):
12 NameError: name 'EbayBadSite' is not defined
13 -----
14 Ran 1 test in 0.004s
15

```

```

17 FAILED (errors=1)
17 Destroying test database for alias 'default'...
```

The second iteration brings a slightly different result. The exception class is now available, but there is no code that raises it, instead a different error arises when the bad input is used to retrieve any of the pre-loaded sites, the dictionary object throws a `KeyError` exception, signifying that the bad input *PK* is nowhere in the list of pre-defined country codes:

```

1 Creating test database for alias 'default'...
System check identified no issues (0 silenced).
3 E
=====
5 ERROR: test_bad_ebay_site (cases.tests.FindItemsTest)
-----
7 Traceback (most recent call last):
  File "/home/stefano/Projects/linkero/cases/tests.py", line 13, in
    test_bad_ebay_site
9     self.ea.find_items(ebay_site=wrong_site, keywords="baby oil")
  File "/home/stefano/Projects/linkero/cases/ebayapi.py", line 39, in
    find_items
11     'GLOBAL-ID' : globalSiteMap[ebay_site]['globalID'],
KeyError: 'PK'
13
-----
15 Ran 1 test in 0.004s
17 FAILED (errors=1)
17 Destroying test database for alias 'default'...
```

Eventually, some iterations later, after including an initial step to check the input validity ...

```

2     # check input provided
2     try:
        global_site_id = globalSiteMap[ebay_site]['globalID']
4     except KeyError:
        raise EbayBadSite("Wrong ebay site input!")
```

... the test is a pass:

```

1 Creating test database for alias 'default'...
```

```

System check identified no issues (0 silenced).
3 .
-----
5 Ran 1 test in 0.003s
7 OK
Destroying test database for alias 'default'...
```

This iterative approach was repeated for the other classes and methods. The class `EbayApi` for instance was tested for the following cases:

- test if the `EbayBadSite` error raised by the `get_multiple_items()` method when a bad eBay site is passed as argument;
- test if the `EbayTooManyItems` error is raised by `get_multiple_items()` method if more than 20 items are passed at a time.
- test if the `EbayInvalidSellerId` error is raised by the `get_seller_details()` method if a non-existent seller id is passed as argument.

The class `CasesView` was tested for:

- the `post()` method returning a JSON response equivalent to

```
{ 'status' : 'fail',
  'error_msg' : 'provide at least one seller id or keyword'}
```

when the user leave blank both seller id and keyword fields.

- the `get()` method returning a JSON response equivalent to

```
{ 'status' : 'fail',
  'error_msg' : 'invalid input provided'}
```

when the wrong platform input is provided to search for cases.

## 4.3 Security testing

Security testing was done with Zed Attack Proxy (ZAP), an open source web application security scanner. The application is developed maintained by the Open Web

Application Security Project, an online community that focus on online security awareness and support for security testers and penetration testers [OWA17], and it is considered one of the most reliable to detect vulnerabilities.

I performed the default active scan against `https://linkero.ie`. This first attack was performed on the public pages `/login/` and `/logout/`, and the tool identified six low priority issues:

**Password autocomplete enabled** : this option in any other input field allows the browser to predict the value while the user is typing. Values are predicted based on previous inputs, previously stored. Allowing the password to be stored for future auto-completion, is not a good idea. The fix involves adding the option `autocomplete=off` in the HTML password input field.

**Web Browser XSS Protection Not Enabled** : X-XSS-Protection is a value set by the web server in the HTTP response header to allow modern browsers to enable XSS protection feature. This setting is disabled in default Django settings. To fix this is necessary to specify `SECURE_BROWSER_XSS_FILTER = True` in the `settings.py` file.

**X-Content-Type-Options Header Missing** : another response header value, this one clarifies if the client browser is allowed to sniff MIME content or should only render the content based on the content-type declared. This option was introduced to prevent MIME abuse attacks, where malicious content would be uploaded on a site, and tricking the site visitors to render it in a different way from the official content type of the website, thus executing malicious code. Django keeps this option disabled, and considering that Linkero does not allow users to upload any content, it represent a very low risk. In any case, this can be fixed by declaring `SECURE_CONTENT_TYPE_NOSNIFF = True` in the `settings.py` file.

**Incomplete or No Cache-control and Pragma HTTP Header Set** : these response header values prevent the client browser from caching the page content. In this case the fix relies on setting the `@never-cache` decorator on the web-app page that I do not want the user to cache. This has to be done on every page.

**Cookie Without Secure Flag** : session cookies make sure that the user that is contacting the server is the same that originally authenticated during a login event, and was not replaced by another malicious actor after authentication. This is only possible if sessions are not intercepted and read by man-in-the-middle attacks. Django does not set the Secure flag by default because not every site need user authentication and not every site has SSL enabled. To fix this issue is sufficient to declare `SESSION_COOKIE_SECURE = True` in `settings.py` file.

**Cookie No HttpOnly Flag** : finally, this header controls cookie access by JavaScript.

In a public forum, where un-vetted code can be uploaded by anyone, a malicious actor could upload some JavaScript code designed to read other clients' cookies, stealing their sessions. Linkero restricts significantly the users ability to upload code, and there are limited options to share content. On the other end, part of Linkero's client JQuery script needs to access clients' cookies to append CSRF tokens to AJAX requests. I decided to keep this option disabled, considering the risk near to null and the benefits of improved usability that AJAX requests provide.

A second scan, after the necessary fixes were put in place, returned only a **Cookie No HttpOnly Flag** issue, which is something I expected and decided to accept.

## 4.4 Deployment

The deployment is relatively simple, since the platform was developed on the same virtual server that will be employed in production mode.

The Django application framework can run in *developer* or in *production* mode. The first mode is designed to return detailed traceback reports, whenever the application generates an error. This is very useful during development stage, but it is a clear security concern if left enabled, since error messages are public.

The other change required before deployment relates to the passwords that were created during the testing and development stages, and hard-encoded in the `settings.py` file. The entire Django application and its settings were cyclically uploaded on GitHub™. I do not have a private GitHub account, therefore every information, including passwords used during development are publicly visible at <https://github.com/srichiardi/linkero>. That means that the passwords to access MariaDB and MongoDB need to be changed. Django also provides a default `SECRET_KEY`, a 50 chars long alphanumeric string used for:

- providing unique “recover my account” URLs, or other one-time access pages;
- ensure that data in hidden form fields has not been tampered with [Fun18a].

The `SECRET_KEY` was stored just like any other password in the `settings.py` file, as such it needs to be replaced by a new one by editing directly the file stored on the actual virtual machine serving the site.





## Chapter 5

# Evaluation

### 5.1 Lessons learned

Configuring third party applications can be as daunting and time consuming than developing your own code, and it is definitely more frustrating when for some unknown reason individual configurations do not seem to work.

The Django framework is supposed to make web-developers' work quick and easy, unless it is the first time you use it: steep learning curve.

Django's ORM is a convenient way to interact with a relational database, keeping the code consistent and clean, as long as there are no advanced table joins needed, in that case SQL is a much simpler language.

I aspired to apply the principles of the SOLID method, but that remained in large part an aspiration. I realized that, given the time constraints, having a working system was the priority over clean and elegant coding practices.

Testing and breaking your own code is emotionally challenging, but a test-driven development, such as the approach recommended in the Extreme Programming methodology, is a way to circumvent this emotional attachment, and to design applications based on what they should do as well as how they could fail, which is an aspect that could otherwise be over-sought.

Estimating in advance the time required to complete each part of the project has been very difficult, especially when learning time needed to be factored in, the best strategy seemed to be to put as much hours as early as possible, rather than trying to distribute evenly the work throughout a period of 12 months.

## 5.2 Future work

Linkero is fully operational and is currently being used by 3 people on a weekly basis. Nonetheless, in its current implementation, it is a limited platform, especially for two principal aspects:

- there is no text search feature to retrieve historical data that is currently available in the local database;
- external data gathering ability is limited to eBay data.

The feature that should be implemented first is a search engine to take advantage of MongoDB's text searching capabilities. The reason for this is that, storing historical data only makes sense if it is possible to consume it afterward. Once the data is extracted and stored from external sources, it is possible to interrogate it in many different ways that are not always available through third party API. For instance, extremely useful is to see if one string (e.g. an email address) is

## 5.3 Conclusions

# Bibliography

- [Baz15] Michael Bazzell. *Open Source Intelligence Techniques*. Amazon.co.uk Ltd, 2015.
- [Ber93] E. Berard. *Essays on Object-Oriented Software Engineering*. Addison-Wesley, 1993.
- [Ber15] Stewart K. Bertrom. *The Tao of Open Source Intelligence*. IT Governance Publishing, 2015.
- [Bos00] J. Bosch. *Design & Use of Software Architectures*. Addison-Wesley, 2000.
- [Bra18] CSC Digital Brand. Brand monitoring, 2018. accessed on 28 December 2018.
- [Cel09] Project Celery. Celery, May 2009. last version 5.0.2 on 22 November 2018.
- [Che03] W. Cheswick. *Firewalls and Internet Security*. Addison-Wesley, 2003.
- [CR17] Shane Fuller Chad Russell. *Handbook on European data protection law*. John Wiley & Sons Ltd, 2017.
- [Deb18] Debian. Debian linux gnu, November 2018. last accessed on 26 November 2018.
- [Fun18a] Django Software Foundation. Cryptographic signing, 2018. accessed on 28 December 2018.
- [Fun18b] Django Software Foundation. Django web framework, November 2018. last accessed on 26 November 2018.
- [Gra87] R. B. Grady. *Software metrics: establishing a company wide program*. Prentice Hall, 1987.

- [GT09] P. Garcia-Teodoro. *Anomaly-Based Network Intrusion Detection*. Computer & Security, Vol. 28, 2009.
- [Hay07] Joseph Hayes. *Analytic Culture in the U.S. Intelligence Community*. Ed. Center for the Study of Intelligence, 2007.
- [Haz95] Philip Hazel. Exim, 1995. last version 4 on 15 April 2018.
- [Ja04] Cyril Jaquier. Fail2ban, 2004. last version 0.10.3.1 on 4 April 2018.
- [Lam04] B. Lampson. *Computer Security in the Real World*. Computer, 2004.
- [Man97] Theo Mandel. *The elements of user interface design*. Wiley, 1997.
- [Mar03] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2003.
- [Mat96] R. Mattison. *Data Warehousing: Strategies, Technologies and Techniques*. McGraw Hill, 1996.
- [Mey78] G. Meyers. *Composite Structured Design*. Van Nostrand, 1978.
- [Net98] NetFilter. Iptables, 1998. last version 1.8.2 on 13 November 2018.
- [Ngi04] Project Nginx. Nginx, October 2004. last version 1.14.1 on 6 November 2018.
- [oC18] US Department of Commerce. Quarterly retail e-commerce sales. *US Census Bureau News*, 2018.
- [OEC16] EUIPO OECD. *Trade in Counterfeit and Pirated Goods: Mapping the Economic Impact*. OECD Publishing, Paris, 2016.
- [oECtECohRE18] The Council of Europe (CoE) and the European Court of Human Rights (ECtHR). *GDPR for Dummies*. Publications Office of the European Union, Luxembourg, 2018.
- [otC15] Presidency of the Council. Proposal for a regulation of the European parliament and of the council on the protection of individuals with regard to the processing of personal data and on the free movement of such data (general data protection regulation). *Official Journal of the European Union*, 2015.
- [OWA17] OWASP. Owasp-zap, 2017. last version 2.7.0 on 28 November 2017.

- [Par72] D. L Parnas. On criteria to use in decomposing systems into modules. *CACM*, 14(1):221–227, 1972.
- [Pip00] Donald Pipkin. *Information security: Protecting the global enterprise*. New York: Hewlett-Packard Company, 2000.
- [Pre05] Roger S. Pressman. *Software engineering*. McGraw-Hill, 2005.
- [Sal09] Sanfilippo Salvatore. Redis, May 2009. last version 5.0.2 on 22 November 2018.
- [Shi00] R. Shirey. *RFC 2828 Internet Security Glossary*. Internet Engineering Task Force, 2000.
- [Tec18] Technopedia. Brand monitoring, 2018. accessed on 28 December 2018.
- [Ubi18] Ubit. uwsgi, November 2018. last accessed on 26 November 2018.
- [Vlio8] Hans Van Vliet. *Software Engineering, principles and practice*. John Wiley and Sons, 2008.
- [Wat18] Watchdog. Watchdog app, 2018. accessed on 28 December 2018.
- [Wik18a] Wikipedia. Nginx, November 2018. last edited on 26 November 2018.
- [Wik18b] Wikipedia. Nginx, November 2018. last edited on 26 November 2018.
- [Wir71] N. Wirth. Program development by stepwise refinement. *CACM*, 14(4):221–227, 1971.
- [Woo10] T. Wood. *Disaster Recovery as a Cloud Service Economy Benefits*. Proceedings USELINUX HotCloud 10, 2010.
- [WS15] Lawrence Brown William Stallings. *Computer Security, principles and practice*. Pearson Education Ltd, 2015.