CROWDSOURCING AND MANAGEMENT OF NATURE OBSERVATIONAL DATA

By

Giannis Skevakis

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE
IN ELECTRONIC & COMPUTER ENGINEERING

at the

SCHOOL OF ELECTRONIC & COMPUTER ENGINEERING
TECHNICAL UNIVERSITY OF CRETE

2014

Dedicated to my family

Abstract

Observations of plants and animals in nature is highly valued information for the experts in the area of biodiversity. They can be used to define changes in the population of animals, plants, or track their movements throughout long periods of time. Moreover, the richer the information following the observations, the more knowledge can be extracted from them. However, the limited number of experts and the limited funding in the area, makes the observation gathering procedure almost impossible. We present the design and implementation of a framework for the management of biodiversity observations captured by users roaming in the nature. This aims to alleviate the need for experts capturing biodiversity information, and propagates the collection of information to simple users wandering in the nature. Our framework consists of a model supporting the observations, and an infrastructure that allows the capturing, enrichment and storage of the observations using state-of-the-art technologies. Our architecture provides a scalable, highly efficient management of the collected data. The collection of the observational data is performed in real-time using mobile devices that most of the people have available with them, like mobile phones and tablets. Additionally, we describe the meta-model that we have defined, allowing the personalization of the metadata that follow the observations. This provides our framework with the freedom and extensibility needed so as to be implemented for various domains other than biodiversity. Finally, we describe the process of migrating the data collected by the Natural Europe project to our infrastructure.

Acknowledgements

I would like to thank my supervisor, Prof. Stavros Christodoulakis, for his encouragement and his continuous guidance and support throughout my research. I would like also to thank him for the important experiences he offered me during my stay at the Lab. of Distributed Multimedia Information Systems and Applications (TUC/MUSIC), as well as for the positive influence in broadening my horizons.

Special thanks go to Konstantinos Makris for his continuous support, his valuable advices, as well as the long fruitful discussions.

I would also like to express my gratitude to the readers of this thesis Prof. Antonios Deligiannakis and Prof. Aikaterini Mania for the time they devoted and their critical evaluation and helpful comments on my thesis.

My appreciation goes to Polyxeni Arapi, Chrisa Tsinaraki, Nektarios Gioldasis and Fotis Kazasis for being always ready to offer their help whenever needed. Furthermore, I would like to thank all my colleagues in the Laboratory for their support and for the pleasant environment they provided.

Last but not least, I would like to thank my family for their never ending love, support, and their patience with me especially over the past few months. Without them, many of my dreams would not be possible.

Giannis Skevakis
Technical University of Crete
April 2014

Publications

Part of the work that is included in this thesis, or carried out during my M.Sc. studies has been published in the following journals and conference proceedings:

- **Skevakis G.**, Tsinaraki C., Trochatou I., Christodoulakis S.:*"A Crowdsourcing Framework for the Management of Mobile Multimedia Nature Observations"*, International Journal on Mobile Information Systems, 2014.
- Tsinaraki C., **Skevakis G.**, Trochatou I., Christodoulakis S.: *"MoM-NOCS: Management of Mobile Multimedia Nature Observations using Crowd Sourcing"*. Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia, Vienna, Austria, November 2013.

- **Skevakis G.**, Makris K., Kalokyri V., Arapi P., Christodoulakis S.: *"Metadata Management, Interoperability and Linked Data Publishing Support for Natural History Museums"*. International Journal on Digital Libraries (IJDL), 2014.
- Makris K., **Skevakis G.**, Kalokyri V., Arapi P., Christodoulakis S., Stoitsis J., Manolis N., Rojas S. L.: *"Federating Natural History Museums in Natural Europe"*. Proceedings of the 7th Metadata and Semantics Research Conference, Special track on Metadata & Semantics for Cultural Collections & Applications (MTSR '13), Thessaloniki, Greece, November 2013. *Best Student Paper Award.*
- Makris K., **Skevakis G.**, Kalokyri V., Arapi P., Christodoulakis S.: *"Metadata Management and Interoperability Support for Natural History Museums"*. Proceedings of the 17th International Conference on Theory and Practice of Digital Libraries (TPDL '13), Valletta, Malta, September 2013. *Best Student Paper Nomination.*
- **Skevakis G.**, Makris K., Arapi P., Christodoulakis S.: *"Elevating Natural History Museums' Cultural Collections to the Linked Data Cloud"*. Proceedings of the 3rd International Workshop on Semantic Digital Archives (SDA '13), Valletta, Malta, September 2013.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Observations of plants and animals in nature is a very rich source of information for the experts in the context of biodiversity. They can be used to define changes in the population of animals, plants, or track their movements throughout long periods of time. However, the limited number of experts and the limited funding in the area, makes the observation gathering procedure almost impossible.

The same need for observations along with the same problems extends to other contexts as well. Examples to these could be observations about weather conditions, reports of the damages from earthquakes, reports of car accidents, reports of fungus appearance on crops, etc. All these fields would be greatly helped by the mass cataloging of information which could later be useful for analyzing and inferring new knowledge.

Studying large-scale patterns in nature requires a vast amount of data to be collected across an array of locations and habitats over spans of years or even decades. One way to obtain such data is through *citizen science*, a research technique that enlists the public in gathering scientific information [19]. Large-scale projects can engage participants in continental or even global data-gathering networks. Pooled data can be analyzed to illuminate population trends, range changes, and shifts in phenologies.

Crowdsourcing is the new era in citizen-science, allowing the collaborative resolution of tough problems and the distillation of new knowledge. Coined by Jeff Howe and Mark Robinson [43], the term crowdsourcing describes a new web-based business model that harnesses the creative solutions of a distributed network of individuals through what amounts to an open call for proposals. In other words, a company posts a problem online, a vast number of individuals (the ‘crowd’) offer solutions to the problem, the winning ideas are awarded some form of a bounty, and the company mass-produces the idea for its own gain [21].

To understand the workings of crowdsourcing, it is best to examine some of the most successful and profitable cases in a variety of industries. Notable business examples of crowdsourcing include the t-shirt company Threadless.com [1], microstock photography agency iStockphoto.com [20, 44], corporate research and development clearing-house InnoCentive.com [2].

More than just an online business model, crowdsourcing is also a problem solving model that can have profound influence in the way we solve our world's most pressing social and environmental problems, a sentiment increasingly echoed by others. In that spirit, the business model of crowdsourcing is already being applied in non-profit, government projects.

The most recent non-profit crowdsourcing event in global scale is the search for the wreckage of flight MH370 of the Malaysia Airlines. The initiative for this act was taken by Tomnod[3]. Tomnod provides the users with recent satellite images of the sea around Malaysia, and the users try to identify objects in the ocean that could be parts of the airplane. A similar initiative taken by Tomnod in the past was a search and rescue operation when a light airplane with a crew of five went missing over the Idaho wilderness. The Tomnod crowd rallied to search satellite images for signs of the crash, which was found a few days later.

Studies [73] have shown that users change their established practices upon adopting widely used crowdsourcing systems because the additional effort supports individual satisfaction and community recognition. This dramatically increases the value of the data for research, promoting improved scientific outcomes.

The advance of computers and the Internet to global scale and the rise of people involved in online communities gave vast amount of power to crowdsourcing. Mobile devices have been increasing in numbers ever since they were introduced to the public. Especially during the last 5 years and due to the advances in mobile computing, mobile phones and tablets have become more powerful to the extend that they have started to replace computers in lots of routine tasks (e.g. checking email, browsing, reading books). The current numbers and the future projections of mobile devices have forced more and more companies to move towards mobile applications, available instantly to the users wherever they are.

In this thesis, we present the design and implementation of a framework for the management of observations made by users at real time. The observations that we support follow a flexible meta-model, allowing our framework to support various domains, e.g. biodiversity observations about species, observations about natural disasters like earthquakes, observations about car accidents. The collection of the observational data is performed using mobile devices that most of the people have available with them, like mobile phones and tablet devices. Moreover, our technical platform has

[1]`https://www.threadless.com`
[2]`http://www.innocentive.com`
[3]`http://www.tomnod.com/nod/`

been build with efficiency in mind, providing a scalable, highly efficient system for the management of the collected data.

In more detail, the aim and objectives of this thesis are to: (*a*) *define the model* needed to represent observations made by users, (*b*) *define the metamodel (application profile)* providing extensibility to the models so as to be able to support different contexts, (*c*) *design and develop an infrastructure* supporting the model and the application profiles, (*d*) *design and develop an application* supporting the creation of observations conforming to the model and the application profile, (*e*) *develop web services* enabling the exploitation of the collected data from external systems

1.1 Summary of Contributions

In this thesis, we present a framework for the management of observations made by users at real time. In more detail, we propose a model for describing observations and a meta-model for personalizing this model according to the user needs. The model refers to the basic information concerning the creation of observations along with the captured multimedia objects. The meta-model refers to the model that we have developed, enabling the creation of application profiles concerning the description of the observations. This allows the parametrization of the model depending on the context, by defining the information that is collected about each observation.

Additionally to the model and meta-model, we have developed a technical platform to support them. The implemented platform is comprised of two parts, the backend system and the client applications. The backend system is the core of the infrastructure, holding all the data of the system and performing all the business logic while providing services to the client side as well as external systems. The client applications are the graphical user interfaces that the users interact with during the creation and management of the observations. The whole technical platform has been build using state-of-the-art web technologies, and the applications provided include a native mobile application compatible with all the major mobile platforms.

Our novel infrastructure enables the creation of external applications for the exploitation of the collected data, as well as systems that are able to generate and provide data, e.g. sensors capturing data and creating observations.

Finally, we present the process of migrating the data collected during the Natural Europe project to our new infrastructure. This process is extensible and can be applied to other systems as well.

1.2 Reader's Guide

Apart from the introduction, the preliminaries, the related work, and the conclusion, this thesis can be divided into three parts. In the first part, we define the model and the meta-model for the observations. In the second part, we describe the infrastructure that we developed in order to support the model and the meta-model, and in the third part we present the use interfaces that we developed for the interaction between our systems and the users. More precisely, this thesis is structured as follows:

- Chapter 2 presents the systems and research that are most relevant to the issues addressed in this thesis.
- Chapter 3 provides a brief overview of the technologies used for the implementation of the systems. These include the state of the art advances in developing web applications (Javascript), web applications/services (Spring) and modern database systems (MongoDB).
- Chapter 4 presents the Natural Europe project on which we have worked for the most duration of this thesis and was the main influence for our research.
- Chapter 5 describes the functional specification of the system that has been developed for the management of the whole lifecycle of the observations and their metadata.
- Chapter 6 specifies the model and the meta-model for describing observations. The model refers to the basic information concerning the creation of observations along with the captured multimedia objects. The meta-model refers to the model that we have developed, enabling the creation of application profiles concerning the description of the observations.
- Chapter 7 presents the architecture designed and implemented for the infrastructure.
- Chapter 8 presents the implementation details of some of the components in more detail, providing some more insight as to how specific parts of the system have been implemented.
- Chapter 9 presents the User Interfaces that have been developed for the interaction with the user.
- Chapter 10 presents the process and the software that we have developed in order to achieve the transition of the data collected from the Natural Europe project to our infrastructure.

Chapter 2

Related Work

In what follows, we present products and research that are most relevant to the issues addressed in our work. For each of the systems described below, we include the features that are closest to our framework and we add screenshots of the user interfaces whenever possible. Furthermore, we compare them to our framework, identifying the advantages and disadvantages that they pose.

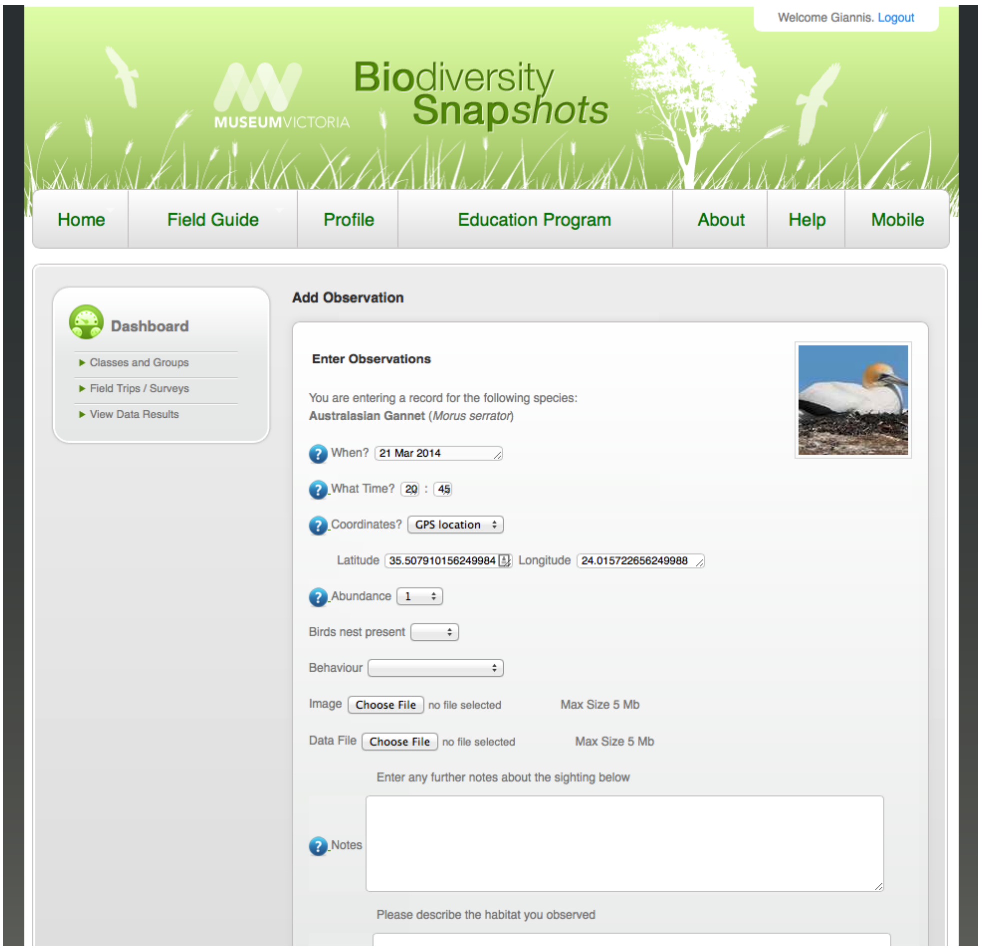
2.1 FieldData

FieldData[1] (also known as the BDRS, Biological Data Recording System) was developed by Gaia Resources on behalf of the Atlas of Living Australia (see Section 2.4) to help individuals, researchers, community groups and natural resource management groups collect and manage biodiversity data. For the citizen scientist, it provides a means to contribute sightings, photos and other files to a project and to then see and edit their records.

It's features include the definition of the species that the observations are to be made, as well as the parametrization of the presentation of them. Additionally, the administrators have the freedom to define the application profile used for the descriptions of the observations - that is the metadata elements that the users need to complete for each observation that they document. Figure 2.1 presents the user interface for the creation of a new observation. On the center of the screen we find the metadata that the administrator has defined for the new observations and the user needs to fill.

All the occurrences are presented either as a list or on a map for better visualization, while it is also possible to generate reports on the data and online surveys. Finally, the interface is fully customizable, in order to provide different look and feel depending on the needs of the organizations hosting it. There are also certain procedures that allow the collected data to be shared with the Atlas.

[1] http://www.ala.org.au/get-involved/citizen-science/fielddata-software/



Add Observation

Enter Observations

You are entering a record for the following species:
Australasian Gannet (*Morus serrator*)

When? 21 Mar 2014

What Time? 20 : 45

Coordinates? GPS location

Latitude 35.507910156249984 Longitude 24.015722656249988

Abundance 1

Birds nest present

Behaviour

Image Choose File no file selected Max Size 5 Mb

Data File Choose File no file selected Max Size 5 Mb

Enter any further notes about the sighting below

Notes

Please describe the habitat you observed

Figure 2.1: FieldData - Creation of observation using the predefined application profile.

Some of the uses of FieldData are described below:

Citizen scientists can use the software to contribute something meaningful to scientific research on biodiversity in Australia, discover new things about the environment you live in or see your data online and compare it to what others are contributing.

Naturalist groups can use the software to easily build forms for recording observations that are then published online, enable members to login and participate in online surveys or quickly generate maps and reports from the data collected by your members.

Environmental educators can use the software to engage with a wider audience using the web, easily build recording forms and publish them online, create species pages to provide your volunteers with meaningful information on species or add simple identification tags to species to allow people to dynamically identify species.

Researchers and scientists can use the software to easily build recording forms and publish

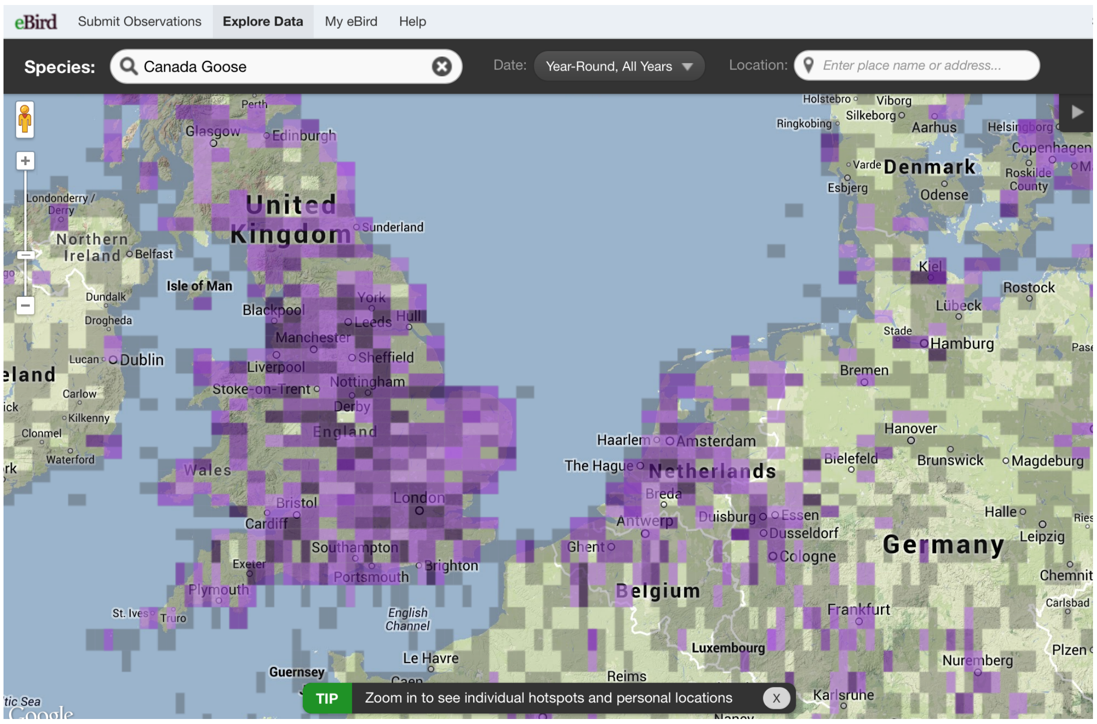

eBird
Submit Observations
Explore Data
My eBird
Help
Species: Canada Goose
Date: Year-Round, All Years
Location: Enter place name or address...
TIP Zoom in to see individual hotspots and personal locations

Figure 2.2: eBird observations on maps.

them online, allow interested people to register and start recording their observations or extract information for further analysis.

FieldData is very close to our framework in terms of the model and the application profile support. However it only provides a web interface for the creation of observations, so the users will have a lot of troubles trying to create the observations in real time using their mobile devices. That is they will need to have a big device since the interfaces are not optimized for mobile phones, and additionally they will need internet access. Moreover, FieldData is bound to the biodiversity context since the observations are connected with predefined species, while our application profiles can be extended to other contexts as well by defining the profiles of the objects.

2.2 eBird

eBird[2] [45, 74] is a real-time, online checklist program. eBird has revolutionized the way that the birding community reports and accesses information about birds. Launched in 2002 by the Cornell Lab of Ornithology and National Audubon Society, eBird provides rich data sources for basic information on bird abundance and distribution at a variety of spatial and temporal scales.

[2]`http://ebird.org/content/ebird/`

eBird's goal is to maximize the utility and accessibility of the vast numbers of bird observations made each year by recreational and professional bird watchers. It is amassing one of the largest and fastest growing biodiversity data resources in existence. For example, in March 2012, participants reported more than 3.1 million bird observations across North America.

The observations of each participant join those of others in an international network of eBird users. eBird then shares these observations with a global community of educators, land managers, ornithologists, and conservation biologists. In time these data will become the foundation for a better understanding of bird distribution across the western hemisphere and beyond.

The way that eBird works is that it documents the presence or absence of species, as well as bird abundance through checklist data. A simple web-interface allows users to submit their observations or view the observations of others. eBird encourages users to participate by providing Internet tools that maintain their personal bird records and enable them to visualize data on top of maps. All the features are available in English, Spanish, and French.

Additionally, eBird collects observations from birders through portals managed and maintained by local partner conservation organizations. This way eBird targets specific audiences with the highest level of local expertise, promotion, and project ownership. Portals may have a regional focus or they may have more specific goals and/or specific methodologies. Each eBird portal is fully integrated within the eBird database and application infrastructure.

Comparing eBird with our framework, one can easily realize the fact that eBird only supports observations about birds. This means that it has predefined fields for the information that the users can supply, since their major concern is the place and the time of the sightings. Moreover, eBird does not offer mobile device support and capturing of multimedia.

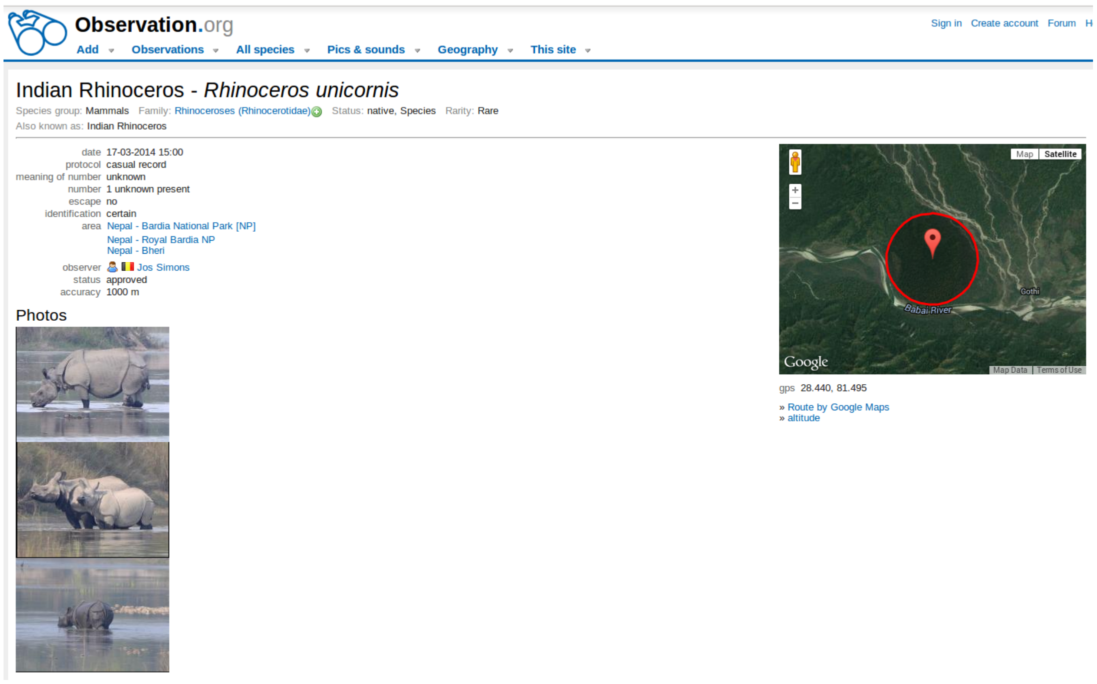
2.3 Observation.org

Observation.org[3] is a web information system that enables the creation of biodiversity observations. It's objective is to provide a central point of contact for nature minded people and biodiversity data. All the data collected is available to the public, and in real time, enabling the instant exploitation of new observations. Observation.org's long term goal is to cover all species groups.

Observation.org cooperates with local and national workgroups and provides them with their personalized interfaces, so these groups can collect and view local data. Observation.org pays a lot of attention to the quality of the data. To achieve this, the data is checked daily by a number of volunteers, who spend many hours checking the data and most of them have expertise in certain

[3] `http://observation.org/`

species groups.



Indian Rhinoceros - *Rhinoceros unicornis*

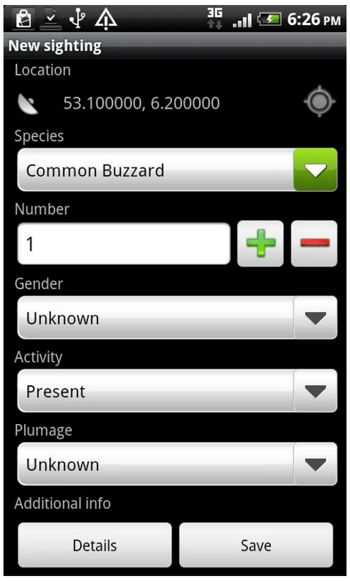
Species group: Mammals Family: Rhinoceroses (Rhinocerotidae) Status: native, Species Rarity: Rare
Also known as: Indian Rhinoceros

date	17-03-2014 15:00
protocol	casual record
meaning of number	unknown
number	1 unknown present
escape	no
identification	certain
area	Nepal - Bardia National Park [NP] Nepal - Royal Bardia NP Nepal - Bheri
observer	Jos Simons
status	approved
accuracy	1000 m

gps 28.440, 81.495

» Route by Google Maps
» altitude

Figure 2.3: Observation.org - Sample observation of a herd of Rhinoceros.



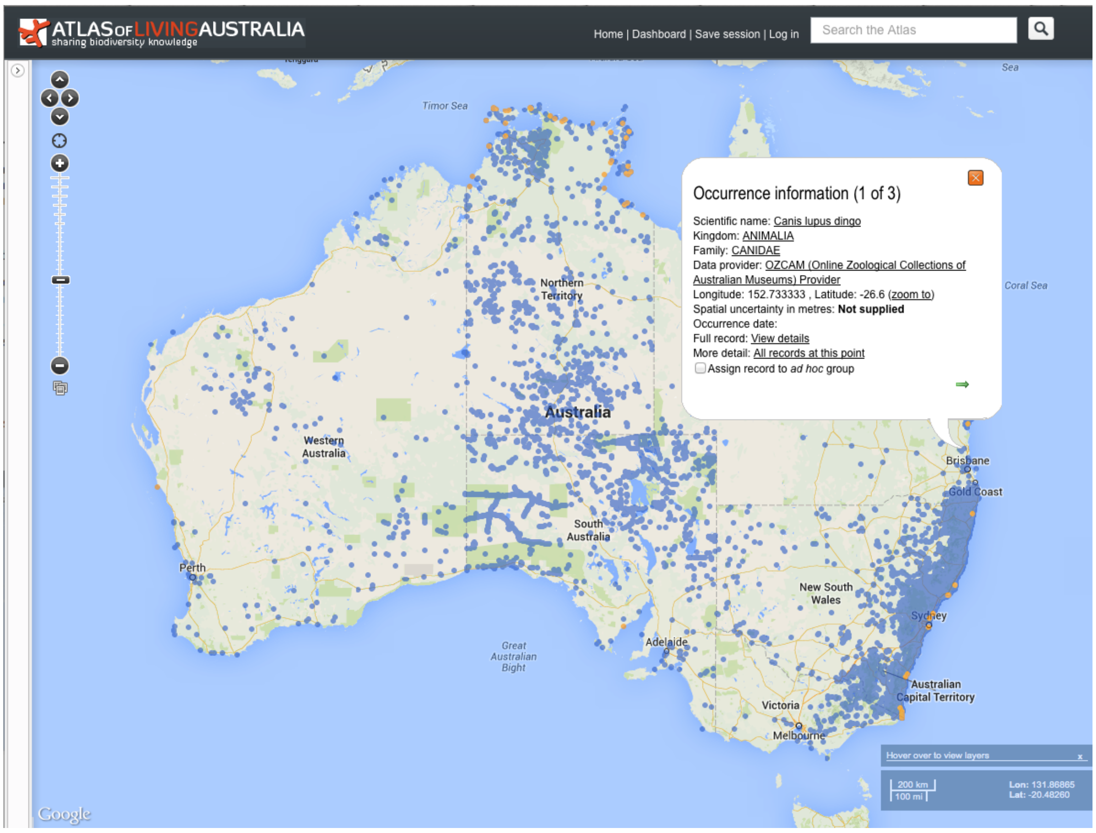
Location	53.100000, 6.200000
Species	Common Buzzard
Number	1
Gender	Unknown
Activity	Present
Plumage	Unknown
Additional info	

Figure 2.4: Observation.org - Sample observation of a herd of Rhinoceros.

Observation.org is another system very close to our framework in terms of the functionality. Although it provides a mobile application for Android and iOS devices, it does not support the realtime capturing of observations or the offline usage. Moreover, the model is predefined and the same for all the species, whether plants or animals, which forces very poor metadata collection along the observations.

2.4 Atlas of Living Australia

The Atlas of Living Australia[4] contains information on all the known species in Australia, aggregated from a wide range of data providers: museums, herbaria, community groups, government departments, individuals and universities. The number of providers supported so far is around 500.



Occurrence information (1 of 3)

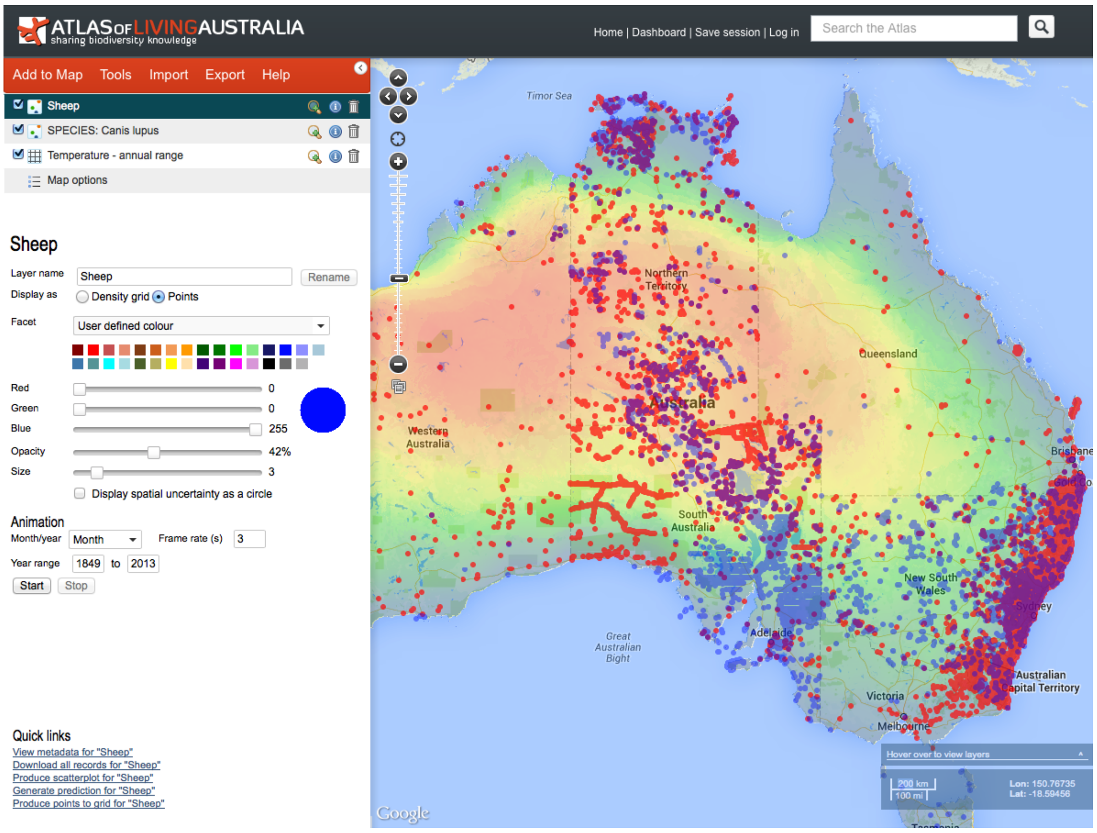
Scientific name: Canis lupus dingo
Kingdom: ANIMALIA
Family: CANIDAE
Data provider: OZCAM (Online Zoological Collections of Australian Museums) Provider
Longitude: 152.733333 , Latitude: -26.6 (zoom to)
Spatial uncertainty in metres: **Not supplied**
Occurrence date:
Full record: View details
More detail: All records at this point
☐ Assign record to *ad hoc* group

Figure 2.5: Atlas of Living Australia - Visualization of occurrences for the species 'Canis Lupus'.

The data of the Atlas include information about species and observations. The catalog of species is very comprehensive, containing the full taxonomic classification, common names from various sources, extensive gallery, as well as literature references. This data is aggregated from multiple sources around Australia, like museums and archives, as well as from individuals that want to contribute their sightings. Figure 2.5 shows the visualization tool containing the map of Australia with the observations about the species "Canis Lupus". We can also see the data of a single oc-

[4]`http://www.ala.org.au/`

currence, describing the scientific name, the taxonomic classification, the data provider, the spatial information, etc.



Sheep

Layer name: Sheep

Display as: Density grid [✓ Points]

Facet: User defined colour

Red: 0
Green: 0
Blue: 255
Opacity: 42%
Size: 3

Display spatial uncertainty as a circle

Animation

Month/year: Month   Frame rate (s): 3

Year range: 1849 to 2013

Quick links

View metadata for "Sheep"
Download all records for "Sheep"
Produce scatterplot for "Sheep"
Generate prediction for "Sheep"
Produce points to grid for "Sheep"

Figure 2.6: Atlas of Living Australia - Complex visualization of occurrences for the species 'Canis Lupus' (red dots), along with the Sheep (blue dots) and average temperature data.

The available data for visualization on the maps does not only cover the occurrences of the species, but it also extends to data about temperature, humidity, etc. Figure 2.6 is an example of the visualization tool holding three different layers. The first one is the occurrences of wolfs "Canis Lupus", shown with red dots. The second layers contains occurrences of sheep, depicted by the blue dots. Finally, the third layer shows the average temperature throughout the year in the whole continent.

The Atlas of Living Australia collects data from a number of providers. The harvesting is performed using the Darwin Core format, which means that the provides need to export their data in Darwin Core before it is usable by the system. Moreover, the system exposes services to the outside world, making the data accessible and exploitable by external systems. One of these systems is GBIF [5], which aggregates the data along with data from other systems around the world.

[5] `http://www.gbif.org`

Compared to our framework, although the Atlas of Living Australia provides great visualization interfaces for biodiversity observations, it only aggregates data from official sources (museums, clubs, etc.), thus it does not allow simple users to submit their observations.

Summary

In this section we presented the systems and research that are most relevant to the issues addressed in our work. For each of these systems we described the features that are closest to our framework and we compared them to our framework.

Chapter 3

Background

This chapter presents a brief overview of the standards and technologies used in this thesis. Section 3.1 describes The Spring Framework, the core of the back-end system. Section 3.2 presents MongoDB, the document-oriented databased system used for persisting the data. Section 3.4 presents JavaScript, the scripting language used mainly for the implementation of the client side logic and the interaction with the users, as well as the JavaScript libraries used. Section 3.5 presents Elasticsearch, the system used for indexing and searching the data. Section 3.6 presents Xuggle, the library used for analyzing and manipulating video and audio files. Finally, Section 3.7 presents Redis, the system used for in-memory cache management, speeding up various aspects of the system.

3.1 The Spring Framework

The Spring Framework [9] is an open source application framework and inversion of control container for the Java platform. It provides a comprehensive programming and configuration model for modern Java-based enterprise applications on any kind of deployment platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform. Although the framework does not impose any specific programming model, it has become popular in the Java community as an alternative to, replacement for, or even addition to the Enterprise JavaBean (EJB) model.

Modules

The Spring Framework includes a number of different modules that provide a wide range of functionality and services. Although the list of modules is extensive, we will describe the ones used in this work.

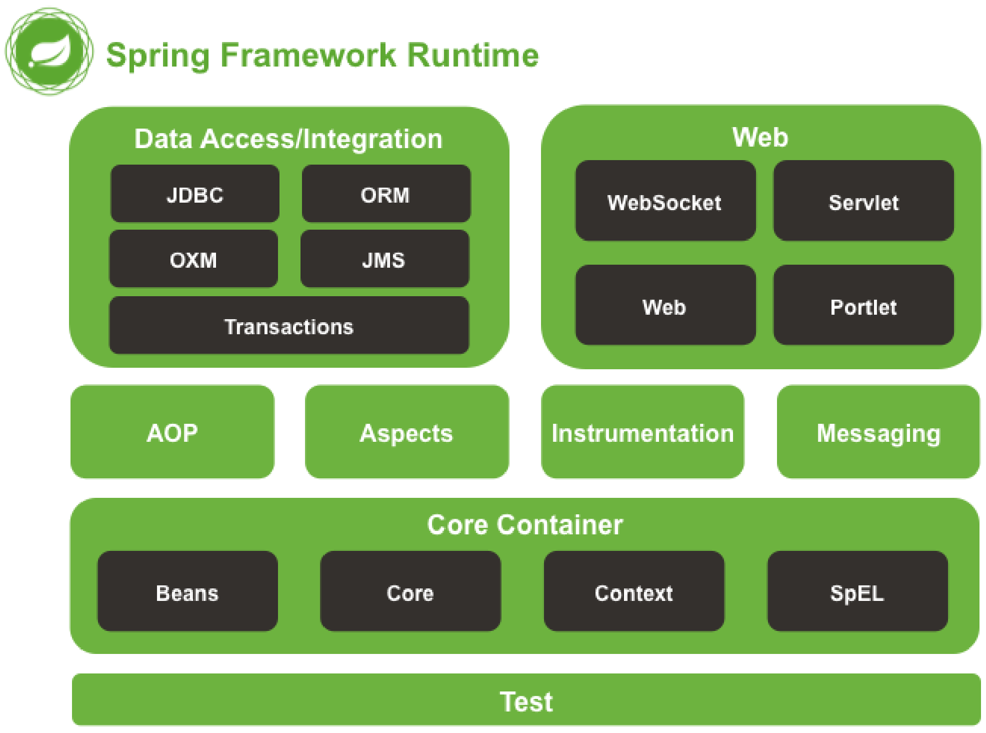


Figure 3.1: Overview of the Spring Framework.

Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules.

The Core and Beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The Context module builds on the solid base provided by the Core and Beans modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module also supports Java EE features such as EJB, JMX ,and basic remoting. The ApplicationContext interface is the focal point of the Context module.

The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules. The *JDBC module* provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes. The *ORM module* provides integration layers for popular object-relational mapping APIs, including JPA, JDO, and Hibernate. Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously. The *OXM module* provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream. The *Java Messaging Service (JMS) module* contains features for producing and consuming messages. The *Transaction module* supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs (plain old Java objects).

Web

The Web layer consists of the Web, Web-Servlet, WebSocket and Web-Portlet modules. *Spring's Web module* provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring's remoting support. The *Web-Servlet module* contains Spring's model-view-controller (MVC) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework. The *Web-Portlet module* provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Test

The Test module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

Inversion of control container

Applications that are build with the Java language typically consist of objects that collaborate to form the application proper. Thus the objects in an application have dependencies on each other.

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers. This drove the development and usage of popular design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* in order to compose the various classes and object instances that make up an application. However, these patterns are simply best practices, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth. Patterns are formalized best practices that the developer must implement himself in his application.

The *Spring Framework Inversion of Control (IoC) component* addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s). Numerous organizations and institutions use the Spring Framework in this manner to engineer robust, maintainable applications.

Central to the Spring Framework Inversion of Control (IoC) component is the inversion of control (IoC) container. The container can manage the whole lifecycle of the objects. Objects created by the container are also called managed objects or beans. The container can be configured by loading XML files or detecting specific Java annotations on configuration classes. These data sources contain the bean definitions which provide the information required to create the beans.

Objects can be obtained by means of either dependency lookup or dependency injection. Dependency lookup is a pattern where a caller asks the container object for an object with a specific name or of a specific type. Dependency injection is a pattern where the container passes objects by name to other objects, via either constructors, properties, orfactory methods.

In many cases one need not use the container when using other parts of the Spring Framework, although using it will likely make an application easier to configure and customize. The Spring container provides a consistent mechanism to configure applications and integrates with almost all Java environments, from small-scale applications to large enterprise applications.

Data access framework

Spring's data access framework [39] addresses common difficulties developers face when working with databases in applications. Support is provided for all popular data access frameworks in Java: JDBC, iBatis/MyBatis, Hibernate, JDO, JPA, Oracle TopLink, Apache OJB, and Apache Cayenne, among others.

For all of these supported frameworks, Spring provides these features:

- Resource management - automatically acquiring and releasing database resources.
- Exception handling - translating data access related exception to a Spring data access hierarchy.
- Transaction participation - transparent participation in ongoing transactions.
- Resource unwrapping - retrieving database objects from connection pool wrappers.
- Abstraction for BLOB and CLOB handling.

All these features become available when using template classes provided by Spring for each supported framework. Critics have said these template classes are intrusive and offer no advantage over using (for example) the Hibernate API directly. In response, the Spring developers have made it possible to use the Hibernate and JPA APIs directly. This however requires transparent transaction management, as application code no longer assumes the responsibility to obtain and close database resources, and does not support exception translation.

Together with Spring's transaction management, its data access framework offers a flexible abstraction for working with data access frameworks. The Spring Framework doesn't offer a common data access API; instead, the full power of the supported APIs is kept intact. The Spring Framework is the only framework available in Java which offers managed data access environments outside of an application server or container.

3.2 MongoDB

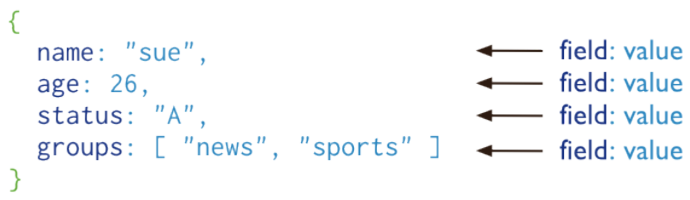
MongoDB [7, 23] is a cross-platform document-oriented database system. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON [2]), making the integration of data in certain types of applications easier and faster. Released under a combination of the GNU Affero General Public License and the Apache License, MongoDB is free and open source software.

First developed by the software company 10gen (now MongoDB Inc.) in October 2007 as a component of a planned platform as a service product, the company shifted to an open source development model in 2009, with 10gen offering commercial support and other services. Since then, MongoDB has been adopted as backend software by a number of major websites and services, including Craigslist, eBay, Foursquare, SourceForge, and The New York Times, among others.

The following is a brief summary of some of the main features:

Document-Oriented Storage

Data in MongoDB has a flexible schema. Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections do not enforce document structure. This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the data has substantial variation. In practice, however, the documents in a collection share a similar structure.



```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

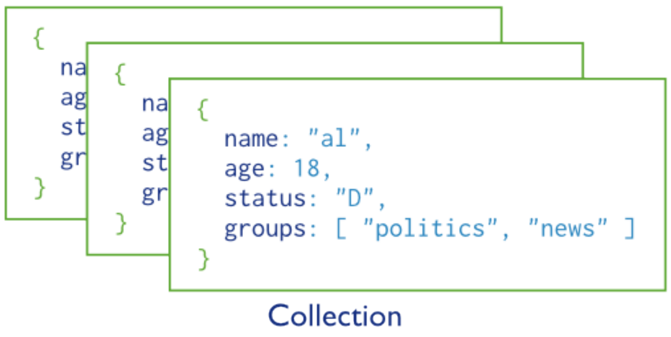
Figure 3.2: A MongoDB document with 4 fields and values of types text, number and list.

MongoDB stores data in the form of documents, which are JSON-like field and value pairs. Documents are analogous to structures in programming languages that associate keys with values, where keys may hold other pairs of keys and values (e.g. dictionaries, hashes, maps, and associative arrays). Formally, MongoDB documents are BSON documents, which is a binary representation of JSON with additional type information. Figure 3.2 depicts a sample MongoDB document containing four fields: (*a*) a *name* with textual content, (*b*) an *age* with numeric content, (*c*) a *status* with textual content, (*d*) a list of *groups* with textual content

MongoDB stores all documents in collections. A collection is a group of related documents that have a set of shared common indexes. Collections are analogous to a table in relational databases. Figure 3.3 depicts a collection of the previous described document.

Full Index Support

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement.



```
{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

Figure 3.3: A MongoDB collection of documents.

These collection scans are inefficient because they require MongoDB to process a larger volume of data than an index for each operation.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection. Any field in a MongoDB document can be indexed and secondary indices are also available.

If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect. In some cases, MongoDB can use the data from the index to determine which documents match a query.

Replication

Replication is the process of synchronizing data across multiple servers. Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate a server to disaster recovery, reporting, or backup.

In some cases, replication can be used to increase read capacity. Clients have the ability to send read operations to different servers. You can also maintain copies in different data centers to increase the locality and availability of data for distributed applications.

MongoDB provides high availability and increased throughput with replica sets. A replica set consists of two or more copies of the data. Each replica may act in the role of primary or secondary replica at any time. The primary replica performs all writes and reads by default. Secondary replicas maintain a copy of the data on the primary using built-in replication. When a primary replica fails, the replica set automatically conducts an election process to determine which secondary should become the primary. Secondaries can also perform read operations, but the data is eventually consistent by default.

A replica set is a group of MongoDB instances that host the same data set. One MongoDB instance, the primary, receives all write operations. All other instances are secondary instances and apply operations from the primary so that they have the same data set. Figure 3.4 presents the interaction between the MongoDB instances in a replica set.

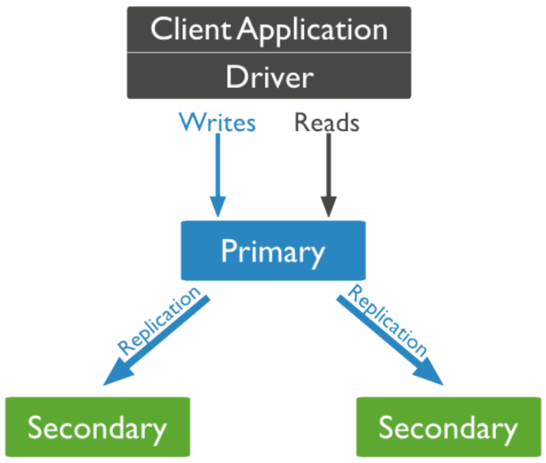


Figure 3.4: Default routing of reads and writes to the primary.

The primary accepts all write operations from clients. Replica sets can have one and only one primary at any given moment. Because only one member can accept write operations, replica sets provide strict consistency. To support replication, the primary records all changes to its data sets in its operation log.

The secondaries replicate the primary's operation log and apply the operations to their data sets. Secondaries' data sets reflect the primary's data set. If the primary is unavailable, the replica set will elect a secondary to be primary. By default, clients read from the primary; however, clients can specify a read preference to send read operations to secondaries. See secondaries for more information.

Sharding

Sharding is the process of storing data records across multiple machines and is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations. Figure 3.4 presents an example of sharding a large collection across 4 shards.

Database systems with large data sets and high throughput applications can challenge the capacity of a single server. High query rates can exhaust the CPU capacity of the server. Larger data sets exceed the storage capacity of a single machine. Finally, working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

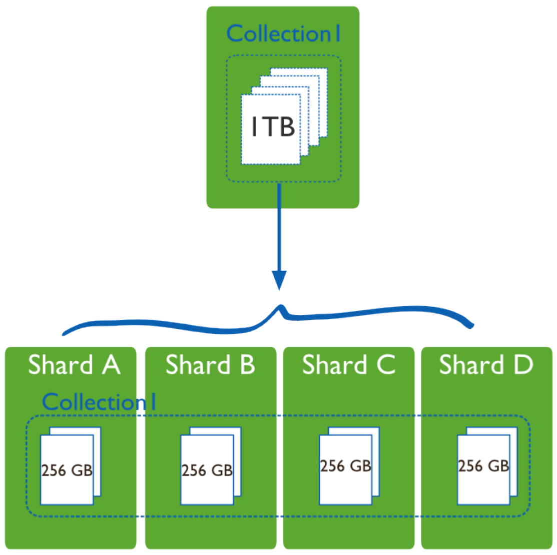


Figure 3.5: Large collection with data distributed across 4 shards.

Sharding addresses the challenge of scaling to support high throughput and large data sets:

- Sharding reduces the number of operations handled by each shard. Each shard processes fewer operations as the cluster grows. As a result, shared clusters can increase capacity and throughput horizontally. For example, to insert data, the application only needs to access the shard responsible for that records.
- Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows. For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data.

MongoDB distributes data, or shards, at the collection level. Sharding partitions a collection's data by the shard key. A shard key is either an indexed field or an indexed compound field that exists in every document in the collection. MongoDB divides the shard key values into chunks and distributes the chunks evenly across the shards. To divide the shard key values into chunks, MongoDB uses either range based partitioning and hash based partitioning.

Querying

In MongoDB a query targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a projection that specifies the fields from the matching documents to return. You can optionally modify queries to impose limits, skips, and sort orders.

An example query and the evaluation process can be seen in Figure 3.6:

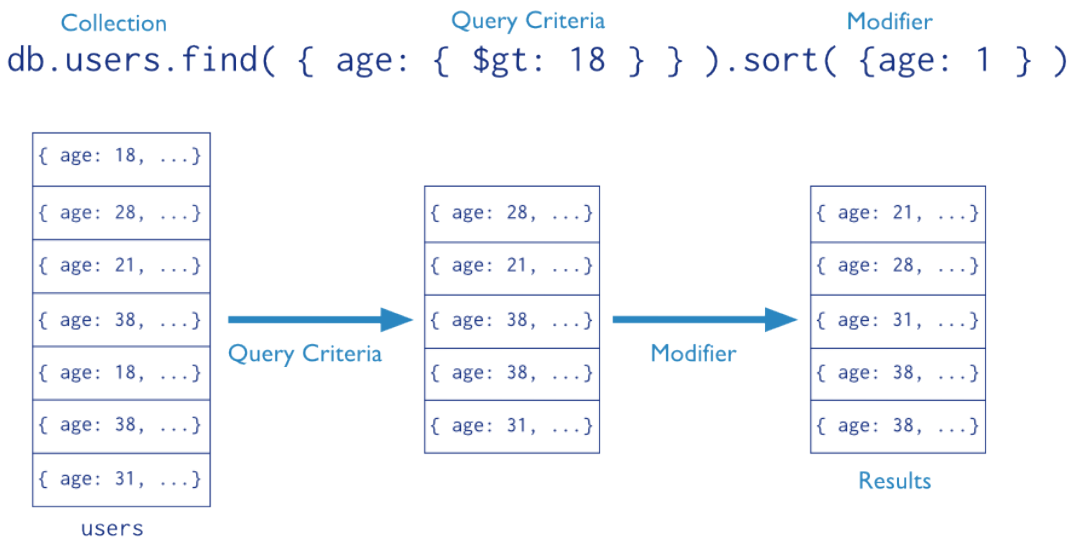


Figure 3.6: The stages of a MongoDB query with a query criteria and a sort modifier.

File storage

MongoDB can be used as a file system, taking advantage of load balancing and data replication features over multiple machines for storing files. The specification created for this purpose is called GridFS and it is used for storing and retrieving files that exceed the BSON-document size limit of 16MB. GridFS is included with MongoDB drivers and available with no difficulty for development languages.

MongoDB exposes functions for file manipulation and content to developers. Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, and stores each of those chunks as a separate document. By default GridFS limits chunk size to 256k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

In a multi-machine MongoDB system, files can be distributed and copied multiple times between machines transparently, thus effectively creating a load-balanced and fault-tolerant system. When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed.

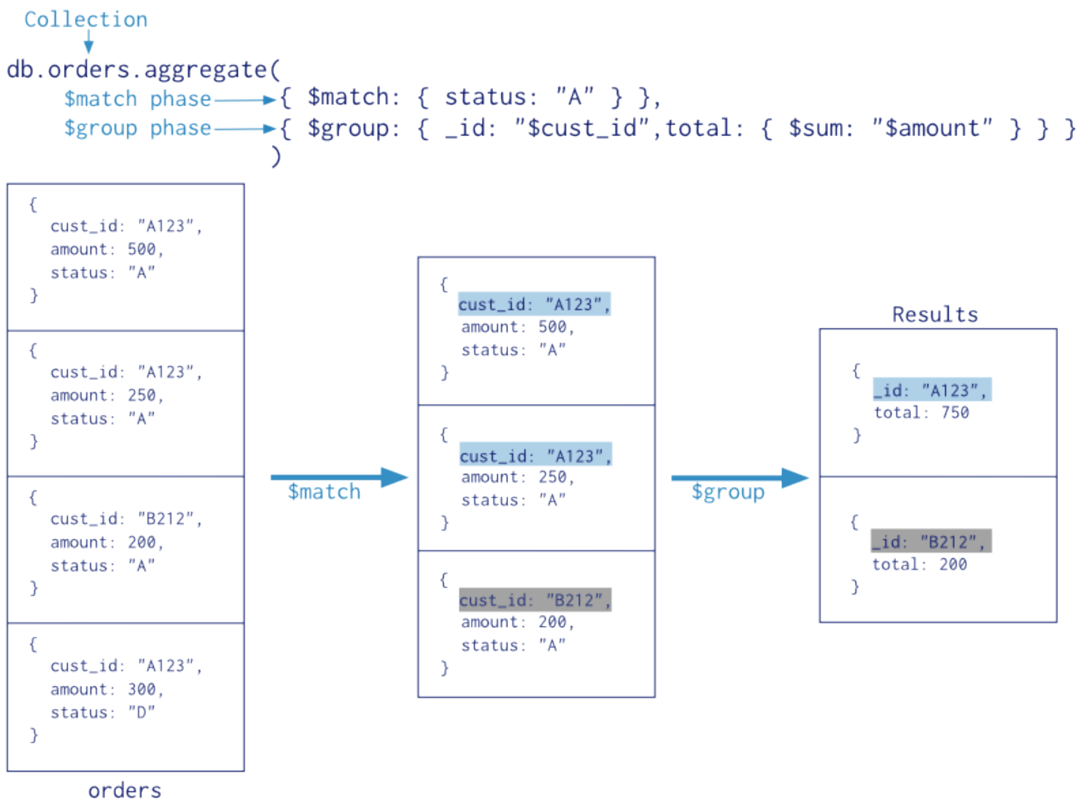
You can perform range queries on files stored through GridFS. You also can access information from arbitrary sections of files, which allows you to “skip” into the middle of a video or audio file.

GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory.

Aggregation

Aggregations are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Running data aggregation on the MongoDB instance simplifies application code and limits resource requirements.

Like queries, aggregation operations in MongoDB use collections of documents as an input and return results in the form of one or more documents.



```
db.orders.aggregate(
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
)
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{
  _id: "A123",
  total: 750
}
{
  _id: "B212",
  total: 200
}
```

Figure 3.7: Sample aggregation pipeline operation.

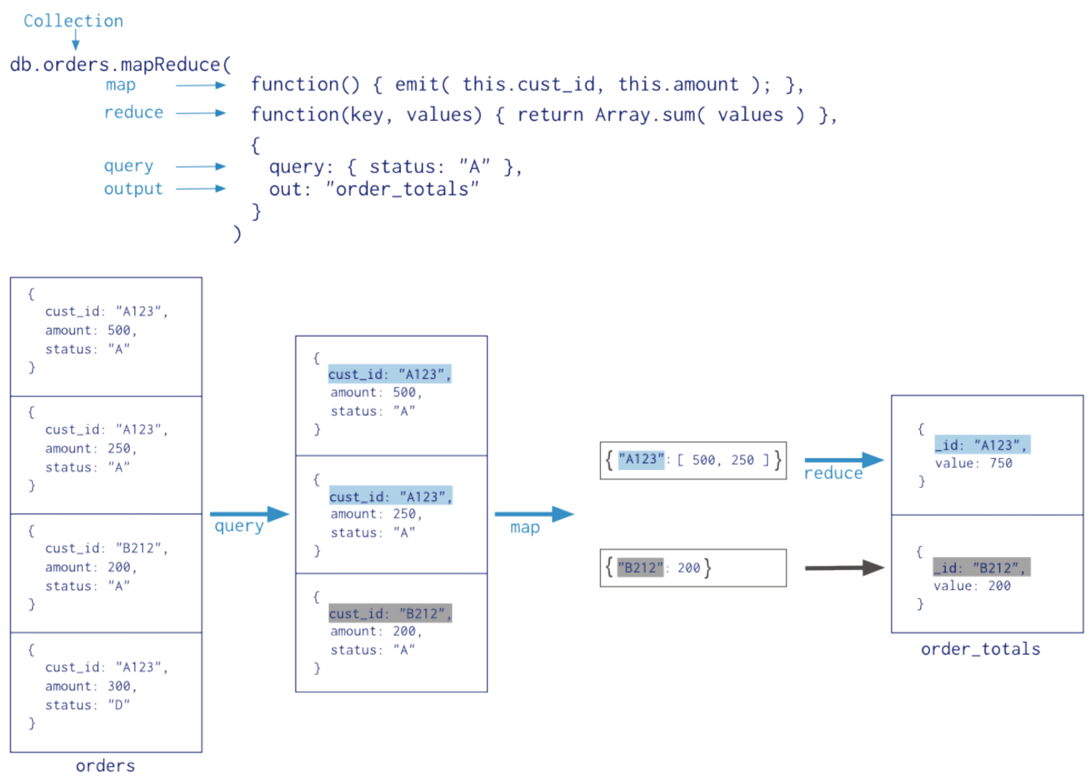
Aggregation Pipelines

Documents enter a multi-stage pipeline that transforms the documents into an aggregated result. The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document. An example aggregation pipeline operation is presented in Figure 3.7.

Map/Reduce

Map-reduce [27] is a data processing paradigm for condensing large volumes of data into useful aggregated results. For map-reduce operations, MongoDB provides the mapReduce database command.

Consider the map-reduce operation shown in Figure 3.8:



```
db.orders.mapReduce(
                 function() { emit( this.cust_id, this.amount ); },
                 function(key, values) { return Array.sum( values ) },
                 {
                   query: { status: "A" },
                   out: "order_totals"
                 }
               )
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
{ "A123": [ 500, 250 ] }
{ "B212": 200 }
```

```
{
  _id: "A123",
  value: 750
}
{
  _id: "B212",
  value: 200
}
```

Figure 3.8: Sample map-reduce operation.

In this map-reduce operation, MongoDB applies the map phase to each input document (i.e. the documents in the collection that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the reduce phase, which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a finalize function to further condense or process the results of the aggregation.

All map-reduce functions in MongoDB are JavaScript and run within the MongoDB process. Map-reduce operations take the documents of a single collection as the input and can perform any arbitrary sorting and limiting before beginning the map stage. Map-Reduce can return the results of a map-reduce operation as a document, or may write the results to collections. The input and the output collections may be sharded.

3.3 Phonegap

Phonegap is an web-based mobile development framework, based on the open-source Cordova project. PhoneGap allows the use of standard web technologies such as HTML5, CSS3, and JavaScript for cross-platform development, avoiding each mobile platforms' native development language. Applications execute within wrappers targeted to each platform, and rely on standards-compliant API bindings to access each device's sensors, data, and network status.

Examples of developers that can benefit from the use of PhoneGap are:

- a mobile developer who wants to extend an application across more than one platform, without having to re-implement it with each platform's language and tool set.
- a web developer who wants to deploy a web app that's packaged for distribution in various app store portals.
- a mobile developer interested in mixing native application components with a WebView (browser window) that can access device-level APIs, or wants to develop a plugin interface between native and WebView components.

Basic Components

PhoneGap applications rely on a common config.xml file that provides information about the app and specifies parameters affecting how it works, such as whether it responds to orientation shifts. This file adheres to the W3C's Packaged Web App, or widget, specification.

The application itself is implemented as a web page, named index.html by default, that references whatever CSS, JavaScript, images, media files, or other resources are necessary for it to run. The app executes as a WebView within the native application wrapper, which you distribute to app stores. For the web app to interact with various device features the way native apps do, it must also reference a phonegap.js file, which provides API bindings. The PhoneGap-enabled WebView may provide the application with its entire user interface. It can also be a component within a larger, hybrid application that mixes the WebView with native application components. PhoneGap provides a plugin interface for these components to communicate with each other.

Development Paths

The easiest way to set up an application is to run the phonegap command-line utility, also known as the command-line interface (CLI). Depending on the set of platforms that the developer wants to target, he can rely on the CLI for progressively greater shares of the development cycle:

In the most basic scenario, the CLI is simply used to create a new project that is populated with default configuration for the developer to modify. Once a mobile platform's SDK is installed, the applications can be compiled locally.

Adobe has also introduced PhoneGap Build server, allowing the developer to upload the source code, while the system takes care of the compilation process in various platforms.

For many mobile platforms, the CLI can also be used to set up additional project files required to compile within each SDK. For this to work, each targeted platform's SDK needs to be installed For the supporting platforms, the CLI can compile executible applications and run them in an SDK-based device emulator. For comprehensive testing, one can also generate application files and install them directly on a device.

At any point in the development cycle, the developer can also rely on platform-specific SDK tools, which may provide a richer set of options. An SDK environment is more appropriate for the implementation of a hybrid app that mixes web-based and native application components.

3.4 JavaScript

JavaScript is a dynamic computer programming language released by Netscape and Sun Microsystems in 1995. It is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed. It is also being used in server-side programming, game development and the creation of desktop and mobile applications.

JavaScript is a prototype-based scripting language with dynamic typing and has first-class functions. It copies many names and naming conventions from *Java*, but the two languages are otherwise unrelated and have very different semantics. The key design principles within JavaScript are taken from the *Self* and *Scheme* programming languages. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

JavaScript has also significant support outside of web pages for example, in PDF documents, site-specific browsers, and desktop widgets. Newer and faster JavaScript VMs and platforms built upon them (notably *Node.js*) have also increased the popularity of JavaScript for server-side web applications. On the client side, JavaScript was traditionally implemented as an interpreted language but just-in-time compilation is now performed by recent browsers.

JavaScript was formalized in the ECMAScript [32] language standard and is primarily used as part of a web browser (client-side JavaScript). This enables programmatic access to computational objects within a host environment. As of 2011, the latest version of the language is JavaScript 1.8.5. It is a superset of ECMAScript (ECMA-262) Edition 3. The following sections highlight the most important features of JavaScript.

Dynamic typing

As in most scripting languages, types are associated with values, not with variables. For example, a variable `x` could be bound to a number, then later rebound to a string.

Object-based

JavaScript is almost entirely object-based. JavaScript objects are associative arrays, augmented with prototypes (see below). Object property names are string keys. They support two equivalent syntaxes: dot notation (obj.x = 10) and bracket notation (obj['x'] = 10). Properties and their values can be added, changed, or deleted at run-time. Most properties of an object (and those on its prototype inheritance chain) can be enumerated using a for...in loop. JavaScript has a small number of built-in objects such as Function and Date.

Run-time evaluation

JavaScript includes an `eval` function that can execute statements provided as strings at run-time.

First-class functions

Functions in JavaScript are first-class citizens. They are complete objects themselves. As such, they have properties and methods, such as `call()` and `bind()`. JavaScript also supports nested functions. A nested function is a function that is defined within another function. It is created each time the outer function is invoked. In addition, each created function forms a closure: the scope of the outer function, including any constants, local variables and argument values, becomes part of the internal state of each inner function object, even after execution of the outer function concludes.

Prototypes

JavaScript uses prototypes where many other object oriented languages use classes for inheritance. It is possible to simulate many class-based features with prototypes in JavaScript [54].

Functions as object constructors

Functions double as object constructors along with their typical role. Prefixing a function call with new will create an instance of a prototype, inheriting properties and methods from the constructor (including properties from the Object prototype). ECMAScript 5 offers the Object.create method, allowing explicit creation of an instance without automatically inheriting from the Object prototype (older environments can assign the prototype to null). The constructor's prototype property determines the object used for the new object's internal prototype. New methods can be added by modifying the prototype of the object used as a constructor. JavaScript's built-in constructors, such `asArray` or `Object`, also have prototypes that can be modified. While it is possible to modify the Object prototype, it is generally considered bad practice because most objects in JavaScript will inherit methods and properties from the Object prototype and they may not expect the prototype to be modified.

Functions as methods

Unlike many object-oriented languages, there is no distinction between a function definition and a method definition. Rather, the distinction occurs during function calling; when a function is called as a method of an object, the function's local this keyword is bound to that object for that invocation.

Type Composition and Inheritance

Whereas explicit function based delegation does cover composition in JavaScript, implicit delegation already happens every time the prototype chain is walked in order to e.g. find a method that might be related to but is not directly owned by an object. Once the method was found it gets called within this object's context. Thus inheritance in JavaScript is covered by a delegation automatism that is bound to the prototype property of constructor functions.

Run-time environment

JavaScript typically relies on a run-time environment (e.g. a web browser) to provide objects and methods by which scripts can interact with the environment (e.g. a webpage DOM). It also relies on the run-time environment to provide the ability to include/import scripts (e.g. HTML `<script>`

elements). This is not a language feature per se, but it is common in most JavaScript implementations.

Variadic functions

An indefinite number of parameters can be passed to a function. The function can access them through formal parameters and also through the local arguments object. Variadic functions can also be created by using the apply method.

Array and object literals

Like many scripting languages, arrays and objects (associative arrays in other languages) can each be created with a succinct shortcut syntax. In fact, these literals form the basis of the JSON data format.

Regular expressions

JavaScript also supports regular expressions in a manner similar to Perl, which provide a concise and powerful syntax for text manipulation that is more sophisticated than the built-in string functions.

Asynchronous JavaScript and XML (AJAX)

In 1990's user interaction in web applications was request-wait-response based, which slowed down the user interaction considerably. The most web sites were based on complete HTML pages where each user action required that the page should be re-loaded from the server. Each time a page was reloaded due to a partial change, all of the content was re-sent instead of only the changed information. This placed additional load on the server and use of excessive bandwidth. Asynchronous JavaScript and XML (AJAX) [71, 38] came as a boon to the web application development, providing mechanisms for user experience similar to desktop applications.

In the classic web application model, addressed as pre-AJAX web model, user interaction triggers an HTTP [33] request to the web server. The server performs necessary processing for example, retrieving data or doing some calculations etc. When the processing is completed the server returns an HTML [18] page to the client. The problem is that, during the server processing time, the user can do nothing but wait for a page to be loaded or refreshed from the server.

AJAX increases the web page's interactivity, speed, and usability in order to provide richer user experience. AJAX places an AJAX engine between the client and server. This engine is written in JavaScript and behaves like a hidden frame. The AJAX engine renders the user interface and handles the communication between client and server. The client-server communication with AJAX is asynchronous. Asynchronous communication means the client does not need to wait for the server response. After sending the request to the server the execution in the client program does not halt, rather the execution is continued. The response is sent to the client when it is available. The AJAX engine sends requests to the server on behalf of the client and receives data or responses from the server. In a web model with AJAX, the server sends small data instead of the HTML page. The AJAX engine shows the received data or response by updating the page partially. Thus user is free to do other interactions after sending a request to the server.

3.5 Elasticsearch

Elasticsearch [3] is a flexible and powerful open source, distributed, real-time search and analytics engine. Architectured from the ground up for use in distributed environments where reliability and scalability are must haves, it provides much more powerful functionality than simple full-text search. Through its robust set of APIs and query DSLs, plus clients for the most popular programming languages, Elasticsearch delivers on the near limitless promises of search technology.

Elasticsearch is a search server based on Lucene. It provides a distributed, multitenant-capable full-text search engine with a RESTful web interface and schema-free JSON documents. It is developed in Java and is released as open source under the terms of the Apache License. The first version was released by Shay Banon on February 2010.

Elasticsearch can be used to search all kinds of documents. It provides scalable search, has near real-time search, and supports multitenancy. Elasticsearch is distributed, which means that indices can be divided into shards and each shard can have zero or more replicas. Each node hosts one or more shards, and acts as a coordinator to delegate operations to the correct shard(s). Rebalancing and routing are done automatically.

It is based on Lucene for the persistence of the data and tries to make all its features available through the JSON and Java API. It supports faceting and percolating, which can be useful for notifying if new documents match for registered queries.

Another feature is called ‘Gateway’ and handles the long term persistence of the index- i.e. an index can be recovered from the Gateway in a case of a server crash. Elasticsearch supports real-time GET requests, which makes it suitable as a NoSQL solution, but it lacks distributed transactions.

3.6 Xuggler

Xuggler [11] is a Java library that allows developers to easily uncompress, modify, and re-compress any media file or stream. It is built on top of the FFMPEG [1] and is provided under the Lesser GNU Public License. We have used it in our systems in order to manipulate video and audio files and produce thumbnails.

3.7 Redis

Redis [8] is an open-source, networked, in-memory, key-value data store with optional durability. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets and sorted sets. Redis supports atomic operations on these types, like appending to a string; incrementing the value in a hash; pushing to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set.

In order to achieve its outstanding performance, Redis works with an in-memory dataset. Depending on your use case, you can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log. Redis also supports trivial-to-setup master-slave replication, with very fast non-blocking first synchronization, auto-reconnection on net split and so forth. Other features include Transactions, Pub/Sub, Lua scripting, Keys with a limited time-to-live, and configuration settings to make Redis behave like a cache.

Finally, Redis is available for the most programming languages.

[1]http://ffmpeg.org/

Chapter 4

The Natural Europe project

This chapter presents the Natural Europe project, which was the main influence for this job. Having worked on designing and building the whole technical infrastructure for the project, we were able to identify the most important requirements in the context of biodiversity. Moreover, we realized the limitations of the supported model and the platform, which drove the identification of the new requirements that are presented and met in this thesis.

The Natural Europe project [10, 65, 50] offers a coordinated solution at European level that aims to overcome the aforementioned barriers, making the natural history heritage available to formal and informal learning processes. Its main objective is to improve the availability and relevance of environmental cultural content for education and life-long learning use, in a multilingual and multicultural context. Cultural heritage content related to natural history, natural sciences, and natural/environmental preservation is collected from six Natural History Museums around Europe into a federation of European Natural History Digital Libraries, directly connected with Europeana.

The Natural Europe project offers appropriate tools and services that allow the participating NHMs to: (*a*) uniformly describe and semantically annotate their content according to international standards and specifications, (*b*) interconnect their digital libraries, and (*c*) expose their Cultural Heritage Object (CHO) metadata records to Europeana.eu and BioCASE/GBIF.

The Biological Collection Access Service for Europe (BioCASE) [16] is a transnational network of biological collections of all kinds, while the Global Biodiversity Information Facility (GBIF) [5] is an open infrastructure which provides a single point of access to global biodiversity data.

This chapter describes the Natural Europe Cultural Environment, i.e., the infrastructure and toolset deployed on each NHM allowing their curators to publish, semantically describe, manage and disseminate the CHOs that they contribute to the project.

4.1 The Natural Europe Cultural Environment (NECE)

The *Natural Europe Cultural Environment (NECE)* [48] is a node in the cultural perspective of the Natural Europe project architecture [49]. It refers to the toolset deployed at each participating NHM, consisting of the Multimedia Authoring Tool (MMAT), the CHO Repository and the Vocabulary Server, facilitating the complete metadata management life-cycle: *ingestion*, *maintenance*, *curation*, and *dissemination* of CHO metadata. NECE also specifies how legacy metadata are migrated into Natural Europe. Figure 4.1 presents the architecture of Natural Europe with a focus on the Natural Europe Cultural Environment.

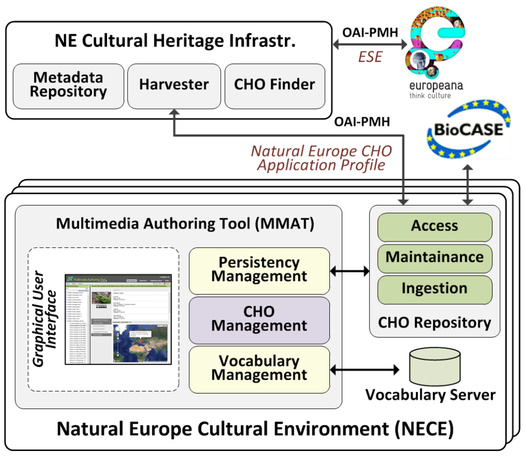


Figure 4.1: The Natural Europe Architecture.

In the Natural Europe context, the participating NHMs provide metadata descriptions about a large number of Natural History related CHOs. These descriptions are semantically enriched with Natural Europe shared knowledge (vocabularies, taxonomies, etc.) using project provided annotation tools and services. The enhanced metadata are aggregated by the project, harvested by Europeana (to become available through its portal) and exploited for educational purposes. Furthermore, they are exposed to the BioCASE/GBIF networks, contributing their high quality content to biodiversity communities.

The following sections present the Natural Europe CHO Application Profile, as well as the architectural components of NECE (i.e., Multimedia Authoring Tool, CHO Repository and Vocabulary Server), focusing on their internal functionality.

4.1.1 The Natural Europe CHO Application Profile

The *Natural Europe CHO Application Profile* is a superset of the Europeana Semantic Elements (ESE) [4] metadata format. It has been developed through an iterative process involving the NHMs' domain experts and the technical partners of the project, driven by the needs and requirements of the stakeholders and the application domain of the project. The Natural Europe CHO Application Profile describes 3 main element categories for each CHO.

The *Cultural Heritage Object (CHO)* metadata category provides information about the analog resource or born digital object. It is composed of the following sub-categories: (*a*) the *Basic information*, dealing with descriptive information about the Cultural Heritage Object, (*b*) the *Species information* is applicable to describe information related to the species of a described specimen (animals, plants, minerals, rocks, etc.), and (*c*) the *Geographical information* contains metadata about the location in which a specimen has been collected.

The *Digital Object* metadata category provides information about a digital or digitized resource. It contains the following sub-categories: (*a*) the *Basic information* deals with general descriptive information about a digital or digitized resource, (*b*) the *Content information* holds the physical characteristics and technical information exclusive to a digital or digitized resource, and (*c*) the *Rights information* describes the intellectual property rights and the accessibility to a digital or digitized resource.

The *Meta-metadata* category provides metadata information for a CHO record. These include the creator of the record, the languages that appear in the metadata, etc. Additionally, it describes the history of the record during its evolution in the MMAT, including the operations and entities that affected it.

4.1.2 The MultiMedia Authoring Tool (MMAT)

The *Multimedia Authoring Tool (MMAT)* [1] is the first step towards allowing the connection of digital collections with Europeana and BioCASE/GBIF. It is a multilingual web-based management system for museums, archives and digital collections, which facilitates the authoring and metadata enrichment of cultural heritage objects. MMAT establishes interoperability between NHMs, cultural heritage and biodiversity networks. Moreover, it supports seamless ingestion of legacy metadata.

MMAT supports a rich metadata element set, the Natural Europe CHO Application Profile, as well as a variety of the most popular multimedia formats. Its main features include the publication of multimedia objects, the semantic linkage of the described objects with well-established controlled vocabularies, and the real-time collaboration among end-users with concurrency control mechanisms. Additionally, it provides the means to directly import the museums' legacy metadata for further enrichment and supports various types of users with different access rights. The

[1]MMAT demo version: `http://natural-europe.tuc.gr/mmat`

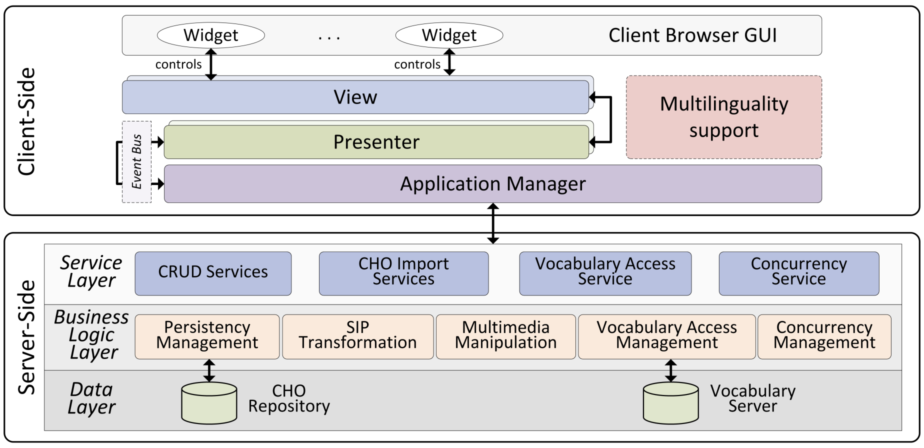
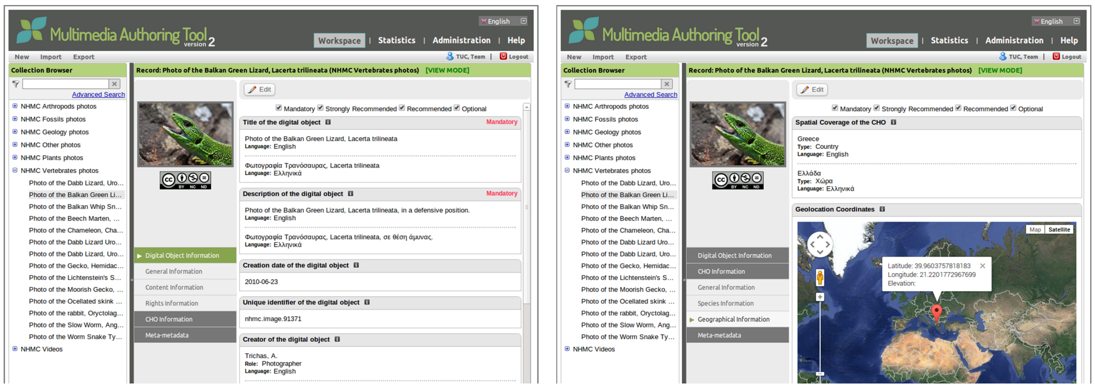


Figure 4.2: The Multimedia Authoring Tool Architecture.

three types of user that are currently supported are: (*a*) the administrators, able to manage the user accounts and the application, (*b*) the curators, administering CHO records/collections, and (*c*) the simple users, allowed only to inspect the data.

MMAT has been built as a Rich Internet Application, offering engaging experience and increased productivity. It adopts the Google Web Toolkit (GWT) [**?**] technology that enables web applications to perform part of their business logic into the client side and part on the server side. The client side refers to the business logic operations performed within a web browser running on a user's local computer, while the server side refers to the operations performed by a web server running on a remote machine. The overall architecture of MMAT is presented in Figure 4.2. Additionally, Figure 4.3 contain two screenshot of the tool.


Multimedia Authoring Tool version 2
English
Workspace | Statistics | Administration | Help
New Import Export
TUC, Team | Logout
Collection Browser
Advanced Search
NHMC Arthropods photos
NHMC Fossils photos
NHMC Geology photos
NHMC Other photos
NHMC Plants photos
NHMC Vertebrates photos
Photo of the Dabb Lizard, Uro...
Photo of the Balkan Green Li...
Photo of the Balkan Whip Sn...
Photo of the Beech Marten, ...
Photo of the Chameleon, Cha...
Photo of the Dabb Lizard Uro...
Photo of the Dabb Lizard, Uro...
Photo of the Gecko, Hemidac...
Photo of the Lichtenstein's S...
Photo of the Moorish Gecko, ...
Photo of the Ocellated skink ...
Photo of the rabbit, Oryctolag...
Photo of the Slow Worm, Ang...
Photo of the Worm Snake Ty...
NHMC Videos
Edit
Mandatory Strongly Recommended Recommended Optional
Digital Object Information
General Information
Content Information
Rights Information
CHO Information
Species Information
Geographical Information
Meta-metadata
Map
Satellite

Record: Photo of the Balkan Green Lizard, Lacerta trilineata (NHMC Vertebrates photos) [VIEW MODE]

Title of the digital object — Mandatory
Photo of the Balkan Green Lizard, Lacerta trilineata
Language: English
Φωτογραφία Τρανόσαυρας, Lacerta trilineata
Language: Ελληνικά

Description of the digital object — Mandatory
Photo of the Balkan Green Lizard, Lacerta trilineata, in a defensive position.
Language: English
Φωτογραφία Τρανόσαυρας, Lacerta trilineata, σε θέση άμυνας.
Language: Ελληνικά

Creation date of the digital object
2010-06-23

Unique identifier of the digital object
nhmc.image.91371

Creator of the digital object
Trichas, A.
Role: Photographer
Language: English

Spatial Coverage of the CHO
Greece
Type: Country
Language: English
Ελλάδα
Type: Χώρα
Language: Ελληνικά

Geolocation Coordinates
Latitude: 39.9603757818183
Longitude: 21.2201772967699
Elevation:

Figure 4.3: Screenshots presenting the graphical user interface of MMAT.

4.1.3 The CHO Repository

The CHO Repository handles both content and metadata and adopts the OAIS Reference Model [12] for the ingestion, maintenance and dissemination of Information Packages (IPs). To this end, it accommodates modules for the ingestion, archival, indexing, and accessing of CHOs, CHO records/ collections etc. This functionality refers to a complete information preservation lifecycle, where the producer is the MMAT and the consumers are the MMAT, the harvester application of the Natural Europe federal node and the BioCASE/GBIF networks. Figure 4.4 presents the overall architecture of the CHO Repository with emphasis to its internal software modules.

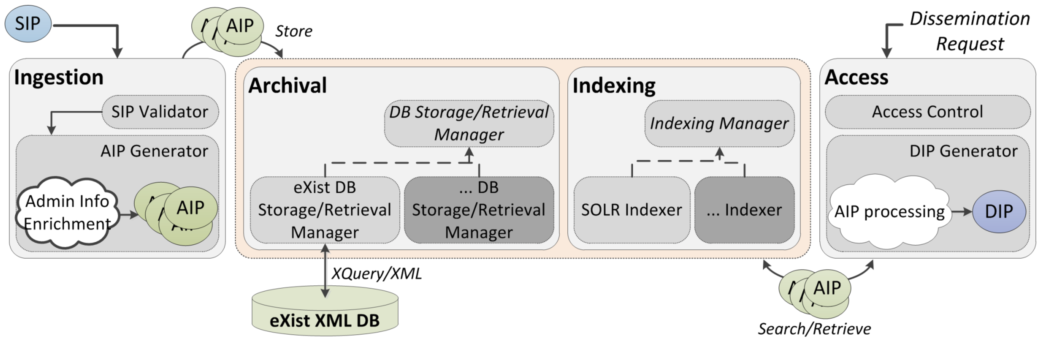


Figure 4.4: The CHO Repository Architecture.

The *Ingestion Module* is responsible for the ingestion of an information package (i.e., CHOs, CHO records, CHO collections, and user information) in order to store it as a new Archival Information Package (AIP) to the repository, or to update/delete an already existing AIP.

The *Archival Module* receives AIPs from the Ingestion Module for storage purposes, as well as AIP retrieval requests from the Access Module for dissemination purposes. In order to support storage and retrieval operations, it employs a DB Storage/Retrieval Manager component which is implemented in a flexible way for supporting any DBMS (relational, XML). A dedicated eXist DB Storage/Retrieval Manager has been implemented, supporting database specific storage and retrieval operations in an eXist XML DB instance, using XQuery/XML.

The *Indexing Module* receives AIPs from the Archival Module in order to build and maintain AIP index structures, as well as AIP retrieval requests from the Access Module for dissemination purposes.

The *Access Module* provides services allowing Dissemination Information Package (DIP) consumers (i.e., MMAT, the harvester application of the Natural Europe federal node, the BioCASE/ GBIF networks and other external applications) to request and receive information stored in the CHO Repository. It provides functionality for receiving information access requests, while applying access control policies through the Access Control component. Furthermore, it exploits any available indices maintained by the Indexing module in order to retrieve the requested AIPs.

4.1.4 The Vocabulary Server

The Vocabulary Server manages the vocabularies and authority files used during the semantic annotation process. Authority files refer to information about organizations, persons and places, while vocabularies refer to the taxonomies used for the enrichment of the CHO metadata. Vocabulary Server is also responsible for the indexing and retrieval of authority and taxonomic information, allowing us to provide fast auto-complete functionality to MMAT end users. This saves time from the curation process, increasing user productivity and providing error prevention during semantic annotation. For this purpose, the server employs a Lucene/Solr instance, managing the indexing and querying of data.

The semantic annotation of resources is a strong requirement for any system supporting metadata editing within a museum. Apart from the use of controlled vocabularies during the annotation process, it provides great cross-institution interoperability. To this end, the Vocabulary Server has been developed to support any taxonomic classification that the museums might use. This is achieved through the ingestion of taxonomies represented in SKOS format.

Simple Knowledge Organization System (SKOS) [52] is the leading format for the representation of thesauri, classification schemes, taxonomies, or any other type of controlled vocabularies. It is based on the Semantic Web principles and therefore enables the smooth transition of data to RDF. The exploitation of well-established SKOS vocabularies during the annotation process, provides a solid basis for the production of linked data, and subsequently an additional dissemination channel towards the Linked Data communities.

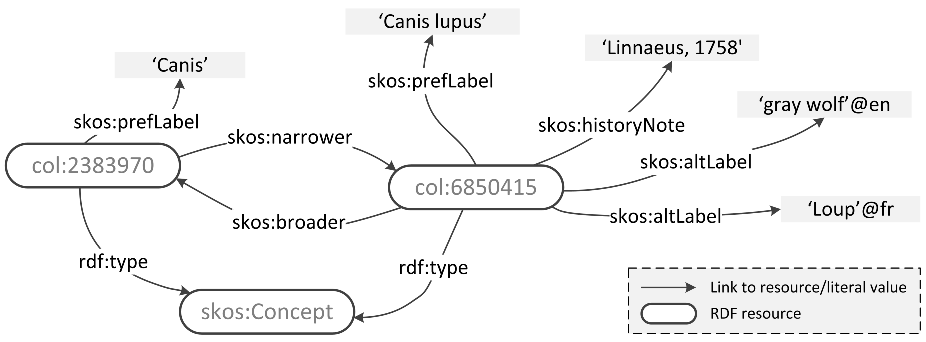


Figure 4.5: An example of the Catalogue of Life SKOSified data in the form of a graph.

A vocabulary that was extensively used in the context of Natural Europe was the Catalogue of Life (CoL), which is the most comprehensive catalogue of living species, containing over 1.4 million species along with their relationships. It is widely used in biological classification and serves as an important point of reference for many institutions, including Natural History Museums. CoL offers a web-based system for browsing the species taxonomy, as well as a set of web services for searching. However, CoL lacks support for persistent URIs able to be referenced by external applications, as well as SKOS representation of its data. Towards this end, we worked on a method of exposing the taxonomy of CoL to SKOS, using the CoL annual checklist and a D2R Server [**?**]. The features of SKOS that we employed are: (*a*) the class `Concept`, and (*b*) the properties `broader`, `narrower`, `prefLabel` and `altLabel`.

The first step in the process was the representation of all the taxonomic nodes as Concepts. The scientific name of each node was transformed into a `prefLabel`, and the common names into `altLabels`. Finally, the taxonomic hierarchy was retained by connecting parent and child nodes using the properties `broader` and `narrower`. An example of the CoL SKOSified data is shown in Figure 4.5.

4.2 The metadata management life-cycle

The complete life-cycle that NECE defines for the metadata management comprises four phases: (*a*) pre-ingestion, (*b*) ingestion, (*c*) maintenance, and (*d*) dissemination.

During the *pre-ingestion phase (preparatory phase)* each NHM selects the CHO records/collections that will be contributed to the project and ensures that they will be appropriately migrated into Natural Europe. This includes the web publishing of CHOs along with their respective thumbnails, and the metadata unification of existing CHO metadata. Publishing of CHOs refers to the uploading of CHO descriptions on the museum's website, or simply to the uploading of digital object thumbnails to a web server. The most important part of this step is the acquisition of a persistent URI for each resource. The web publishing of media files, the creation of thumbnails and the assignment of persistent URIs can be undertaken by MMAT. On the other hand, the metadata unification of existing CHO metadata is performed by preparing XML records conforming to the Natural Europe CHO Application Profile. This step can be easily carried out by any well-known legacy database system and even from Excel documents.

During the *ingestion phase* any existing CHOs and CHO metadata are imported to the Natural Europe environment. The latter are further enriched through a semantic annotation process. MMAT provides functionality for loading metadata conforming to the Natural Europe CHO Application Profile, as well as CHOs into its underlying repository. Afterwards, museum curators have the ability to inspect, modify, or reject the imported CHO descriptions. As far as the ingestion through the normal metadata curation/annotation activity is concerned, MMAT allows museum curators to maintain (create/view/modify/enrich) CHO metadata. This is facilitated by the access and concurrency control mechanisms, ensuring security, integrity, and consistency of the content.

The *maintenance phase* refers to the storage and management of CHOs and CHO metadata using MMAT and the CHO Repository. It addresses policies related to the integrity, authenticity and chain of custody. Integrity of data is guaranteed by performing full and incremental database backups on a weekly and daily basis respectively. These backups are persisted on remote machines and provide the means to overcome any failure on the servers hosting the systems with minimum loss. Concerning authenticity and chain of custody, both are controlled by the system's logging mechanism keeping track of actions/changes on the CHO records/collections, along with information about the user that performed each action/change. This enables rollback to previous states of the CHO record/collection when required.

The *dissemination phase* refers to the controlled provision of the maintained metadata to third party systems and applications, like the Natural Europe federal node and the BioCASE/GBIF networks. Metadata dissemination is mainly performed by the Access Module of the CHO Repository, which provides functionality for receiving information access requests and replying in several response formats, while applying various access control policies. It exposes (*a*) an OAI-PMH interface for the selective harvesting of CHO metadata, (*b*) a service interface implementing the BioCASE protocol, and (*c*) an OpenSearch endpoint. The OAI-PMH interface supports metadata dissemination in DC and Natural Europe CHO Application Profile format.

4.3 Connection of the Natural Europe Cultural Environment with BioCASE/GBIF

The Biological Collection Access Service for Europe (BioCASE) [16] is a transnational network of biological collections of all kinds. It enables widespread unified access to distributed and heterogeneous European collections and observational databases using open-source, system independent software and open data standards/protocols. Similarly, the Global Biodiversity Information Facility (GBIF) [5] is an open infrastructure which provides a single point of access to global (world-wide) biodiversity data.

In order for data providers to connect to BioCASE, they have to install the BioCASE Provider Software. This software offers an XML data binding middleware for publishing data residing in relational databases to BioCASE. The information is accessible as a web service and retrieved through BioCASE protocol requests. The BioCASE protocol [2] is based on the ABCD Schema [17], which is the standard for access and exchange of data about specimens and observations. Furthermore,

[2]http://www.biocase.org/products/protocols/

this protocol is supported (among others) by GBIF for accessing data from its providers.

Figure 4.6 presents an overview of the BioCASE architecture. On the top left resides the BioCASE portal, backed up by a central cache database, accessing information from the data providers (bottom). The BioCASE Provider Software (wrapper) is attached on top of each provider's database, enabling communication with the BioCASE portal and other external systems like GBIF. This wrapper is able to analyze BioCASE protocol requests and transform them to SQL queries using some predefined mappings between ABCD concepts and table columns. The SQL queries are executed over the underlying database and the results are delivered to the client after being transformed to an ABCD document.

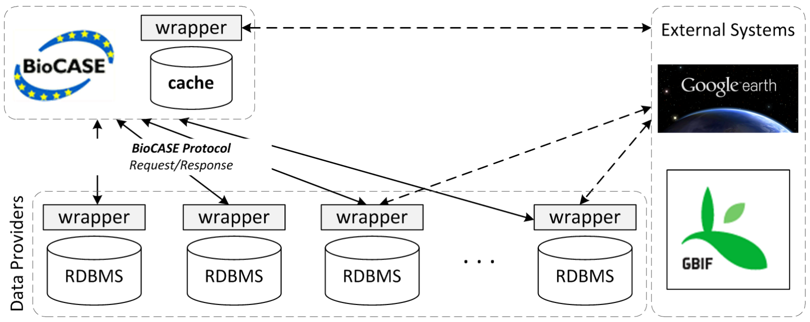


Figure 4.6: The BioCASE Architecture.

Although BioCASE supports a variety of RDBMSs, it does not support non-SQL databases. This is also the case of MMAT, which is backed by an eXist XML Database. To address this problem, we have built and installed a customized wrapper to the Access Module of the data providers' CHO Repositories (Fig. 4.7). The wrapper is able to analyze BioCASE protocol requests and transform them to XQueries, exploiting mappings between the Data Provider's schema ABCD. Towards this end, a draft mapping of the Natural Europe CHO Application Profile to ABCD was produced based on BioCASE practices. The XQueries are executed over the providers' repositories and the results are delivered to the client after being transformed to an ABCD document.

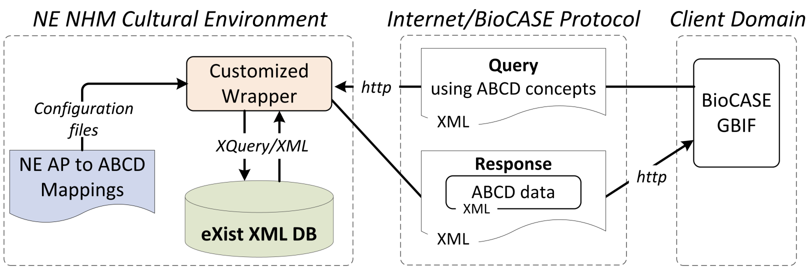


Figure 4.7: Connecting Natural Europe Cultural Environment with BioCASE/GBIF.

Although, the wrapper has been implemented for the XML databases of Natural Europe, it is able to support any underlying database either relational or XML with minimum effort. To this end, it follows a modular multi-tier architecture consisting of the following layers:

- The Service Layer controls the communication between the data provider and the BioCASE Portal by implementing the BioCASE protocol. It exposes services that comply to the BioCASE protocol specification, while concealing the wrapper's business logic. The basic system services are: (*a*) the Search Service, enabling complex query execution, based on ABCD concepts, over a data provider's database, (*b*) the Scan Service, supporting the retrieval of unique values for a given ABCD concept, and (*c*) the Capabilities Service, providing useful information about the ABCD concepts that can be used for searching in a data provider's database.
- The Business Logic Layer consists of three basic modules: (*a*) the Query Deserialization Module, handling the deserialization of the submitted queries to database-specific format, (*b*) the Mapping Management Module, administering the mappings between ABCD (used by the BioCASE Protocol) and the data provider's schema, and (*c*) the Results Serialization Module, managing the transformation of query results to an ABCD document, utilizing the mappings provided by the Mapping Management Module.
- The Data Layer provides simplified access to the data stored in the persistent storage.

Changes in the persistent storage can be easily supported by providing a new implementation of the Query Deserialization Module, based on the query language supported by the new persistent storage and the underlying data structure. To this end, the module has been designed using the principles of the plugin pattern and is therefore able to automatically recognize new module implementations. On the other hand, changes in the mappings between the ABCD and the data provider's schema can be addressed by modifying the wrapper's configuration files.

The source code of our implementation has been contributed to BioCASE and our approach has been successfully tested by their technical staff. Until the actual connection of the Natural Europe federated nodes (data providers) to the BioCASE/GBIF networks is established, we have deployed a local BioCASE portal installation able to retrieve CHOs from all federated node CHO Repositories [3].

4.4 Deployment and usage

In order to facilitate the deployment of the MMAT, the CHO Repository and the Vocabulary Server in any museum, we have compiled a packaged version of the whole infrastructure which can be

[3]`http://natural-europe.tuc.gr/biocase`

Table 4.1: The number of CHOs annotated by each NHM using MMAT.

Natural History Museums (NHMs)	**Cultutal Heritage Objects (CHOs)**					
	Images	**Videos**	**Sounds**	**Texts**	**3D**	**TOTAL**
Natural History Museum of Crete (NHMC)	3,840	13	0	157	0	**4,010**
National Museum of Natural History of Lisbon (MNHNL)	1,934	37	30	653	32	**2,686**
Jura-Museum Eichstätt (JME)	1,214	42	115	287	0	**1,658**
Arctic Center (AC)	480	0	0	0	0	**480**
Hungarian Natural History Museum (HNHM)	2,418	51	0	1,770	5	**4,244**
Estonian Museum of Natural History (TNHM)	1,736	100	0	136	0	**1,972**
TOTAL	**11,622**	**243**	**145**	**3,003**	**37**	**15,050**

hosted on any server. This allows for rapid deployment of the tools by less experienced people. Moreover, all the components have been built as separate modules, which means that in the case of a new version they can be updated individually.

The infrastructure has been deployed in the six NHMs participating in the project, allowing the curators to publish, semantically describe, manage and disseminate a large volume of CHOs. The participating museums are: (*a*) Natural History Museum of Crete (NHMC), (*b*) National Museum of Natural History of Lisbon (MNHNL), (*c*) Jura-Museum Eichstätt (JME), (*d*) Arctic Center (AC), (*e*) Hungarian Natural History Museum (HNHM), and (*f*) Estonian Museum of Natural History (TNHM). Table 4.1 presents the number of CHOs that have already been published by each NHM using MMAT.

Improvements on the user-interface have been made after continuous feedback from museum partners in a number of tool releases. Heuristic evaluation of the MMAT was performed, while extensive usability studies have been performed in a number of curator workshops organized by the participating NHMs.

Summary

In this chapter we presented the architecture, deployment and evaluation of the infrastructure used in the Natural Europe project, allowing curators to publish, semantically describe, and manage the museums' CHOs, as well as disseminate them to Europeana and to BioCASE/GBIF networks. This infrastructure consists of the Multimedia Authoring Tool, the CHO Repository and the Vocabulary

Server. It is currently used by six European NHMs participating in the Natural Europe project, providing positive feedback regarding the usability and functionality of the tools. Until today, a large number of CHOs has already been published.

Chapter 5

Functional Specification

This chapter describes the functional specification of the system that has been developed for the management of the lifecycle of the observations and their metadata. It specifies the stakeholders of the system, lists the technical requirements that had to be met for achieving the desired functionality and provides an in depth analysis of the system's functionality.

Section 5.1 presents the system's stakeholders and their role in the system, while Section 5.2 discusses the technical requirements of the tool. Finally, Section 5.3 provides the functionality that has to be provided by the system in the form of use cases.

5.1 Stakeholders

Our framework is targeted for organizations that want to provide users with the ability to create and contribute day-to-day observations about the objects that they come across and are of their interest. Although our basic target is the biodiversity context, the framework has been designed with extensibility in mind, to allow the support of other contexts as well. In any case, the system stakeholders are:

- The **organizations** that will collect and hold the data will be able to use the observations to reach meaningful conclusions regarding the changes in the population or the changes of habits of species.
- The **domain experts** experts will gain knowledge based on the observations collected by the users.
- The **common users** will use the systems to search, review and provide the observations and the objects. This will enable them to expand their knowledge and experiences in identifying

species that interest them. Moreover, taking into consideration the observations made by others, they can discover objects that they would not be able to find otherwise.

5.2 Technical Requirements

This section discusses the technical requirements that were identified and set for the development of the model and the systems. The requirements have been split in 5 categories and are summarized below. Section 5.2.1 describes the requirements of the Model. Section 5.2.2 describes the requirements of the Meta-Model used to implement custom Application Profiles. Section 5.2.3 describes the requirements of the Mobile Application, and Section 5.2.4 describes the requirements of the Web Application. Finally, Section 5.2.5 describes the requirements of the system's backend.

5.2.1 Model

As we have already mentioned, a strong requirement that we set for our infrastructure is that it should not be bound to a single context. Instead, it should be configurable so that it can support any domain with the specification of different Application Profiles. However, we have identified three entities that form the base of our model. These are:

- **Observation**: An observation is the cornerstone of our model. It is considered a single occurrence at a specific time and place that the user witnesses and reports.
- **Object**: An object is the item or the fact that is being 'observed' by the user.
- **Multimedia**: The multimedia objects optionally follow the observations and the objects. They can be of the types: images, videos, sounds.

Some examples describing contexts that the model can apply are:

- **Biodiversity observations about species**: In the biodiversity context, this infrastructure aims to alleviate the need for experts capturing biodiversity information which is an extremely slow process due to the limited number of experts and the limited funding in the area. Objects need to be the nodes from the taxonomy of species that is supported. Observations are the sightings of the species (object) at a specific location and time. Alternatively, objects can represent events in nature (e.g. hunting, mating). In this case the main focus is the reports of the sightings of an animal or a plant and enrich them with some visual information (multimedia).

- **Earthquake observations**: Objects will need to be the earthquakes (along with the metadata concerning the size, depth, location, casualties, etc...). This is basically an ‘event’, described as an object. Observations will be the photos captured by users during or after the event ‘earthquake’. These will be connected with the specific earthquake (object). In this case we focus on reporting the outcome/damages of earthquakes. We basically form collections of observations and each collection refers to an earthquake.
- **Road Accidents**: Objects will be the accidents with information about the type, number of people involved, etc. In this case the objects will need to have a gps point. Observations will be the pictures taken from people passing by along with a description. They will have gps location from different positions/angles. In this case we focus on reporting the accidents and enrich them with some multimedia.

5.2.2 Meta-Model

The Meta-model should define the way that the objects and the observations are created, populated and presented. Moreover, it must determine the way that these are searched by the users and other systems.

In more detail, there need to be various types of objects and observations, depending on the context of the system. The Application Profile that derive from the Meta-Model should describe all the types of both objects and observations, and contain one or more attributes for each of them. Each ‘attributes’ in the model needs to have at least a name and a description. Furthermore, for each attribute we need to be able to specify it’s multiplicity (e.g. how many times it can exist in the metadata), and it’s obligation (e.g. if it’s required to exist for the description to be valid). Additionally, we need to have different types of attributes, like *text*, *number*, *list*, *geolocation*, *date*, etc. Each of the attributes also needs to have more ‘options’ specified, depending on it’s type, e.g., the list selection needs to have the list of the possible values.

5.2.3 Mobile Application

The mobile application needs to be compatible with most state-of-the-art mobile devices. A problem with this is the fact that different device vendors provide different own Framework for the development of applications compatible with their devices. Most of these are using completely different programming languages. To this end, we decided to base our application on HTML5 and Javascript, eliminating the need for porting the code in different platforms and programming languages to provide support for different devices.

The application should also be native rather than web, so that the user can create a launch icon and use it just like normal native applications instead of opening the browser on his device and browse to the url. In addition, when the device is not connected to the internet, using native apps is a better fit to the user's state of mind. Although latest standards in HTML5 allow the offline usage of web applications, they still require the user to launch a browser in order to use the application. The requirement for making a web application operate natively on mobile devices has risen from the very beginning of mobile device programming. Although many platforms have been developed to provide this functionality and they each have advantages and disadvantaged [41], we used 'Phonegap', which is the most popular among them.
The features of the native mobile application should include:

- Compatible with most state-of-the-art mobile devices/platforms.
- Real-time creation of observations based on the application profiles.
- Real-time creation of objects based on the application profiles.
- Image/Video/Audio capturing, along with any metadata provided by the mobile device capabilities.
- Offline capturing of observations and their enrichment with the appropriate metadata depending on the application profiles. The observations will be kept temporarily on the device and persisted on the server when internet is available.
- Visualize the observations on maps.
- Allow the user to perform basic searching on the observations/objects.

5.2.4 Web Application

The web application needs to be compatible with state-of-the-art standards. To this end it should be based on HTML5 and Javascript. Moreover, we need to support both desktop computers and mobile devices. In the case of the mobile devices, the interfaces and the features must be similar with those described in Section 5.2.3. In the desktop version of the web application, the users must be provided with the same functionality, but with interfaces optimized for big screens. Additionally, the administrators need to manage the system from an administrator interface. To summarized, the features needed are:

- Compatible with state-of-the-art web standards.
- Management of observations based on the application profiles.

- Management of objects based on the application profiles.
- Administration interfaces
- Visualization of the observations on maps.
- Searching functionality for observations and objects.

5.2.5 Backend

The backend of our system needs to support all server platforms. Moreover, the features needed to be supported are:

- Compatible with state-of-the-art standards.
- Persistence of objects and observations based on the application profiles.
- Restful web services for allowing the access of the data from external applications.
- Publishing the multimedia that are uploaded by the users.
- Creation of thumbnails for the multimedia.
- Efficient searching on the data.

5.3 Use Cases

This section contains the use cases that we have identified for the system. The use cases have been described following the methodology from Alistair Cockurn [26].

Table 5.1: Use Case 1: "Log in"

Use Case 1: "Log in"	
Goal in Context	The user wants to be logged into the system in order to use the full system's functionality.
Scope & Level	System, Sub-function
Preconditions	The user has already created an account to the system.
Trigger	The user visits the application's url without being already logged in.
Description	1. The system displays the log in form. 2. User fills in the form with his credentials and selects to log in. 3. The system validates the submitted form details, checks if the user credentials are correct. 4. The system displays the first screen to the user.

Table 5.2: Use Case 2: “Logout”

Use Case 2: “Logout”	
Goal in Context	The user wants to be logged out from the system.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to logout from the system.
Description	1. The system logs the user out of the system. 2. The system displays the login screen.

Table 5.3: Use Case 3: “Create new account”

Use Case 3: “Create new account”	
Goal in Context	The user wants to create a new account so he can have full access to the system’s functionality.
Scope & Level	System, Sub-function
Preconditions	The user is not logged in the system.
Trigger	The user selects to create a new account in the system.
Description	1. The user fills the required information about his new account. 2. The system validates the user’s input and creates a new user which is persisted in the database.

Table 5.4: Use Case 4: “Create Object”

Use Case 3: “Create Object”	
Goal in Context	The user wants to create a new object.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to create a new object.
Description	1. The user browses to the list of objects with the same type as the object he wants to create. 2. The user selects to create a new object from the menu. 3. The system presents the interface for the creation of the new object, based on the application profile for the object’s type. 4. The user fills the required information about the object and uploads any available multimedia files. 5. The system validates the user’s input and creates a new object which is persisted in the database. 6. The system presents the newly created object to the user.

Table 5.5: Use Case 5: “Create Observation”

Use Case 5: “Create Observation”	
Goal in Context	The user wants to create a new observation.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to create a new observation.
Description	1. The user browses to the list of observation with the same type as the observation he wants to create. 2. The user selects to create a new observation from the menu. 3. The system presents the interface for the creation of the new observation, based on the application profile for the observation’s type. 4. The user fills the required information about the observation and uploads any available multimedia files. 5. The system validates the user’s input and creates a new observation which is persisted in the database. 6. The system presents the newly created observation to the user.

Table 5.6: Use Case 6: “Update Object”

Use Case 6: “Update Object”	
Goal in Context	The user wants to update the metadata of an object.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to update an existing object.
Description	1. The user browses through the list of objects. 2. The user selects an object from the list and reviews it’s metadata. 3. The user selects to edit the object. 4. The system presents the interface for editing the object, based on the application profile for the object’s type. 5. The user makes the required changes to the object. 6. The system validates the user’s input and updated the object metadata in the database. 7. The system presents the updated object to the user.

Table 5.7: Use Case 7: "Update Observation"

Use Case 7: "Update Observation"	
Goal in Context	The user wants to update the metadata of an observation.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to update an existing observation.
Description	1. The user browses through the list of observation. 2. The user selects an observation from the list and reviews it's metadata. 3. The user selects to edit the observation. 4. The system presents the interface for editing the observation, based on the application profile for the observation's type. 5. The user makes the required changes to the observation. 6. The system validates the user's input and updated the observation metadata in the database. 7. The system presents the updated observation to the user.

Table 5.8: Use Case 8: "Delete Object"

Use Case 8: "Delete Object"	
Goal in Context	The user wants to delete an object.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to delete an existing object.
Description	1. The user browses through the list of objects. 2. The user selects an object from the list and reviews it's metadata. 3. The user selects to delete the object. 4. The system asks for confirmation from the user. 5. The system removes the object and it's metadata from the database. 6. The system notifies the user of the successfull deletion of the object.

Table 5.9: Use Case 9: "Delete Observation"

Use Case 9: "Delete Observation"	
Goal in Context	The user wants to delete an observation.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to delete an existing observation.
Description	1. The user browses through the list of observations. 2. The user selects an observation from the list and reviews it's metadata. 3. The user selects to delete the observation. 4. The system asks for confirmation from the user. 5. The system removes the observation and it's metadata from the database. 6. The system notifies the user of the successfull deletion of the observation.

Table 5.10: Use Case 10: "Create new type of Objects/Observations"

Use Case 10: "Create new type of Objects/Observations"	
Goal in Context	The user wants to add a new type of objects / observation to the system, based on the application profile.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system and has administrator privileges.
Trigger	The user selects to create a new type of Objects.
Description	1. The user browses through the list of application profile types. 2. The user selects to create a new application profile type. 3. The system presents the new type form. 4. The user completed the form depending on his needs and submits it. 5. The system validates the user's input and creates a new object type which is persisted in the database. 6. The system presents the newly created object type to the user.

Table 5.11: Use Case 11: “Update user profile”

Use Case 11: “Update user profile”	
Goal in Context	The user wants to update his profile.
Scope & Level	System, Sub-function
Preconditions	The user is already logged in the system.
Trigger	The user selects to update his profile.
Description	1. The user updates the information presented in his profile page. 2. The system validates the user’s input and persists the updated profile in the database. 3. The system presents the updated user’s profile.

Table 5.12: Use Case 12: “Review Observations of a specific Object”

Use Case 12: “Review Observations of a specific Object”	
Goal in Context	The user wants to view all the observations that have been made for a specific object.
Scope & Level	System, Sub-function
Preconditions	The object exists in the system.
Trigger	The user selects to view the object’s observations.
Description	1. The user finds the object that interests him and selects to see it’s observations. 2. The system retrieved the observations that are connected with the object, and presents them to the user.

Table 5.13: Use Case 13: “Review Multimedia of a specific Object”

Use Case 13: “Review Multimedia of a specific Object”	
Goal in Context	The user wants to view all the multimedia that have been captured for a specific object.
Scope & Level	System, Sub-function
Preconditions	The object exists in the system.
Trigger	The user selects to view the object’s multimedia.
Description	1. The user finds the object that interests him and selects to see the respective multimedia. 2. The system retrieved the multimedia that are connected with the object, and presents them to the user.

Summary

In this chapter we described the functional specification of the system that has been developed for the management of the whole lifecycle of the observations and their metadata. We specified the stakeholders of the system, listed the technical requirements that had to be met for achieving the desired functionality and provided an in depth analysis for the system's functionality along with the use case tables.

Chapter 6

Model

This chapter describes the model developed for describing the observation that are supported by our system. Moreover, the classes that it is comprised of are identified and presented. The chapter has been split into two sections, Section 6.1 presents the Meta-Model that has been designed to provide extensibility to the Model, and Section 6.2 presents the Model that enables the creation of Observations that comply to our Meta-Model and the Application Profiles that derive from it. Finally, Section 6.3 presents two example implementations for the Model and the Meta-Model for a system that supports observations of plants.

6.1 Meta-Model

The Meta-Model defines the way that the objects and the observations are created, populated and presented. Moreover, it determines the way that these are searched by the users and external systems. In more detail, the population and usage of the Application Profiles based on the Meta-Model allow the existence of various types of objects and observations in the same instance of the system. The Meta-Model describes all the types of both objects and observations, allowing the configuration of different metadata for each type.

Figure 6.1 presents the Meta-Model in the form of a class diagram. The following sections describe the classes that comprise it in detail.

Object Type

The Object Type class represents a single type of objects and observations that are supported by the Application Profiles. Each profile contains one instance of this class, and further defines the way that the metadata about the objects and the observations are populated.

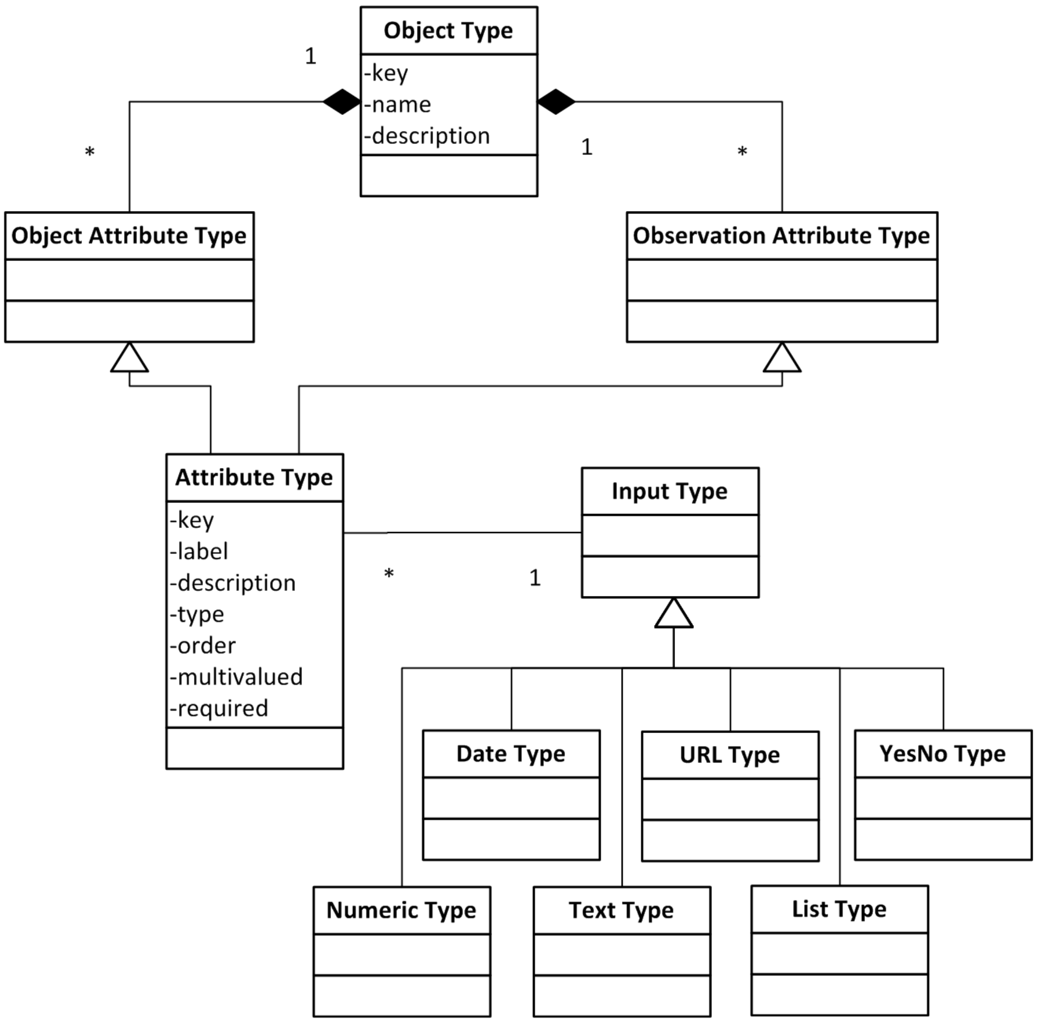


Figure 6.1: The Meta-Model.

The attributes that it holds are:

- *key*: the unique identifier for the Object Type. This will usually be the same as the name of the type.
- *name*: the descriptive name of the Object Type, used in the user interface.
- *description*: a basic description for the contents of the attribute, used as a helping guide for novice users.

Apart from these attributes, the Object Type aggregates all the attributes that are configured in order to describe the Objects and the Observations of the type, which are depicted as the Object and the Observation Attributes respectively.

Observation Attribute

The Observation Attribute represents a single attribute used in the description of the Observations that are of a specific Object Type. The type of this attribute is further defined by the attributes that it inherits from the class 'Attribute Type'.

Object Attribute

The Object Attribute represents a single attribute used in the description of the Objects that are of a specific Object Type. The type of this attribute is further defined by the attributes that it inherits from the class 'Attribute Type'.

Attribute Type

This Attribute Type class is the base class in the Application Profile, providing the extensibility capabilities to the attributes of the Model. It represents any attribute that is used in the metadata descriptions of both the Objects and the Observations. The attributes that it holds are:

- *key*: the unique identifier for the attribute and is also used as the key in the metadata of the objects/observations when persisted in the database.
- *label*: the descriptive name of the attribute, used in the user interface.
- *description*: a basic description for the contents of the attribute, used as a helping guide for novice users.
- *type*: defines the type of the attribute. This is mostly used in the user interfaces for the creation of different user interface elements for different attribute types.

- *order*: defines the order of the attribute as it appears in the user interfaces.
- *multivalued*: defines if the attribute can appear multiple times in the metadata.
- *required*: defines if the attribute must be populated in order for the metadata of the object/observation to be valid.

Input Type

This abstract class has been created so as to provide future extensibility to the attributes. All the different types of attributes extend this class, and can provide any additional options needed in order to be best defined. Furthermore, all instances of the class Attribute Type contain an instance of this class which specifies the type of the attribute.

Date Type

This type of attribute input type represents a date attribute in the metadata.

Text Type

This type of attribute input type enables textual information to exist as an attribute in the metadata.

Number Type

This type of attribute input type represents a numerical value.

URL Type

This type of attribute input type enables URLs to exist as an attribute in the metadata.

List Type

This type of attribute input type represents a list of attributes from which the user has to choose. This requires an additional attribute for the specification of the list, which is a list of all the possible values.

YesNo Type

This type of attribute input type represents a binary choice between ‘yes’ and ‘no’.

GPS Type

This type of attribute input type represents a specific geographic point (GPS). It contains two attributes for the latitude and the longitude of the geographic position.

6.2 Model

The Model developed and supported by our systems has been designed with extensibility in mind, so as to be able to describe generic types of observations. These observations could be from single points on the map, to complex descriptions of the observed items, containing multiple multimedia objects.

The three base entities of the model are: Object, Observation and Multimedia. Figure 6.2 presents the Model in the form of a class diagram. On the top of the diagram we find a shorter version of the Meta-Model that we previously defined. It is replicated here so the relationships between the classes of the Model and the classes of the Application Profile are visible.

As an example, in the biodiversity context, the Observations relate to one or many species, depicting a single occurrence of them in time and space. Object describe the items for which the observations are made and the Multimedia files can follow Observations.

The domain experts will need to create the Objects of the system. Each object will represent an entity for which the users can create observations. The following sections describe the classes of the Model in more detail:

Object

This class represents the taxonomic classification of the objects for which the observations are made. The data entry and management of the instances in this class will in most cases fall on the domain experts.

As an example application, when supporting a system for nature observation concerning flora and fauna, the objects would be the different kinds of plants or animals that the user can observe. In this case the most fitting type of identification for objects is the scientific name along with the common names of the species.

The Objects are followed by a descriptive label, uniquely identifying the object and optionally any multimedia files available. Additionally, using the power of the application profiles supported by the system, the objects can contain any additional metadata that the experts see fit, like a descriptive text with information about the object, or a list of URLs where the user is able to find further information about the object.

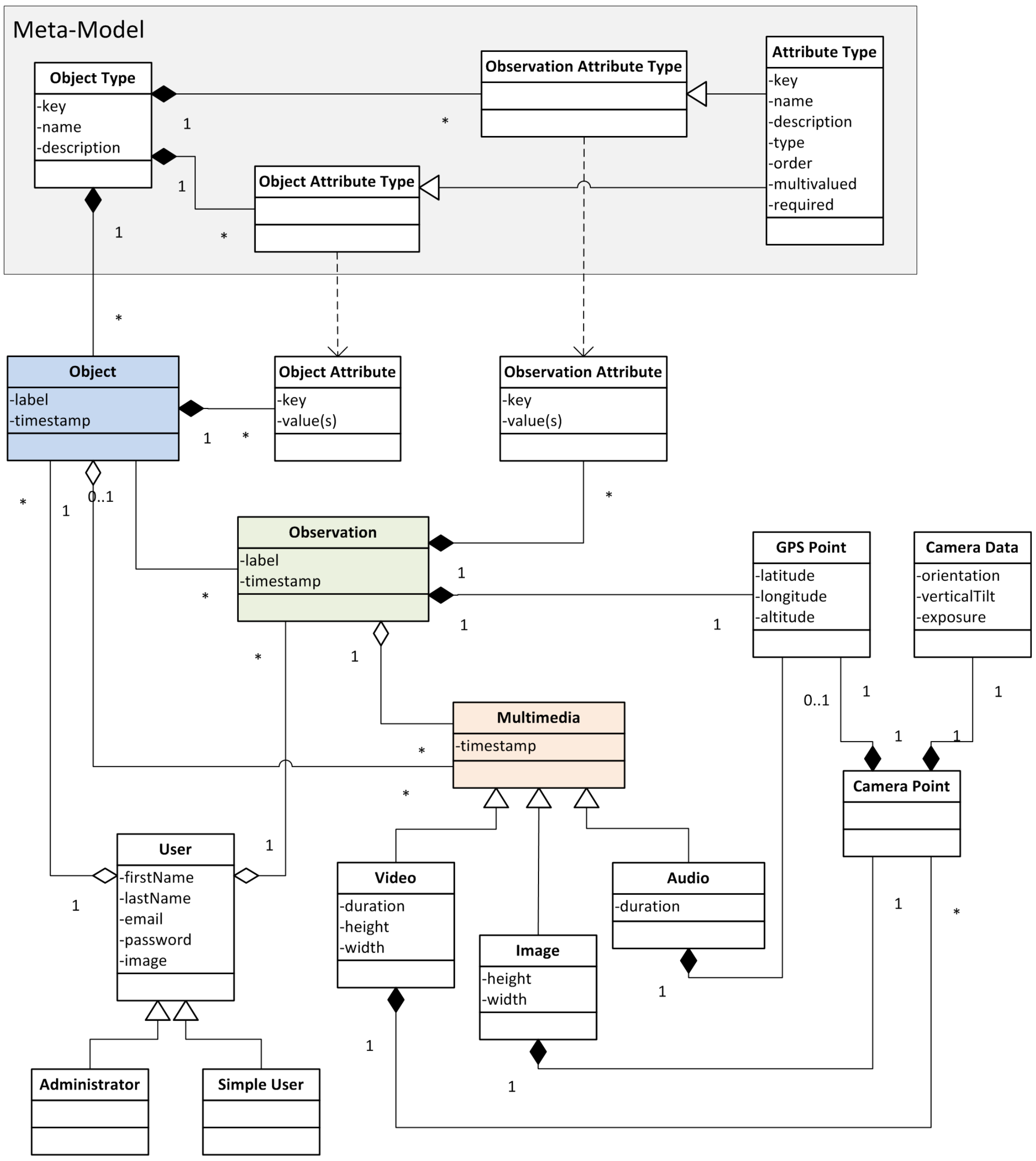


Figure 6.2: Model.

Object Attribute

This class represents an attribute that follows an object, based on the application profile that the object follows. Each instance of this attribute holds the key of the attribute as defined in the Application Profile, and one or more values.

Observation Attribute

This class represents an attribute that follows an observation, based on the application profile that the observation follows. Each instance of this attribute holds the key of the attribute as defined in the Application Profile, and one or more values.

Observation

This class represents an occurrence of an Object that has been observed by a User, along with any Multimedia files that he was able to provide.

Each observation contains at minimum a label, firmly describing the observed occurrence and a timestamp, placing it on a specific point in time. Additionally, it contains geospatial information, which is a GPS point. The observation is optionally connected to a single Object, marking its occurrence. Moreover, it can contain additional attributes, depending on the application profile.

Multimedia

This class represents the multimedia objects that follow the Observations and the Objects. A multimedia object can follow one observation, while a single observation can have multiple multimedia objects attached to it (for example when there are shots from different angles on the same object, or different types of multimedia like sound and picture)

The multimedia objects can be video, image, text or audio files captured with a mobile phone, camera or any other capturing device. Each type of Multimedia is depicted as a different class, holding it's own descriptive attributes. The only attribute that is common in all

These objects can optionally have spatiotemporal information.

If they don't have this data, they might be general images following an observation and we could place them in the space-time point of the observation (since these are mandatory in the observation). Another possibility could be that the multimedia objects have different spatial information than the observation, in cases where the user captures objects in distance (in this case the user should provide the input as to where the observation is relevant to his position). The metadata alongside the images will be height and width, along with the orientation and tilt of the camera. The video can additionally have starting and ending points and possible point in the between (Camera Point),

GPS Point

This describes the geospatial position of an observation. Latitude and longitude can be captured by mobile devices or inserted by the user manually using an interactive map. The altitude can be inferred from the other two properties using existing web services.

Camera Data

This class represents the data collected by the camera during the capturing of the multimedia, when available. It will hold camera tilting when the device provides an accelerometer, the orientation from the compass and all the additionally metadata provided (shutter speed, angle, etc.).

Camera Point

This class represents a specific instance of the camera in space. It contains both GPS Point and Camera Data information. It aggregates information for the GPS Point and the Camera Data.

It was introduced in order to represent path of camera points in a video. Used in videos and images only since the Camera Data can only apply to them.

Audio

This class represents the Multimedia that are of the type of audio. Optionally, it contains a GPS Point with the location of the capturing.

Image

This class represents the Multimedia that are of the type of image. Optionally, it contains a Camera Point with data about the location of the capturing and any additional information about the image.

Video

This class represents the Multimedia that are of the type of video. Optionally, it contains a list of Camera Points with the data about the location of the capturing and any additional information provided by the camera. The list of Camera Points can be used to capture and recreate the path that the user took for the duration of the capturing.

User

This class represents the users of the system. The access level of the user is refined in the two subclasses, Simple User and Administrator that are described below. The attributes that each user has are the following:

- *firstName*: the given first name of the user.
- *lastName*: the last name of the user.
- *email*: the user's email address.
- *password*: the password used for logging in the system.
- *image*: a profile image of the user.

Simple User

This class represents the general users of the system that have no administration privileges.

Administrator

This class represents the users of the system with administrator privileges. The users of this class can access the administrative interface and update the system settings, as well as the application profiles supported by the specific instance of the system.

6.3 Model Examples

This section presents two examples of the model of an imaginary system that supports observations of plants and animals. The first step for an administrator of such a system would be to define the Application Profiles for the plants and the animals, based on our meta-model.

At first we will need to support plants, so the Objects will represent the species of plants. In general, it is up to the experts of the domain to define the common attributes between plant species and the attributes required for their observations, but for the shake of simplicity we will try and identify some basic attributes. Having define the Application Profile for the objects, we can proceed by describing the profile that the observations will need to follow.

We first need to start with the attributes that are needed for the Objects. We need a *label* for the Objects which will be the scientific name of the species as well as a *descriptive text* giving some information about the plant. Moreover, we can add some *URLs* that link to websites with more insight to the plants. As concerns the visual perception, we can have the *color of the leafs* of the plants, as well as the *color of the flowers* or fruits when applicable. Finally, the *countries* that the plant is mostly met and its *lifespan* are two common attributes that can be provided with all plants. A more formal description of this profile is presented in Table 6.1.

Table 6.1: Example Application Profile for Plants. Each row represents an instance of the Object Attribute Type and each column is an attribute of the class Attribute Type. For simplicity we did not include the order, multivalued and required attributes

key	label	description	type
`label`	"Title"	"The scientific name of the plant"	text
`description`	"Description"	"A small description about the plant"	text
`url`	"External URL"	"A URL where the user can get more information about the object"	url
`leaf_color`	"Leaf Color"	"The normal color of the leafs of the plant"	text
`flower_color`	"Flower Color"	"The normal color of the flowers or the fruits of the plant (when applicable)"	text
`commonly_found_in`	"Commonly found country"	"A country where the plant is most commonly found"	list
`lifespan`	"Lifespan (years)"	"The average life of the plant"	number

Having defined the Application Profile for the objects, we need to also define the format of th observations. The attributes needed for the observations are: (*a*) a *label* for the observation, (*b*) a bigger *description*, (*c*) the *leaf color*, (*d*) the *flower color*, (*e*) the *height* of the plant, (*f*) an indication showing if the plant is *flourished*. A more formal description of this profile is presented in Table 6.2.

Table 6.2: Example Application Profile for Observations about Plants. Each row represents an instance of the Observation Attribute Type and each column is an attribute of the class Attribute Type. For simplicity we did not include the order, multivalued and required attributes

key	label	description	type
`label`	"Title"	"A small descriptive title for the observation"	text
`description`	"Description"	"A small description about the observation"	text
`leaf_color`	"Leaf Color"	"The color of the leafs of the observed plant"	text
`flower_color`	"Flower Color"	"The color of the flowers or the fruits of the observed plant (if applicable)"	text
`height`	"Height (meters)"	"The height of the plant in meters"	number
`flourished`	"Flourished"	"Is the plant flourished?"	yesno

Having described the Application Profile for the plants, we will present an instance of each of the two classes, Object and Observation. The Object is presented in Figure 6.3, where we can see the description of the plant "Lilium tsingtauense", and Figure 6.4 presents an observation of this plant in conformance to the Application Profile.

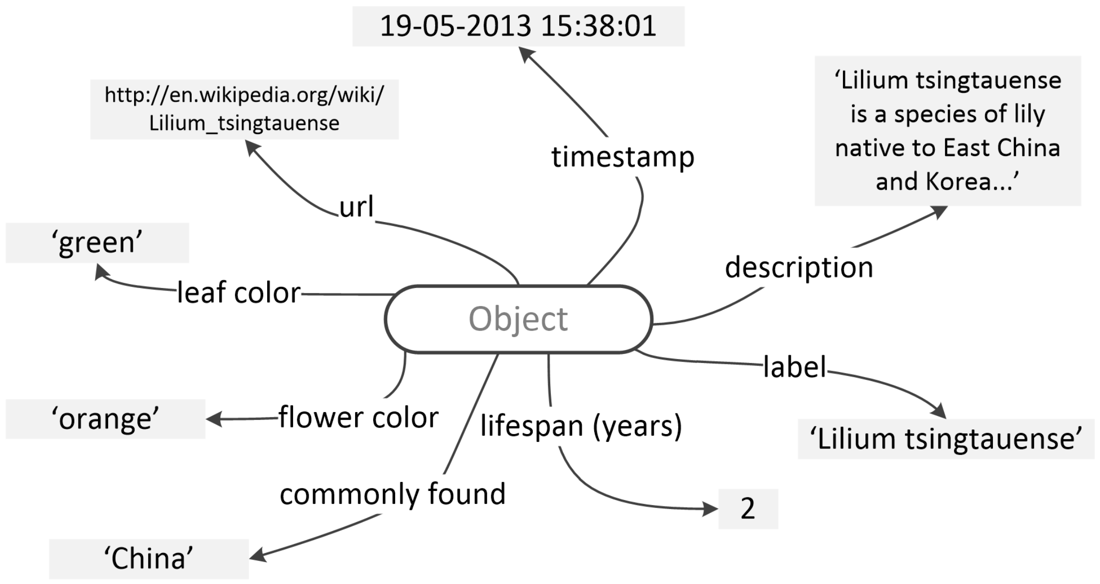


Figure 6.3: Model Example: Object for plant "Lilium tsingtauense"

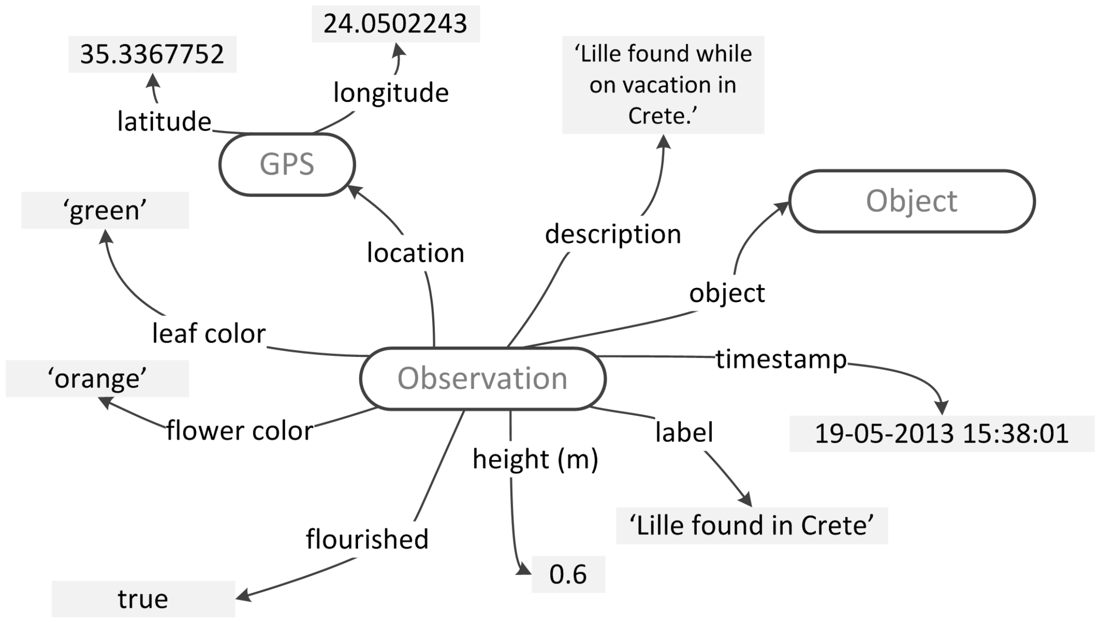


Figure 6.4: Model Example: Observation for the plant "Lilium tsingtauense"

The second type of objects and observations that we need to support are the animals. Table 6.3 presents the Application Profile for the objects of this type and Table 6.4 presents the Application Profile for the observations. Finally, Figures 6.5 and 6.6 contain two example instances of an object and an observation of the type 'animal'.

Table 6.3: Example Application Profile for Animals. Each row represents an instance of the Object Attribute Type and each column is an attribute of the class Attribute Type. For simplicity we did not include the order, multivalued and required attributes

key	label	description	type
`label`	"Title"	"The scientific name of the animal"	text
`description`	"Description"	"A small description about the animal"	text
`url`	"External URL"	"A URL where the user can get more information about the object"	url
`height`	"Mean height (m)"	"The normal height of an adult animal"	number
`weight`	"Mean weight (kg)"	"The normal weight of an adult animal"	number
`commonly_found_in`	"Commonly found country"	"A country where the animal is most commonly found"	list
`lifespan`	"Lifespan (years)"	"The average life of the animal"	number

Table 6.4: Example Application Profile for Observations about Animals. Each row represents an instance of the Observation Attribute Type and each column is an attribute of the class Attribute Type. For simplicity we did not include the order, multivalued and required attributes

key	label	description	type
`label`	"Title"	"A small descriptive title for the observation"	text
`description`	"Description"	"A small description about the observation"	text
`height`	"Height (meters)"	"The height of the animal in meters"	number
`weight`	"Weight (kg)"	"The weight of the animal in kg"	number
`gender`	"Gender"	"The gender of the animal"	list
`color`	"Color"	"The color of the fur of the animal"	text

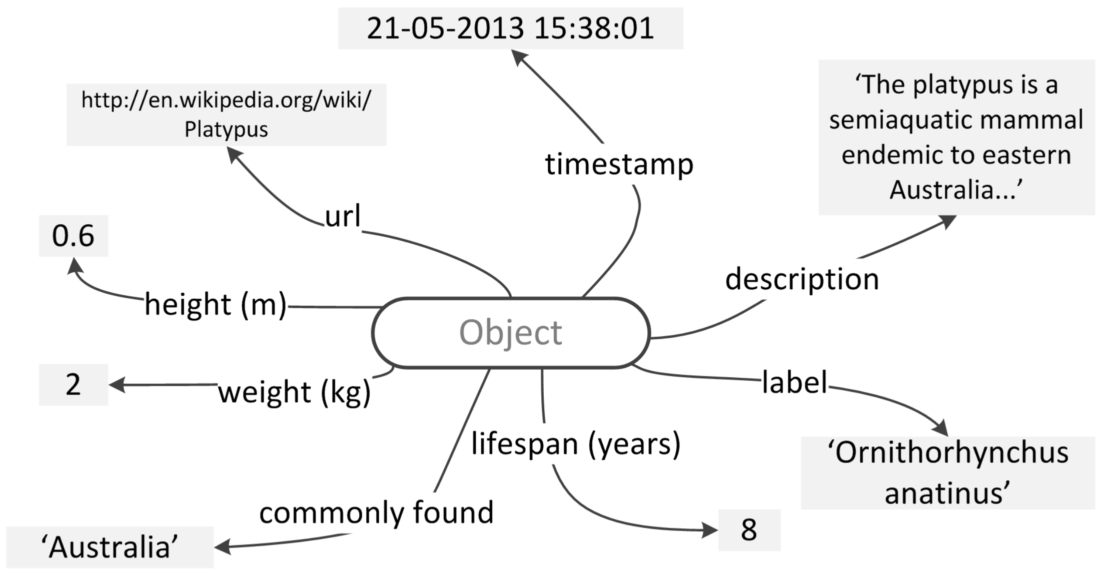

21-05-2013 15:38:01
http://en.wikipedia.org/wiki/Platypus
timestamp
'The platypus is a semiaquatic mammal endemic to eastern Australia...'
url
0.6
height (m)
description
Object
2
weight (kg)
label
lifespan (years)
'Ornithorhynchus anatinus'
commonly found
8
'Australia'

Figure 6.5: Model Example: Object for animal "Platypus"

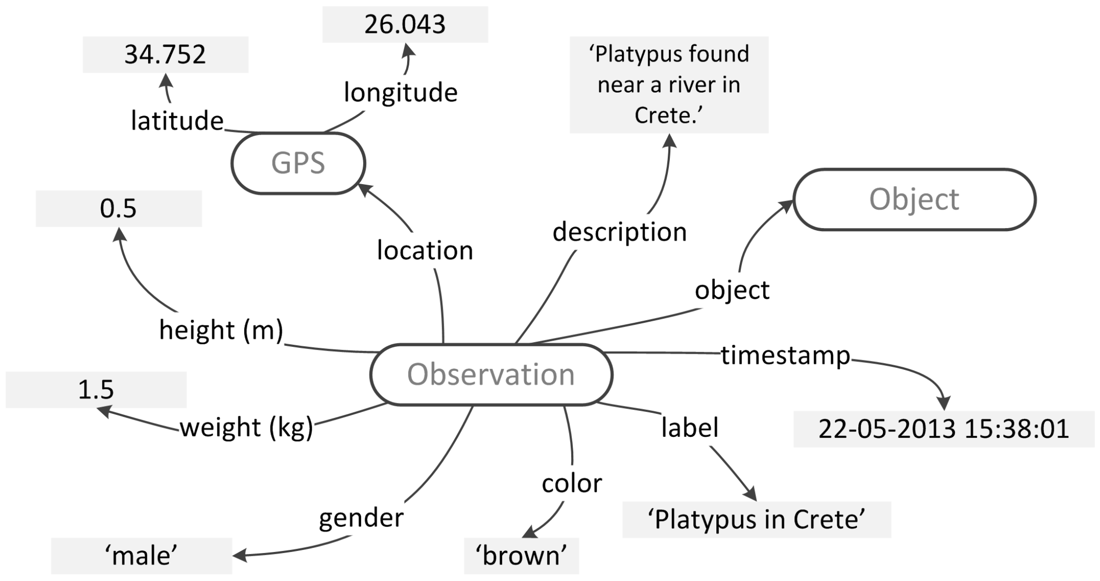


Figure 6.6: Model Example: Observation for the animal "Platypus"

Summary

In this chapter we presented the model that we have developed for the support of the observation in our system. We presented the Meta-Model that has been designed to provide extensibility to the Model, and the Model that enables the creation of Observations that comply to our Meta-Model and the Application Profiles that derive from it. Finally, we described two example implementations for the Model and the Meta-Model for a system that supports observations of plants.

Chapter 7

Architecture

This chapter describes the overall system architecture, identifies its basic components and provides an in depth analysis of the internal functionality. Moreover, it advocates the architectural decisions that ware made for the most important components.

Built as a web application, the system adopts the Rich Internet Application (RIA) principles, which promote the development of web applications as desktop applications performing business logic operations on the server side, as well as on the client side. The Client Side logic operates within the web browser running on a user's local computer, while the Server Side logic operates on the web server hosting the application. Figure 7.1 displays the overall system architecture.

For the development of the application we adopted several design patterns [37]. The use of well-established and documented design patterns, speeds up the development process, since they provide reusable solutions to the most common software design problems [15, 36]. The design patterns that we have used in designing the system's architecture are presented in detail in the following sections. The Model View Controller (MVC) design pattern [62, 63] and the Observer pattern were used on the client side, and a multi-tier architecture was implemented on the server side.

The analysis of the architecture has been broken into two parts; Section 7.1 presents the Client Side Architecture, while Section 7.2 presents the Server Side Architecture.

7.1 Client Side Architecture

The Client Side of the application is responsible for the interaction with the user. All the actions performed by an individual using the system, are handled by the client side logic, which undertakes the presentation of the information as well as the communication with the server. In order to achieve

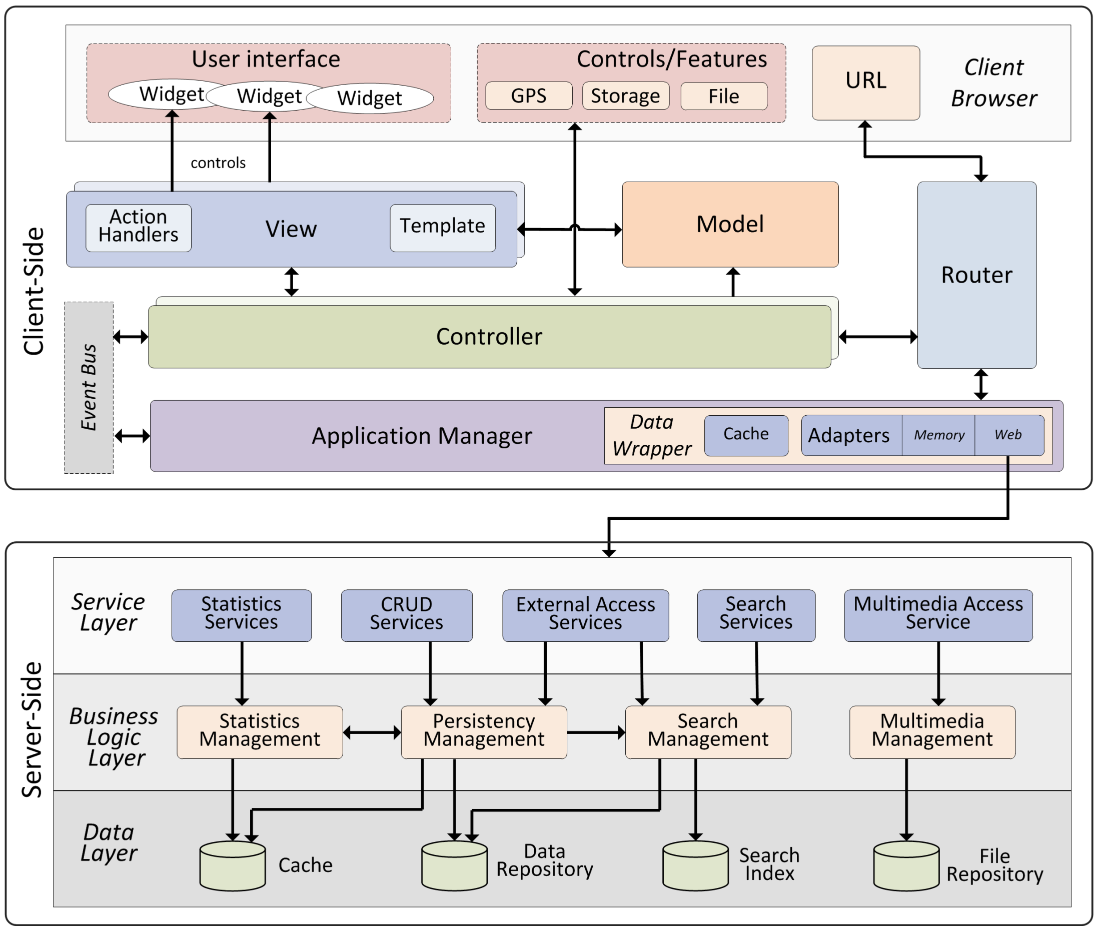


Figure 7.1: Overall System Architecture, containing client-side (top) and server-side (bottom) along with the included modules.

a high level of decoupling between the components forming the client logic we adopted the Model View Controller (MVC) design pattern [62, 63], as well as the Observer pattern.

The usage of the MVC pattern introduces the separation of the responsibilities for the visual display and the event handling behavior into different entities, named respectively, the View and the Controller. Some of the advantages of this approach are: (a) Maximization of the code that can be tested with automation (Web pages containing HTML elements are hard to test); (b) Code sharing between pages that require the same behavior; and (c) Separation of Business logic from User Interface logic to make the code easier to understand and maintain. The MVC pattern that we have implemented is presented in more details in Section 8.1.1.

As can be seen in Figure 7.1, the client side is composed of a number of distinct modules. These are described here in more detail:

7.1.1 Model

The Model encompasses all the business objects used by the system. When the system needs to present information about a business object, the client side requests the respective information. In turn, the server side services transfer the Model to the client side. When an update on the Model needs to be persistent, the client side sends the updated Model to the server side, initiating the corresponding procedures.

7.1.2 View

In the MVC pattern, the view is responsible for all the information presentation. Each view controls a number of widgets on the user's web browser. It contains the Action Handlers which listen to the user's actions, and the Templates that define the presentation of the widgets. More details on the implementation of the View can be found in Section 8.1.5.

7.1.3 Controller

The Controllers contain most of the client side logic of our application. They are the modules that respond to the user input and define the Views that comprise the visual effect on the user interface. Additionally, they maintain the Model and change it appropriately. For every distinct visual interface of the system there is a Controller, who in most cases handles a single view. Moreover there are several cases where a "composite" Controller manages a number of other Controllers, creating complex interfaces. The Controllers also access the browsers features that provide access to the Storage space, the FIle Uploads, the GPS location, etc.

7.1.4 Event Bus

The Event Bus implements the *Publish-Subscribe (Observer)* pattern, enabling the decoupling of the user interface components. It is responsible for the message transmission between the entities that reside on the Client Side. It provides mechanisms for publishing events and subscribing to events.

7.1.5 Application Manager

The Application Manager acts as a centralized point of control, handling the communication between the Controllers and the server side by making calls to the services exposed in the Service Layer, and notifying Presenters for their responses. It contains the *Data Wrapper*, which implements the Mediator pattern, handling all the communication between the Controllers and the Server Side. Moreover, it enables the seamless caching of the data on the client side with the use of a dedicated cache. On each data request, the Data Wrapper checks if it already exists in the cache before requesting it from the server, thus reducing the number of requests and speeding up the application execution. Apart from the Cache, the Data Wrapper contains the Data Adapters. More details on the implementation of the Data Adapters can be found in Section 8.1.7.

7.1.6 Router

The Router manages the URL of the client browsers. Its main purpose is to provide a different URL to each distinct interface, without raising a browser event that will force a reload on the whole page. Additionally, when the URL changes from other controls on the page (e.g. a click on a link), the Router analyses the new location and handles the transition to the new View. This contains a mapping between the different URLs supported in the system, as well as the Controller responsible for each mapped user interface. More details on the implementation of the Router can be found in Section 8.1.3.

7.2 Server Side Architecture

The Server Side part of our framework follows a multi-layered architectural pattern consisting of three basic layers: Service Layer, Business Logic Layer and Data Layer. This increases the system's maintainability, reusability of the components, scalability, robustness, and security. As shown in Figure 7.1, the server side is comprised of a number of distinct modules which will be described in the following sections.

7.2.1 Service Layer

The Service Layer controls the communication between the client logic and the server logic, by exposing a set of services (operations) to the client side components [14]. These services comprise the middleware concealing the application business logic from the client and have been build as RESTful [60, 35]. The basic system services are:

- *CRUD Services*: Facilitate the creation, retrieval, update and deletion of an observation, a multimedia object associated with an observation, a user etc.
- *External Access Services*: Provide the means for external systems to search and use the data of the system.
- *Search Services*: Provide the Client Side with the ability to search the data.
- *Multimedia Access Services*: Allow access to the uploaded multimedia files and their respective thumbnails.
- *Statistics Services*: Responsible for supplying the usage statistics of the observations and the objects.

7.2.2 Business Logic Layer

The Business Logic Layer, also known as Domain Layer, contains the business logic of the application and separates it from the Data Layer and the Service Layer.

- *The Persistency Management Module*, which manages the access to the data on the repository.
- *The Statistics Management Module*, which manages the statistics of the system.
- *The Search Management Module*, which responds to the queries on the data with the appropriate results.
- *The Multimedia Management Module*, which manages the persistence and the serving of the multimedia files, as well as their analysis and thumbnail generation.

7.2.3 Data Layer

The Data Layer accommodates external systems that are used to store data accessed by it. Such systems are the Cache Server which holds the cached information for faster access, the Data Repository which holds all the data of the system, the Search Index allowing faster full text queries, and the File Repository, persisting the multimedia files and the thumbnails. The implementation details of the components that comprise the Data Layer can be found in Section 8.2

Summary

In this chapter we presented the overall system architecture, identifying its basic components and providing an in depth analysis of their internal functionality.

Chapter 8

Implementation

The functionality of the system and the architecture have been implemented successful as described above. This chapter will describe the implementation details of some of the components in more detail, while also providing examples of source code. This way the reader can realize how specific parts of the system have been implemented. We have split the chapter into two Sections. Section 8.1 describes the Client Side, while Section 8.2 described the Server Side.

For the source control management of our application we have used multiple Git repositories hosted on Bitbucket [1]. Moreover, we have used Jenkins [66], which is a continuous integration server, allowing us to run automatic tests and deploy the applications easier and in a more controlled way.

8.1 Client Side

The client side is based on the latest web-application standards and relies on the JavaScript programming language. The client side refers to both the desktop web application and the mobile web application. Although these have different interfaces that are presented to the users, the underlying source code for the client side is almost identical, with the only exception for the user interface templates (see Section 8.1.2).

Additionally to these two systems, the mobile web application has also been packaged as a native mobile application with the use of Phonegap, making it compatible with all the major mobile platforms. This allows the application to run without the need of internet access, or the loading of its source each time that the user loads it. Additionally, Phonegap allows us to provide more functionality by using various native platform features that are otherwise unavailable to web appli-

[1] `http://bitbucket.org/`

cations.

The client side is built with JavaScript, HTML5 and CSS3. For the code organization of the client side we have used various open source libraries/frameworks. Section 8.1.1 presents the implementation of the MVC pattern in our applications. Section 8.1.2 contains some details about the way that the user interfaces have been implemented. Section 8.1.3 describes the parts that handle the user interface screen transitions, while Section 8.1.4 present the modular organizing of the source code. Section 8.1.5 introduces the implementational aspects of the Views in our applications, and Section 8.1.7 describes the Data Adapters. Finally, Section 8.1.8 describes the development process that we followed.

8.1.1 MVC Pattern

The client side of the application has been based heavily on the Model View Controller design pattern. Numerous JavaScript frameworks implementing the MVC pattern have emerged during the last years. Most of them force strict definition format rules to the various components due to the special handling that complex JavaScript libraries need. After having evaluated the most popular open-source frameworks, we decided to use *Backbone.js* [1].

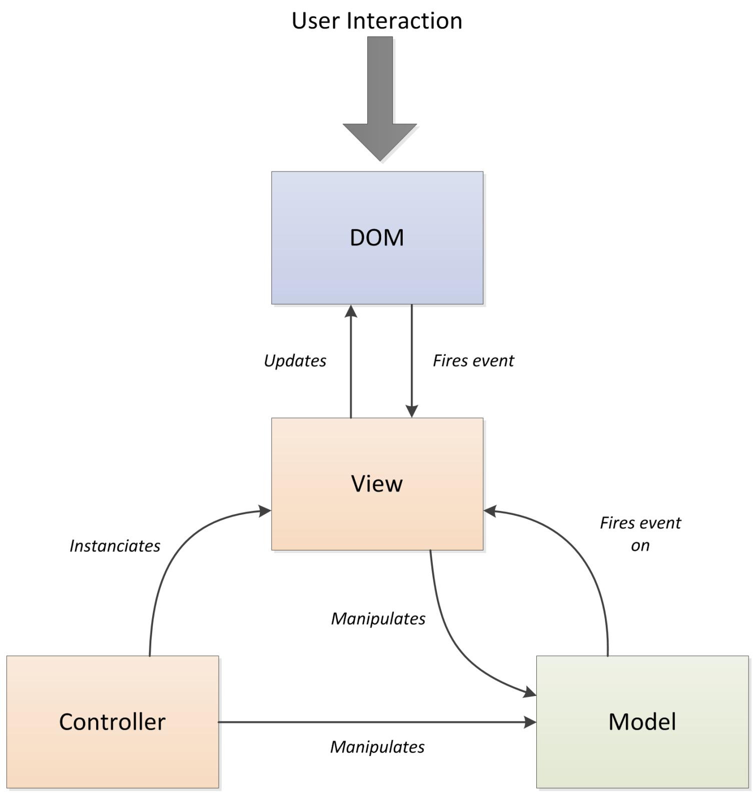


Figure 8.1: Model View Controller Pattern in our application.

Backbone.js [1, 53, 57] has been developed as an MVC framework for JavaScript applications. It's main purpose it to provide the basic structure to the application. It is very lightweight and extremely extensible, allowing developers to customize it according to their needs with minimum effort. Apart from the MVC features, it provides other useful functionality: models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

The use of Backbone's models, collections, events and views provides the application with a very basic "solid" structure. The models and the collections allow the application objects to act as object oriented classes with key-value binding and custom listeners on each of the properties. Moreover, the events allow custom events to be broadcasted from any part of the application, reducing the coupling of the components. Finally, the views allow the creation of independent widget and their composition into complex user interfaces.

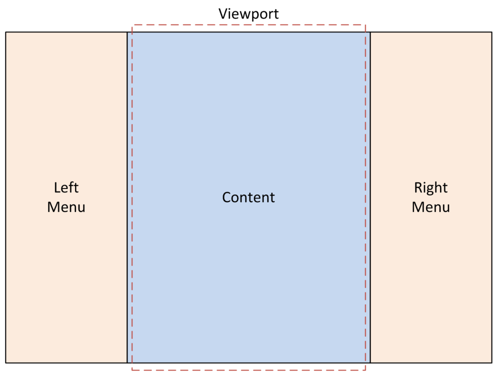
Figure 8.1 presents the MVC components and the interaction between them in our application. The Controller instantiates the View and manipulates the Model. In turn, the View listens for updates of the Model, and when these happen the View is re-rendered with the changes. Additionally, the View updated the Document Object Model (DOM) [42] when needed, as well as handles the interaction events that the user actions generate on the browser. In some cases the View also manipulates the Model.

8.1.2 User Interface

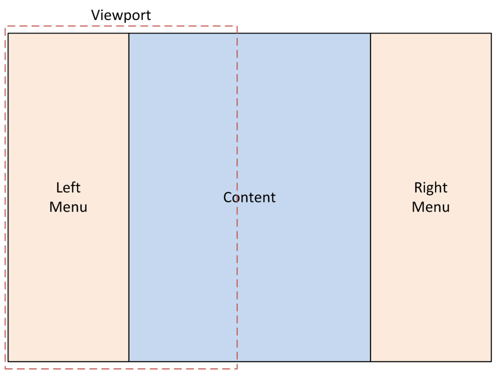
This section will describe come of the implementation details of the User Interfaces in our application.

When building an application optimized for mobile devices, the developer must take into account the small size of the screen which affects the usability of the applications. The menus that are provided with the mobile applications are at most times hidden and become visible only when the user needs them. This happens with the use of a button that reveals the menu. When building native applications for the mobile platforms, the Software Development Kits provide the functionality needed for creating such menus.

However, when building web applications for mobile devices, the developer needs to implement the functionality for the sliding menu by himself. There are a number of ready-to-use implementations available open-source, but they all have their disadvantages when it comes to performance, since they try to apply themselves to the general cases, providing more functionality than needed most of the times. To this end we developed a custom sliding menu, as minimal as possible with performance in mind.



(a) The viewport when the menu is hidden



(b) The viewport when the menu is visible

Figure 8.2: The main User Interface elements of our mobile application.

Our implementation uses the latest features of HTML5 and CSS3 for placing the elements and the visual effects. Figure 8.2 presents the main elements of our user interfaces. These are the two menus (right and left) and the main content area in the middle. When the menus are hidden, they are pushed away from the visible area (Viewport) of the document. When the menus are becoming visible, the whole canvas is moved to the left or the the right (depending on which menu is open), which causes one of the two menus to appear.

Templates

A very important aspect in providing extensible applications that handle information presentation is the use of templates. By using templates, we can isolate the code that generates the interfaces and update it without having to update the application logic [59, 68]. A lot of JavaSctipt templating libraries have been developed during the last few years. We chose Handlebars for our templates, due to it's powerful features and the simpliciy of the markup that it supports.

Handlebars allows the developer to write the HTML code for the user interface, while inserting special sequences in the places that he want dynamic content generation. During the compilation of the template to actual 'valid' HTML, the system provides the templates with the needed model, which are in turn used for the dynamic content.

An example of a template can be seen in Listing 8.1. We can immediately notice the if statement between lines 2 and 6. Moreover, line 11 prints the $fullName$ attribute. Apart from the template, the rendering requires an object in order to find the required attributes and print them. An example of an object that would work with this template can be seen in Listing 8.2.

Listing 8.1: Handlebars template

```
<div class="clearfix">
  {{#if hasThumbnail}}
    <div class="observation-image">
      <img src="{{thumbnailUrl}}"/>
    </div>
  {{/if}}
  <div class="pull-left">
    <div class="observation-label">{{label}}</div>
    {{#with user}}
      <div class="object-label">
        <span>by {{fullName}}</span>
      </div>
    {{/with}}
    <div class="observation-label">
      <span>{{creationTime}}</span>
    </div>
  </div>
  <div class="observation-gotobtn">
    <i class="fa fa-arrow-circle-o-right"></i>
  </div>
</div>
```

Listing 8.2: Handlebars template model example

```
{
    label: 'Observation of canis lupus',
    creationTime: 1984662854,
    thumbnailUrl: 'http://...',
    user: {
        fullName: 'Giannis Skevakis'
    }
}
```

The use of templates during the development of the mobile application allowed us to rapidly move to the desktop interfaces. In most parts of the applications we just needed to update the templates with the new markup to support desktop browsers.

Style

Another aspect of the web application development that has been really hard to achieve extensibility is the styling of the web pages. The part of the web applications that is responsible for the styling is the CSS, which can handle all the graphical details and the presentation of the elements that create a web page. CSS has been build with simplicity instead flexibility in mind. To overcome this issue, a lot of libraries have been developed, acting as pre-processors of CSS. These provide special markup language for the development, which is then compiled into valid CSS.

The library that we used for creating extensible styles in our application was LessCSS. LessCSS adds features that allow variables, mixins, functions and many other techniques, making CSS more maintainable, themable and extendable. An example of LessCSS code can be seen in Listing 8.3. We can see in line 1 that we can include a file inside another. Lines 3-7 contain variable definitions, and lines 10-17 contain the usual CSS rules, although the previously set variables are marked with the symbol '@'. The resulting CSS after the compilation of this can be seen in Listing 8.4.

Listing 8.3: LessCSS example

```
1  @import "mixins.less";
2
3  @body-bg:               #fff;
4  @text-color:            lighten(#000, 20%);   // #333
5  @font-family-base:      "Open Sans", Helvetica, Arial, sans-serif;
6  @font-size-base:        14px;
7  @line-height-base:      1.428571429;
8
9  body {
10   font-family: @font-family-base;
11   font-size: @font-size-base;
12   line-height: @line-height-base;
13   color: @text-color;
14   background-color: @body-bg;
15   height:100%;
16 }
```

Listing 8.4: Generated CSS

```
1  body {
2    font-family: "Open Sans", Helvetica, Arial, sans-serif;
3    font-size: 14px;
4    line-height: 1.428571429;
5    color: #333;
6    background-color: #fff;
7    height:100%;
8  }
```

With the use of LessCSS, we were able to define all the variables used in our CSS inside a single file. This allows any change in the color, text size, font, etc. to be performed directly from the specific file.

8.1.3 Routing

Modern web applications build with the latest technologies enable more powerful interaction with the users. Their features are very close to the desktop applications in terms of complexity and management. These applications are called *RIAs (Rich Internet Applications).*

A very important aspect in the development of RIAs, is the use of AJAX for the communication between the client and the server. This enables the generation and presentation of content without forcing the web browser to reload a new web page. JavaScript source code running on the user's browser is responsible for issuing AJAX calls to the server as well as manipulating the Document Object Model and presenting the data.

As JavaScript applications are becoming more complex, the are used to manipulate all of the interfaces presented to the user. However, two major problems have surfaced from this complexity. Firstly, the change of the interface using JavaScript needs special treatment in order to allow the browser's history support. This appeared because the web browsers pushed to the history whenever there was a change in the URL and a reload of the new one. In RIAs this is not always the case. The second problem is the ability of the users to create bookmarks for a specific user interface screen, which is not possible if the application does not propagate the interface change to the browser's URL.

In our system, these issues are handled by the Router component, which takes care of both the history of the browser and the mapping between the different screens of our application and their URLs. Our Router is based on the Router object provided by the Backbone.js library.

Figure 8.3 presents the interaction of the components in our application that take part in the routing. On the top we have the user's browser which includes the User Interface of the application managed by the DOM, as well as the URL. When the URL changes, an event is raised which is handled by the *URL Change Handler* of the Router. In turn, the Router contains a map which holds the correlation between the URLs and the Views. The change handlers find the respective View for each URL using the map, and initialize it. It is then passed to the *Application View* which is responsible for presenting it to user. The Application View is a special type of View which handles the main HTML content of the application by manipulating the DOM. The content of the Application View is composed of the menus and whichever view is at any time presented to the user. On each change of View, the Application View is responsible for safely discarding the previous View and attaching the new one to the DOM.

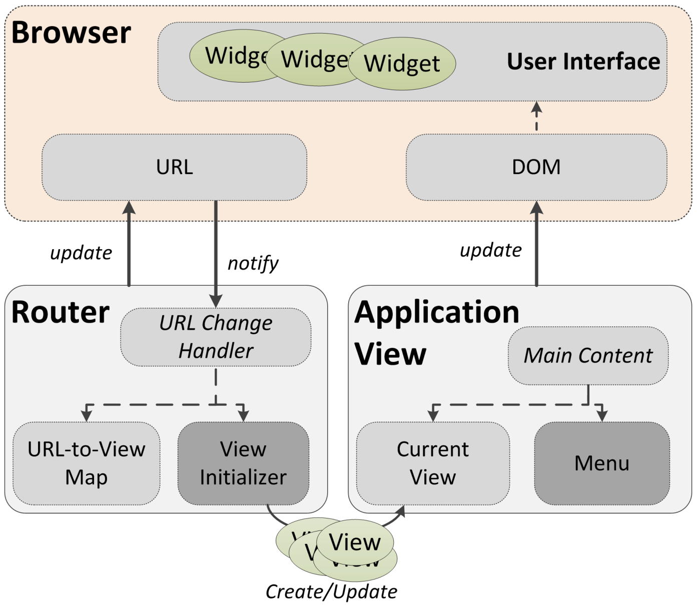


Figure 8.3: Routing of our application.

8.1.4 Modular Development

Modular programming is used to break large applications into smaller blocks of manageable code. Module-based coding eases the effort for maintenance and increases reusability. However, managing dependencies between modules is a major concern developers face throughout the application development process.

Large web applications often require a number of JavaScript files. Generally, they are loaded one by one using `<script>` tags. Additionally, each file can potentially be dependent on other files, which need to be included in the HTML code before that. The most common example would be jQuery [6] plugins, which are all dependent upon the core jQuery library. Therefore, jQuery must be loaded before any of its plugins.

To accomplish the modular design in our applications and allow the organizing of our code in smaller files as well as the management of the dependencies for each JavaScript file, we used the *RequireJS* [2] library. RequireJS is one of the most popular JavaScript module and file loader which is supported in the latest versions of popular browsers. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Rhino and Node.js.

In RequireJS we separate code into modules which each handle a single responsibility. Additionally, dependencies need to be configured when loading files. Using a modular script loader like RequireJS greatly improves the speed and quality of the code.

[2]`http://requirejs.org/`

The management of these dependencies between the modules is handled by RequireJS. The modules are defined in separate files along with their dependencies, and RequireJS takes on the responsibility to provide each module with the required dependencies at runtime. Modular loading also allows incremental "lazy" loading of the different parts of the application at the time that they are needed, instead of loading all the Javascript code at loading time. This can have a big impact on application loading and running, especially in recent web applications that rely heavily on Javascript. Another very important advantage of using RequireJS or any other module loader, is the isolation of the modular code inside functions. Due to the nature of JavaScript, objects are most of the times global, which can cause a lot of problems when different parts of the application use the same variable names. The way that RequireJS works is that it wraps the code inside a function, which isolates the objects inside a JavaScript closure (i.e. special context inaccessible by other parts of the code).

An example of a fully functional module in our application can be seen in Listing 8.5. In line 1 we define the dependencies used by our module. In line 2 we create a function that wraps the module's code and takes as input the objects that match the dependencies. RequireJS loads the dependencies defined and makes a function call to our module, providing the respective objects. The rest of the code is plain JavaScript code.

Listing 8.5: RequireJS example

```
1  define(['backbone', 'app', 'fastclick', 'leaflet',
       'templates/handlebars.helpers'],
2    function(Backbone, App, FastClick, L, handlebarsHelpers) {
3  
4      $(function() {
5        // initialize fastclick
6        FastClick.attach(document.body);
7  
8        // initialize leaflet image path
9        L.Icon.Default.imagePath = "img/leaflet";
10 
11       // create the app
12       window.app = App;
13       App.initialize();
14 
15       $('#loading-css').remove();
16     });
17   }
18 );
```

8.1.5 View

The Views of our application have been based on the View objects provided by Backbone.js, with some extensions.

Our extensions to the Views include the definition of two functions. The first one is `onAttach()`, which holds operations that need to access the DOM of the browser. This was needed because the `render()` function is called before the View is attached to the DOM, which caused undesired effects in the case where we needed to have listeners attached to the elements of our view. These listeners will not work if the elements are not attached on initialization time. That is why the normal usage of our Views includes the call of the `render()` function which creates the DOM element representing our View, followed by the attachment of this element to the DOM. Then, the `onAttach()` method is safely called.

The second addition to the Backbone's View is the close function, which is used just before the View is detached from the DOM. This is a crucial part of the client side logic, since it makes sure that the resources used by the View are cleared from memory and all the listeners are detached from the elements. Without this step, the generations of JavaScript zombies (objects that are not cleared by the garbage collector) is inevitable.

Listing 8.6 presents the sample code of a View in our application. On the top we see the dependencies managed by RequireJS. In lines 6-76 there is the definition of the View. Lines 13,14 define the model objects used by the View, and lines 16-19 holds the user interface events that the View listens for. In line 21 we can see the `initialize()` function that acts as a constructor, retrieving the required information from the data adapter and setting up the View Object. Following, in line 30 we can see the `render()` function which compiles the template (presented in Section 8.1.2) using the model objects. Line 37 defines the `onAttach()` function, which is called after the View has been attached to the DOM and is responsible for operations that require the DOM objects. Lines 54-60 present the two functions called when the user interface events occur, and finally, line 62 holds the `close()` function which is called just before the View is detached from the DOM, and is responsible for freeing up the memory and detaching the events from the DOM objects.

Listing 8.6: View example

```
1  define(['jquery', 'backbone', 'sly', 'solid', 'leaflet',
2    'hbs!templates/observations/observation', ... ],
3    function($, Backbone, Sly, solid, L, template, ...) {
4      "use strict";
5
6      var ObservationView = Backbone.View.extend({
7
```

```
    className: "view-container",

    // the id of the observation. Must be provided with constructor.
    observationId: null,

    multimedia: null,
    observation: null,

    events: {
      "click #backBtn": "goBack",
      "click #deleteBtn": "delete"
    },

    initialize: function() {
      this.player = new MultimediaPlayer();
      this.isLoading = true;

      this.observation =
          app.getDataAdapter().getObservation(this.observationId);

      window.addEventListener("resize", this.resize.bind(this));
    },

    render: function() {
      // Load the compiled HTML into the Backbone "el"
      this.$el.html(template(this.observation));

      return this;
    },

    onAttach: function() {
      for (i = 0; i < this.profile.observationAttributes.length;
          i++) {
        var atr = this.profile.observationAttributes[i];
        var value = this.observation.get(atr.key);
        if(value) {
          this.addContentBox(atr, value);
        }
      }
      for (i = 0; i < this.contentBoxes.length; i++) {
        if(typeof this.contentBoxes[i].onAttach === 'function') {
          this.contentBoxes[i].onAttach();
          }
        }

        this.initSlideshow();
      },

      goBack: function() {
        window.history.back();
      },

      delete: function(e) {
        ...
      },

      close: function() {
        for (var i = 0; i < this.contentBoxes.length; i++) {
          this.contentBoxes[i].close();
        }
        this.contentBoxes.length = 0;

        this.off();
        this.remove();
        delete this.$el;
        delete this.el;
      }
    });

    return ObservationView;
  }
);
```

8.1.6 Maps

For the maps used in our user interfaces, we used an open-source JavaScript library called Leaflet. Leaflet [3] is a minimal Javascript library, optimal for usage on mobile phones. The maps that are used here are provided by OpenStreetMaps [25].

[3] `http://leafletjs.com/`

8.1.7 Data Adapters

The Data Adapters in our application are responsible for handling all the data communication between the client side and the server side. As can be seen in the system's architecture (Figure 7.1), there exist two different types of adapters: the memory adapter and the web adapter.

The *Memory adapter* holds the data when there is no Internet connection, which is applicable only in the case of the native mobile application.

The *Web adapter* issues the AJAX calls to the server and gets the responses. An example to this implementation can be seen in Listing 8.7. Due to the nature of JavaScript and the asynchronous calls between the client side and the server, there is the need to adjust all the calls accordingly. To achieve that we used the Deferred object [24] provided by jQuery (line 2). This allows us to return a 'promise' to the function call, and call the `resolve()` function of this promise when the results have been received from the server (line 13). The rest of the code in lines 4-11 is the normal usage of the jQuery AJAX adapter.

Listing 8.7: Data adapter example

```
 1  getObservations: function(type, count, page) {
 2    var deferred = $.Deferred();
 3
 4    $.ajax({
 5      url: this.baseUrl + "/observation/type/" + type,
 6      type: 'get',
 7      data:{
 8        count: count,
 9        page: page
10      },
11      contentType: "application/json"
12    }).done(function(data) {
13      deferred.resolve(data);
14    });
15
16    return deferred.promise();
17  },
```

8.1.8 Development Process

For the development of the client side of our application we have followed a modern approach, using a lot of the powerful tools that are available to the developers. Two of these tools worth mentioning are *Node.js* [4] and *Grunt* [5].

Node.js [51, 69] is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Grunt is a task-based command-line tool that speeds up workflows by reducing the effort required to prepare assets for production. It does this by wrapping up jobs into tasks that are compiled automatically. Basically, one can use Grunt on most tasks that are considered trivial during the development process, which would otherwise have to manually configure and run. Grunt is build on top of Node.js. The list of tasks that Grunt can handle is exhaustive.

In our process, we have used Grunt to automate the needed tasks during the development and deployment stages. These involve the compilation and compression (minification) of the source code, the unit testing and the static source code checking. It is also used for the compilation of the application to the various formats supported by Phonegap/Cordova.

The organization of our source code can be seen in Listing 8.8.

Listing 8.8: Folder organization

```
app             // The main folder for our code and templates
--adapters      // The adapters (cordova, inmemory, localstorage)
--collections   // The collections of our models
--lib           // The external libraries used
--models        // The models of our application
--templates     // The templates used by our views
--vies          // The views

assets          // The static assets used in our application
--css           // The css files
--fonts         // The fonts
--img           // The images used

dist            // The distribution folder, populated with the files
    after the compilation of the sources and the minification

less            // The LessCSS files, which are compiled to CSS.
```

[4]http://nodejs.org/
[5]http://gruntjs.com/

8.2 Server Side

The server side (backend) of the system has been based on the Java programming language. More specifically, we used the Spring Framework extensively. The main features of Spring that we used were: the Inversion of Control container, the Spring Data framework, Spring Web and Spring Security. Section 8.2.1

8.2.1 Persistency

For the persistence of the data we have chosen a document database, MongoDB [7, 23]. MongoDB is a non-relational (NoSQL) database which allows us to represent the data of the system as JSON documents, while providing rich query expressiveness and selectivity. More information about MongoDB are presented in Section 3.2.

In recent years, non-relational databases are becoming more and more popular and have been replacing relational databases in a lot of cases, especially in big data and cloud based systems [22, 28, 61]. The technology behind most NoSQL databases is based on distributed and parallel systems [30, 47], as well as in-memory databases [31]. There has been a lot of evaluation and criticism on NoSQL databases [46, 40], and a lot of debate about the selection of the best database technology between the two [67, 58]. Comparing relational (SQL) databases to NoSQL, we can note the following:

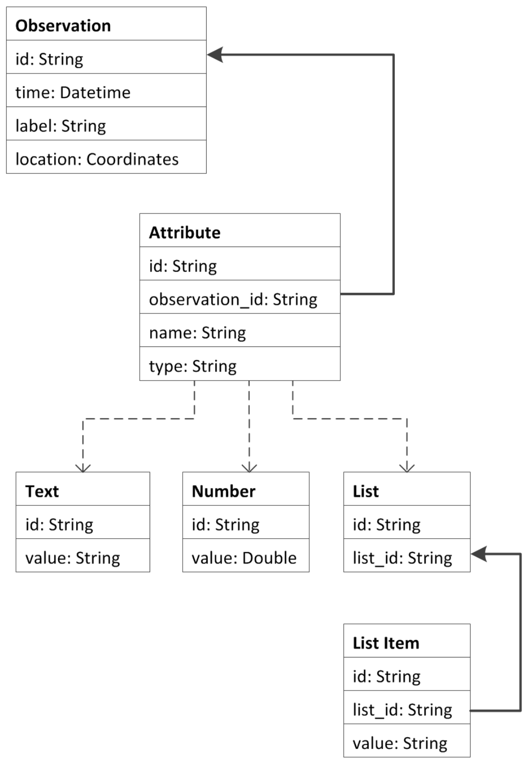
- *Schema*: Regular SQL platforms often have strictly enforced rules for a schema change, while most NoSQL platforms are schema-less, thus allowing any schema updates without any effort.
- *Queries*: SQL supports a growing subset languages for queries, as well as a wide range of filters, sorting options, and projections and index queries. NoSQL does all this as well, but SQL can often go beyond it, allowing powerful aggregations of your data as well, beyond what NoSQL can do.
- *Consistency*: SQL platforms often use a single master to guarantee strong consistency in the database. These use synchronous replication to ensure that important changes queued up to the master are not lost. NoSQL, by contrast, does replication of entity groups without a master, so that data is strong within an entity group, and is eventually updated across all groups. The better option depends on the constraints and needs of the database.
- *Scalability*: For years, database administrators relied on scaling up, buying bigger servers as database load increased. However, as transaction rates and demands on the databases

continue to expand immensely, emphasis is on scaling out instead. Scaling out is distributing databases across multiple hosts, and that is something NoSQL does better than standard SQL. They're designed for optimal use on scaled out databases.

- *Management*: NoSQL databases are generally designed to require less management overall. Repairs are often automatic, and data distribution and simpler data models contribute to less administration required overall.
- *Transactions*: Transactions are important because they ensure that the changes to the database are atomic. SQL databases support transactions, which is one of their most important and powerful feature. NoSQL platforms generally don't support transactions, which means that it is up to the developer to make sure that transactions are handled in the application instead of using the database features.

Since NoSQL is a very broad term, there are a number of different technologies and systems considered as NoSQL databases. As such, the NoSQL are further classified in the following categories: (*a*) *Document store*, (*b*) *Graph stores*, (*c*) *Key-value stores*, (*d*) *Object databases* and (*e*) *Column stores*. A lot of debate between them has also taken place [13]. MongoDB is the one that suited our needs more, and that is why we choose to use it for persisting our data. In more detail, here are a few reasons that lead us to choose MongoDB:

- *Schema-less*: Most of our data does not conform to a rigid relational schema. Hence, we can't bound it in the structure of a relational database and we need some more flexibility. A comparison between the relational schema we would need to implement if we used an SQL database and the actual format of our data is presented in Figure 8.5.
- *Querying*: Our flexible schema will need to support a lot of elasticity in the querying of the data. Using SQL, which is a strictly typed language, we would have to implement big and complex queries, and most of the times even whole procedures with multiple queries until we search for the results and create the objects. On the contrary, the query language supported by MongoDB is extremely flexible.
- *Transactions*: Although MongoDB does not support multi-document transactions, the schema-less design enables us to create multi-level 'nested' documents, taking advantage of the single document transactions and the atomic operations. MongoDB also supports two-phase commit
- *MapReduce*: MongoDB supports MapReduce operations on the data, giving a lot of power to the developer to implement data processing jobs.



(a) SQL implementation.

```
{
    "_id" : "52b4ad06e4b039de5f0f0f4a",
    "type" : "plants",
    "userId" : "52b98c0ee4b013c9e11dccb5",
    "creationDate" : "2013-12-20T20:48:06.966Z",
    "label" : "Observation of canis lupus",
    "location" : [
        37.983715,
        23.72931
    ],
    "attributes" : {
        "color" : "brown",
        "gender" : "male",
        "height" : 0.8,
        "object" : "52b4a398e4b041056bbc442b"
    }
}
```

(b) MongoDB implementation.

Figure 8.4: Comparison between SQL and MongoDB schemas. If we were to extend the model and add an additional property to the observation, in the case of the relational schema we would need to add rows to at least two tables, whereas in the MongoDB implementation we just need to add the key-value pair.

- *Geospatial*: A particularly powerful feature of MongoDB is its support for geospatial indexes. This allows the storage of x and y coordinates within documents and operations like $near$ a set of coordinates or $within$ a box or circle.

Examples of the data as persisted in the database are presented in Appendix A. We have divided the model to a number of entities, with each entity represented by a collection in MongoDB. The most important of these entities are: *Observation, Object, Multimedia, User, Profile, Statistics*. Apart from JSON documents, MongoDB can also persist blobs (files), which allowed us to put the multimedia files and the thumbnails along with model collections.

8.2.2 Indexing

For indexing and searching purposes, the data is also pushed into an Elasticsearch [3] instance. Elasticsearch is based on Lucene, providing a distributed, multitenant-capable full-text search engine with a RESTful web interface and schema-free JSON documents. Every part of the data that

```
SELECT Observation.*
FROM
  Observation
    JOIN
  Observation_Attribute
    JOIN
  Attribute
WHERE
  Attribute.name = 'height'
   AND
  1 > (
        SELECT
          value
        FROM
          Number
        WHERE
          id = Attribute.id
      );
```

(a) SQL sample query.

```
db.observations.find(
    {
        attributes: {
            height: {
                $lt: 9.95
            }
        }
    }
)
```

(b) MongoDB sample query.

Figure 8.5: Comparison between SQL and MongoDB queries. We can notice that the query language of MongoBD is much more flexible, which make it ideal for supporting queries on our data.

is searchable is indexed in an Elasticsearch cluster.

Our index server can be distributed in many machines due to the flexibility that Elasticsearch provides. Our source code takes as a configuration the URL of the server and the port that it is running, so there is no need to be hosted on the same server as our application.

8.2.3 Caching

Another very important part of the server side architecture is the caching system. To this end we have used the Redis server [8], which is an open-source, networked, in-memory, key-value data store. A very important feature that is provides is the optional durability to any data that it holds. This allows us to define the maximum size of main memory that we want to allocate to the cache, while Redis handles the removal of the least used elements when the limits are reached. The usage of the server cache increases the response time of the application when the clients request objects that are in the cache.

Redis has gained a lot of followers in recent years, and it is being used along with Database Management Systems to cache data and provide faster access times [72].

8.2.4 Security

Security is a very important aspect in every system, especially the web-based applications. More importantly, applications based on AJAX and Web 2.0 technologies pose a lot of possible security vulnerabilities due to the nature of the protocols that they use for transferring data [64].

For securing our infrastructure, we have used the security capabilities provided by the Spring Framework. These include the cookie based identification of the users of the system and the login using the RESTful login service. The important services that require the users to be logged in in order to use them check for the existence of the user data in the HTTP session, and deny access otherwise. However, we have also implemented some services that do not require user identification so that they can be used by external applications without problems or need for identification.

Summary

In this chapter we described the implementation details of the most important components. Additionally, we provided source code listings with examples of our implementation.

Chapter 9

Graphical User Interface

This chapter presents the methodology followed for designing the user interfaces of the applications, as well as the final product. We start by describing the initial phase of prototyping the interfaces in Section 9.1. Section 9.2 presents the interfaces as seen by real users.

9.1 Design Process

At the initial stages of the design process, we drew some prototypes of the user interfaces and created storyboards to visualize and organize our ideas. A storyboard is a representation of a particular interaction sequence [55]. Storyboards enable the better visualization of the interaction and the navigation between screens, while presenting most of the functionality of the system. Moreover, they enable brainstorming and allow changes to occur on the fly if a problem is found.

In our user interface design process, the storyboards were used in multiple heuristic evaluations with random users familiar with mobile devices. We were able to receive feedback from these evaluations and refine our design, making it more user friendly and efficient.

Figure 9.1 shows one of the storyboards that were designed. Some more user interface prototypes are presented in Figures 9.2,9.13,9.4,9.5. Notice that each prototype has a number and a name. The name is used as a general indication of the role of the interface and the number is used to indicate the transition between the different interfaces.

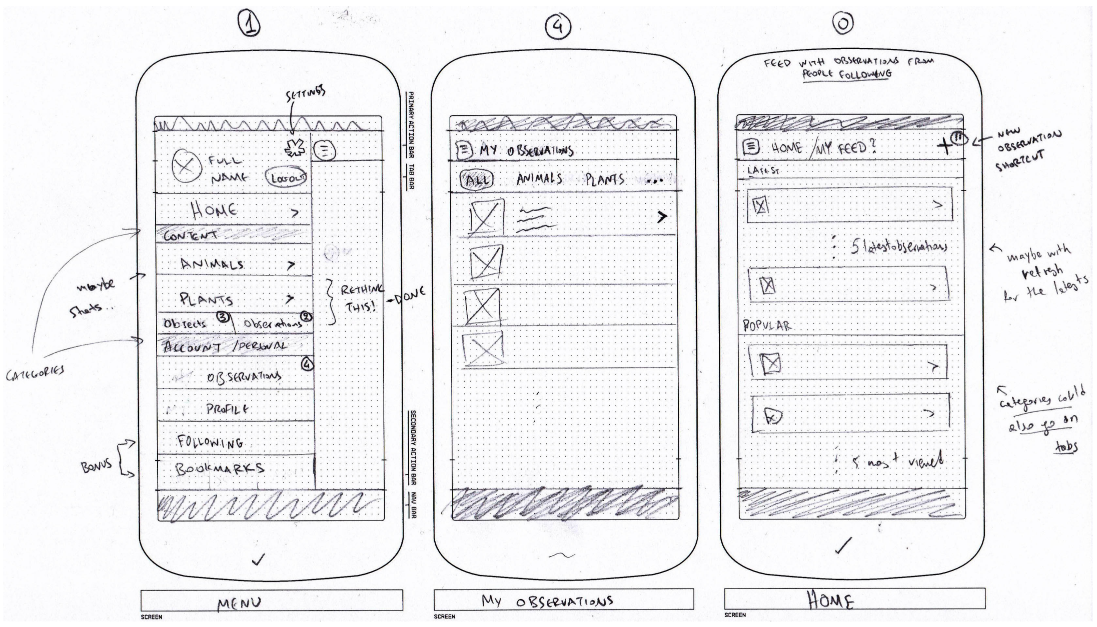
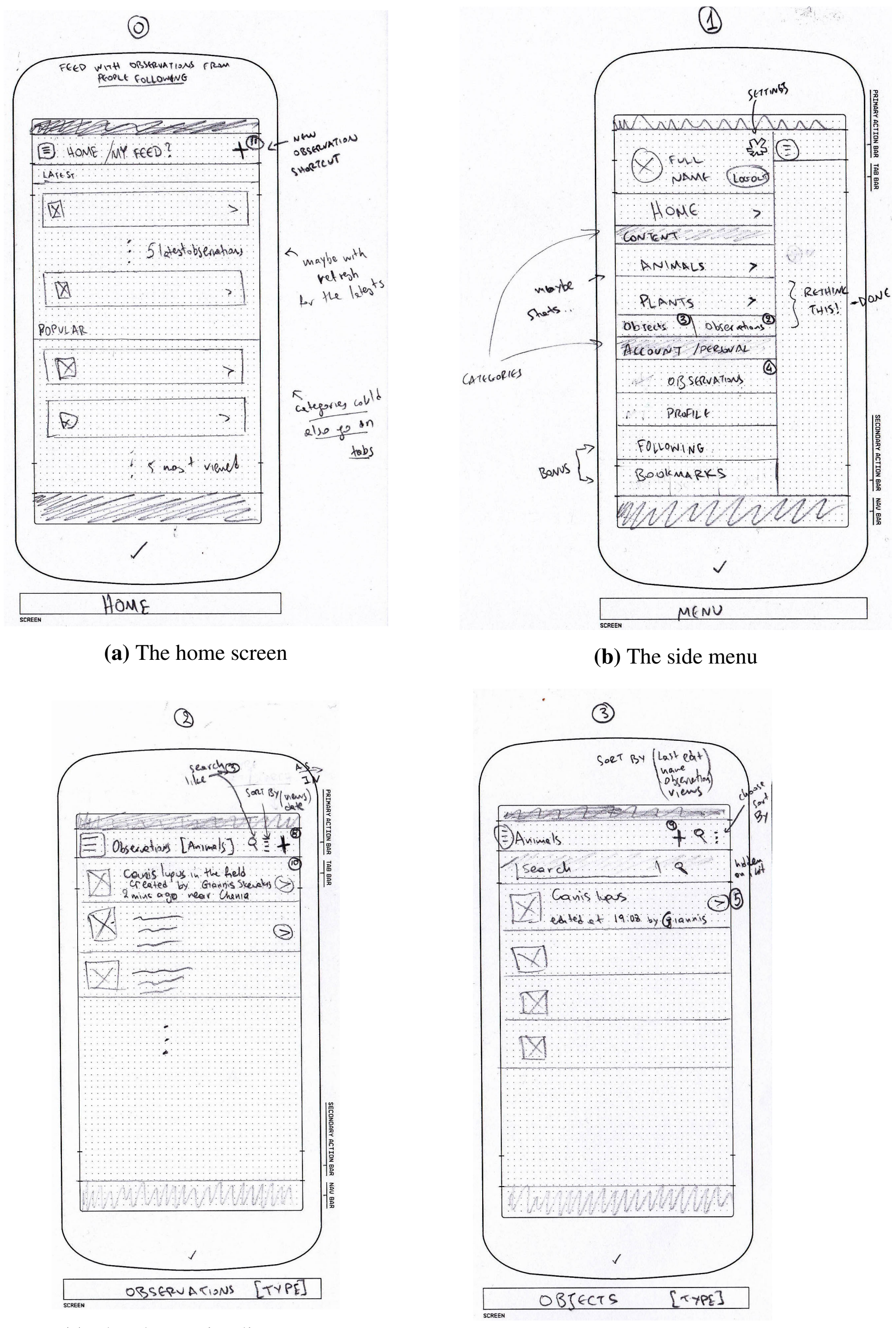


Figure 9.1: Storyboard sketch. The numbers on the user interface elements show the interaction and the next interface that needs to be presented when they are clicked.



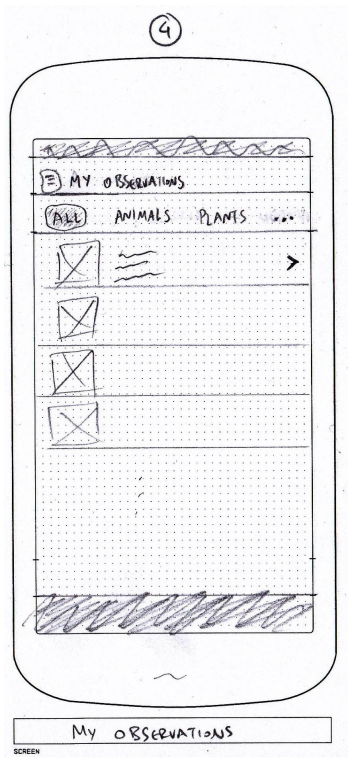
(a) The home screen

(b) The side menu

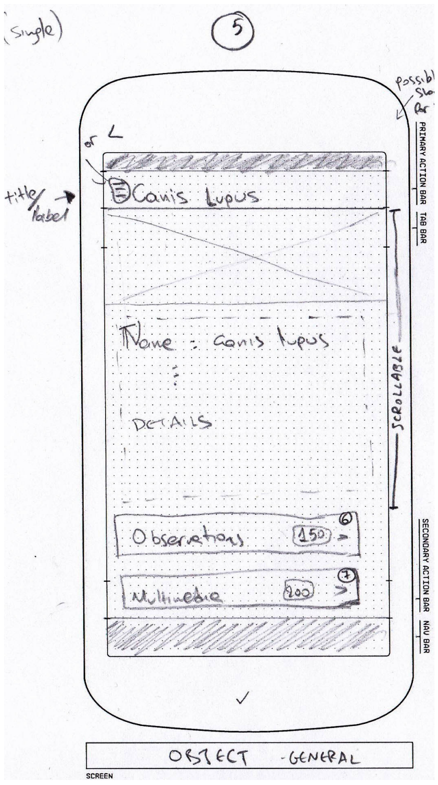
(c) The observation list screen

(d) The object list screen

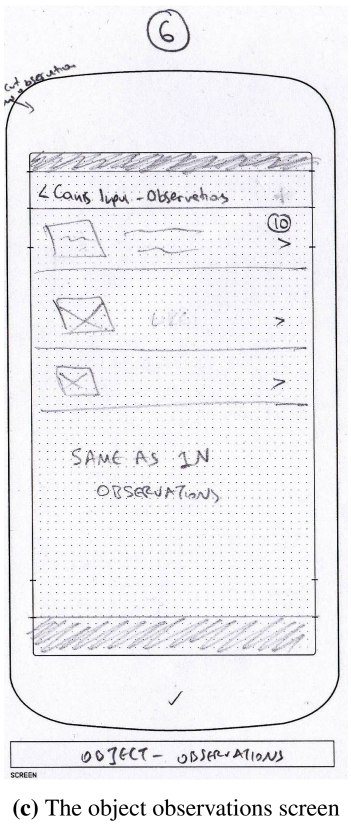
Figure 9.2: User Interface Prototypes.



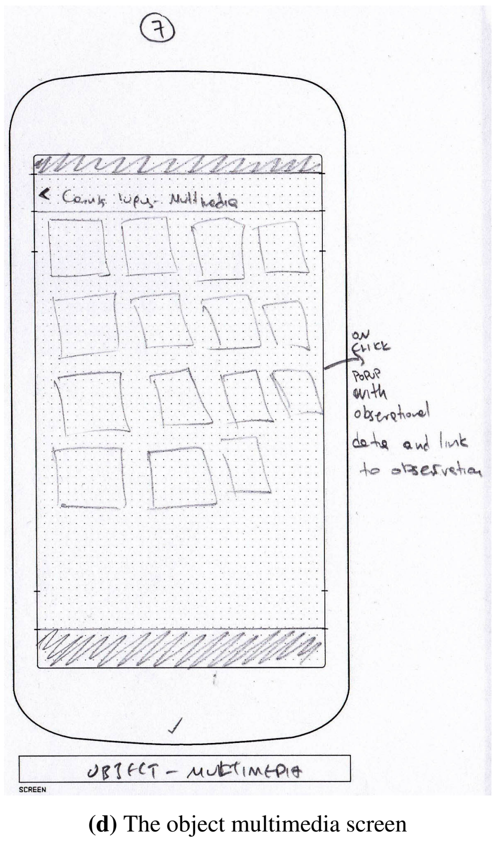
(a) The user's observations screen



(b) The object details screen

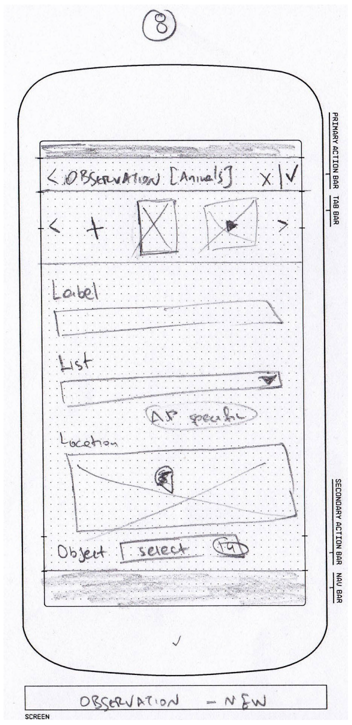


(c) The object observations screen

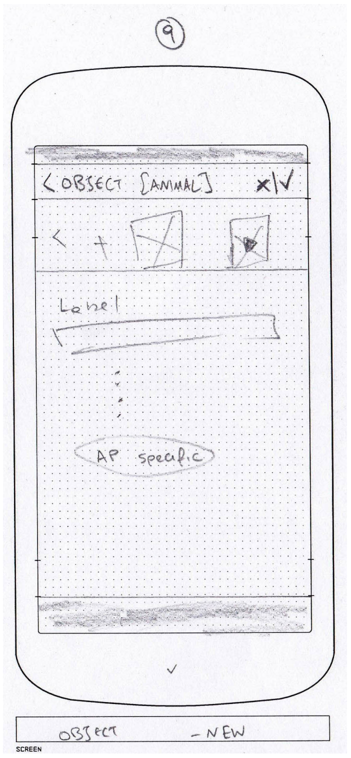

7
< Canis lupus Multimedia
on click popup with observational data and link to observation
OBJECT - MULTIMEDIA
SCREEN

(d) The object multimedia screen

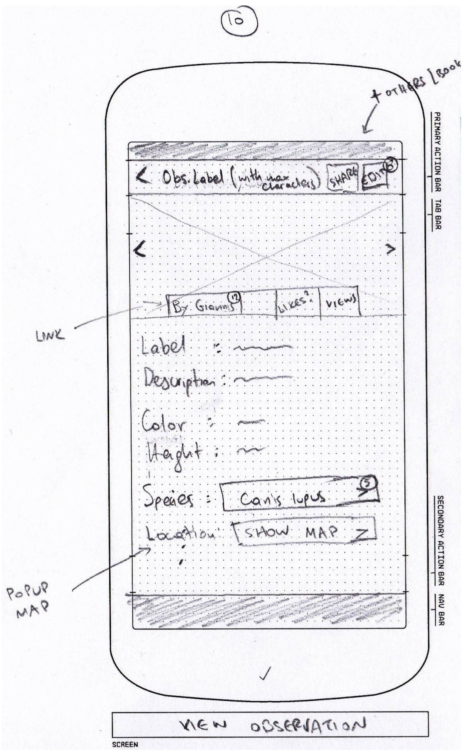
Figure 9.3: User Interface Prototypes (cont'd).



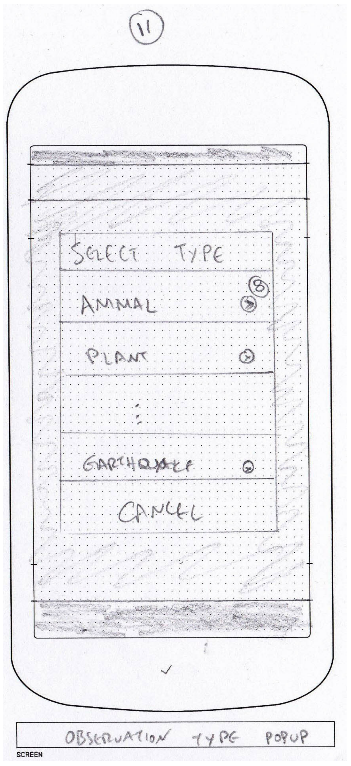
(a) The new observation screen



(b) The new object screen

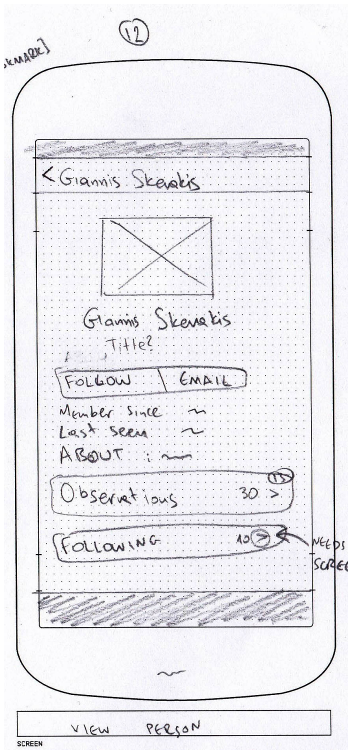


(c) The observation details screen

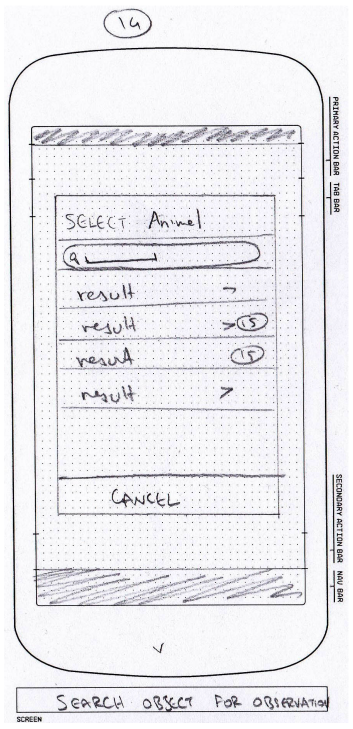


(d) The new observation type select screen

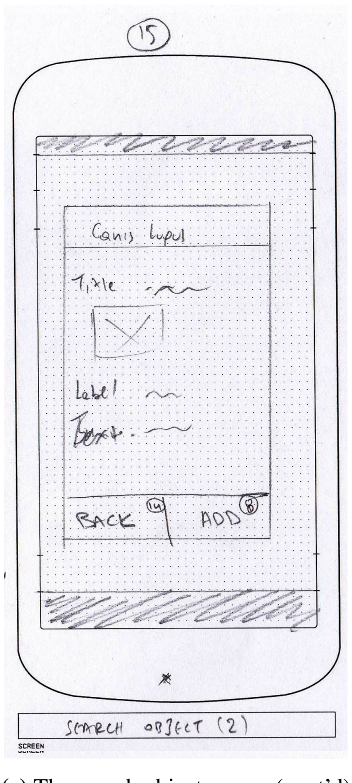
Figure 9.4: User Interface Prototypes (cont'd).



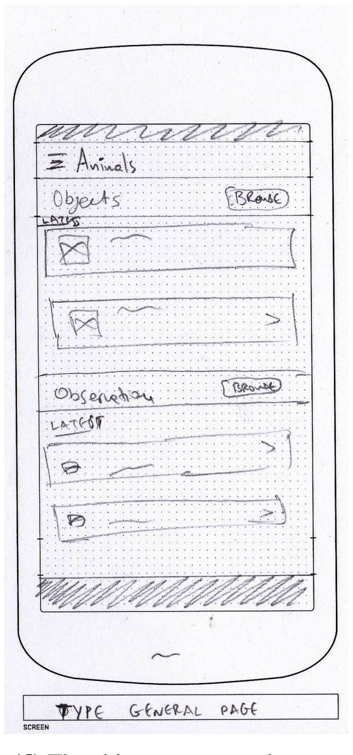
(a) The user details screen



(b) The search object screen



(c) The search object screen (cont'd)



(d) The object type general screen

Figure 9.5: User Interface Prototypes (cont'd).

9.2 User interfaces

The following sections presents some of the graphical user interfaces of the system by describing some of the tasks that the user can complete using the systems. First we will describe the mobile interfaces in Section 9.2.1, and then the desktop interfaces in Section 9.2.2

9.2.1 Mobile

This section refers to the interfaces presented to the user both when using the mobile application or opening the system's url using a mobile browser. There are some minor differences between the two versions in the user interfaces, which will be highlighted for the better understanding of the reader.

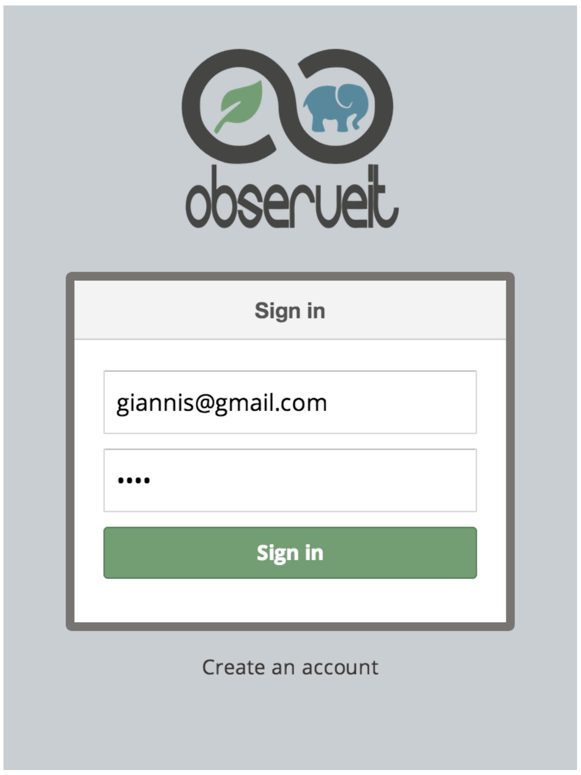


Figure 9.6: Mobile Interface: Login screen

Login

When the user opens the application for the first time, the login window (Figure 9.6) is presented, prompting him to enter his credentials before continuing to the home screen of the tool. The user also has the option to create a new account.

Create account

When the user opens the application for the first time, the login window (Figure 9.7) is presented, prompting him to enter his credentials before continuing to the home screen of the tool. The user also has the option to create a new account.

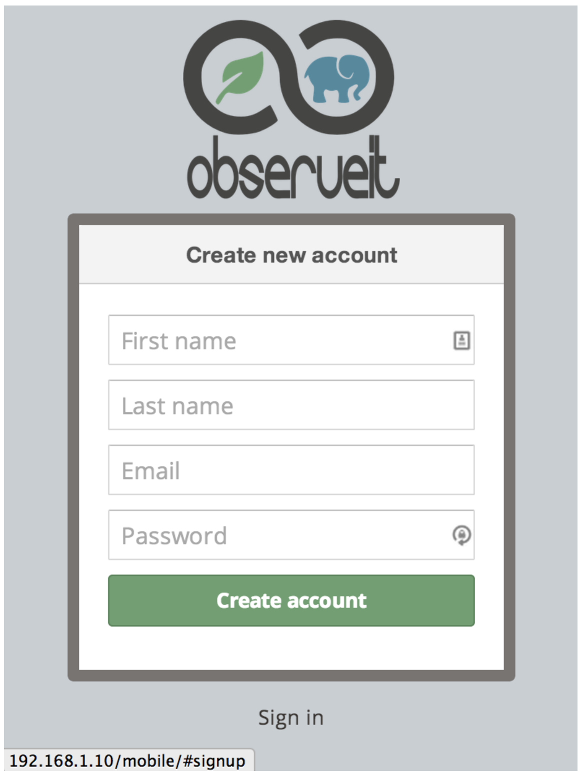
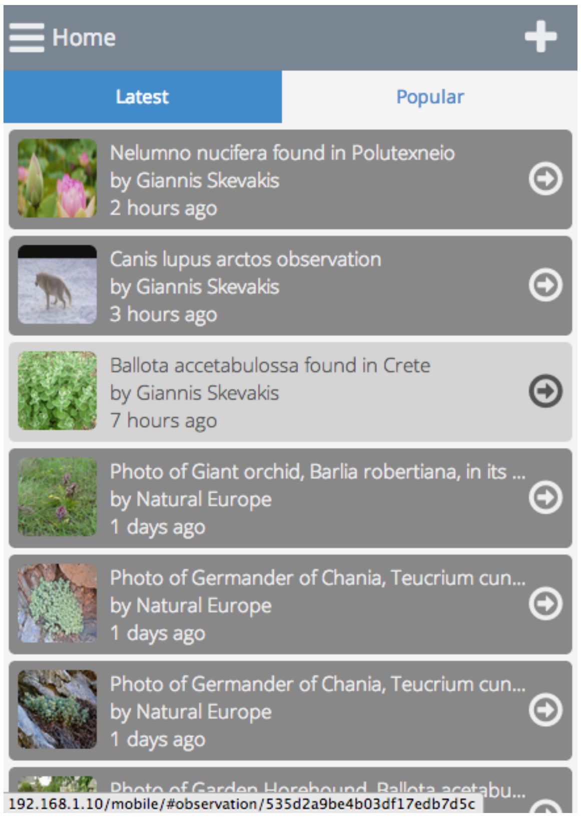


Figure 9.7: Mobile Interface: Account creation screen

Home Page

After a successful login, the user is presented with the home page of the application (Figure 9.8). The home screen contains a list of the latest observations made by the users, as well as a list of the most popular observations in terms of views. On the top right corner of the screen, there is a shortcut allowing the user to create a new observation right away.
On the top left, we find the menu button, which opens the sliding menu as shown in Figure 9.9. On the top side of the menu we can see the user's profile information along with the image he has provided. Below we find the list of the Object Types supported by the Application Profiles of the system, and even lower there are the links to the user's profile update page and the list of his observations.



Nelumno nucifera found in Polutexneio
by Giannis Skevakis
2 hours ago

Canis lupus arctos observation
by Giannis Skevakis
3 hours ago

Ballota accetabulossa found in Crete
by Giannis Skevakis
7 hours ago

Photo of Giant orchid, Barlia robertiana, in its ...
by Natural Europe
1 days ago

Photo of Germander of Chania, Teucrium cun...
by Natural Europe
1 days ago

Photo of Germander of Chania, Teucrium cun...
by Natural Europe
1 days ago

Photo of Garden Horehound, Ballota acetabu...

192.168.1.10/mobile/#observation/535d2a9be4b03df17edb7d5c

Figure 9.8: Mobile Interface: Home screen

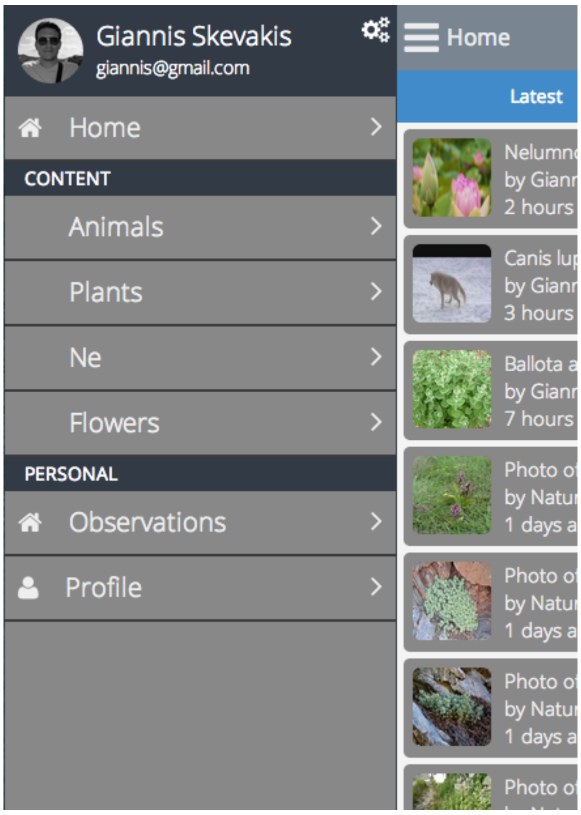
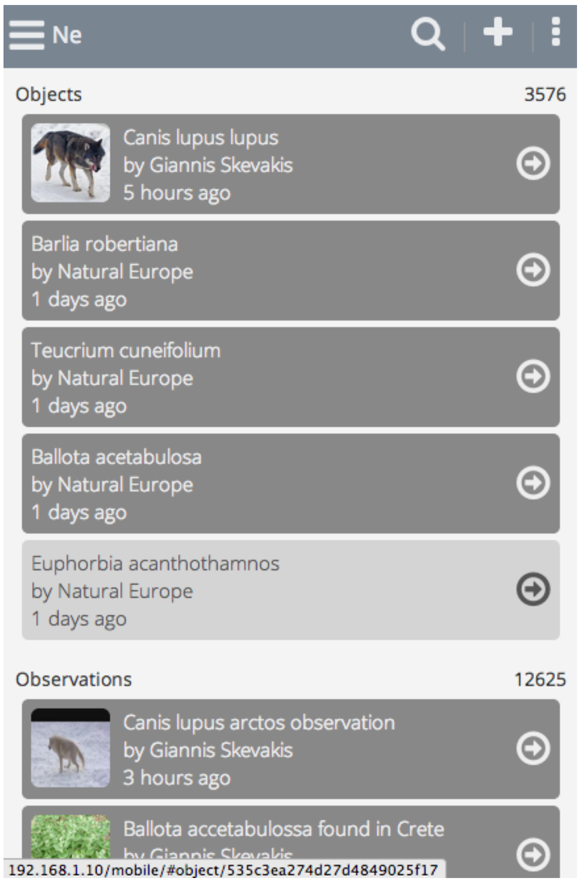


Figure 9.9: Mobile Interface: Menu

Browse and Review Objects


Ne

Objects 3576

Canis lupus lupus
by Giannis Skevakis
5 hours ago

Barlia robertiana
by Natural Europe
1 days ago

Teucrium cuneifolium
by Natural Europe
1 days ago

Ballota acetabulosa
by Natural Europe
1 days ago

Euphorbia acanthothamnos
by Natural Europe
1 days ago

Observations 12625

Canis lupus arctos observation
by Giannis Skevakis
3 hours ago

Ballota accetabulossa found in Crete
by Giannis Skevakis

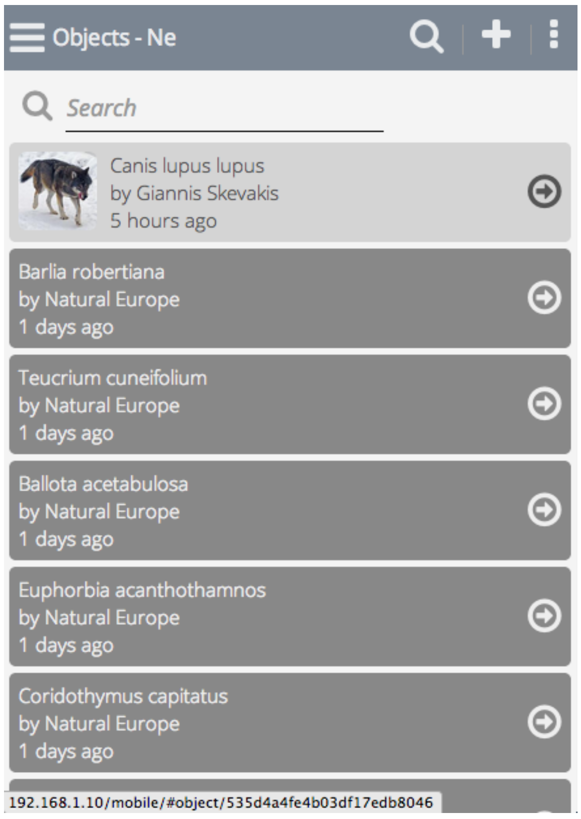
192.168.1.10/mobile/#object/535c3ea274d27d4849025f17

Figure 9.10: Mobile Interface: Object type

In order for the user to browse the objects of a specific type, or search for a specific one, he has to choose the right type from the menu. Then he is presented with the main screen of the type as shown in Figure 9.10, which contains the latest objects and observations for the type. He can then choose to see all of the objects and he is transfered to the object list (Figure 9.11).

From this screen he can use the search functionality to search for a specific object, or scroll through the list. The specific screen implements infinite scroll, meaning that when the user scrolls near the end of the list, the system automatically fetches the next results and appends them to the end. This way we do not need paging support and the user can browse through all the data just by scrolling.

Having found the object that he wants to review, he can then select it and he is shown the details of the specific object as shown in Figure 9.12. On the top of the screen we can see the multimedia files that follow the object. Just below there is the details about the user that created the object



Canis lupus lupus
by Giannis Skevakis
5 hours ago

Barlia robertiana
by Natural Europe
1 days ago

Teucrium cuneifolium
by Natural Europe
1 days ago

Ballota acetabulosa
by Natural Europe
1 days ago

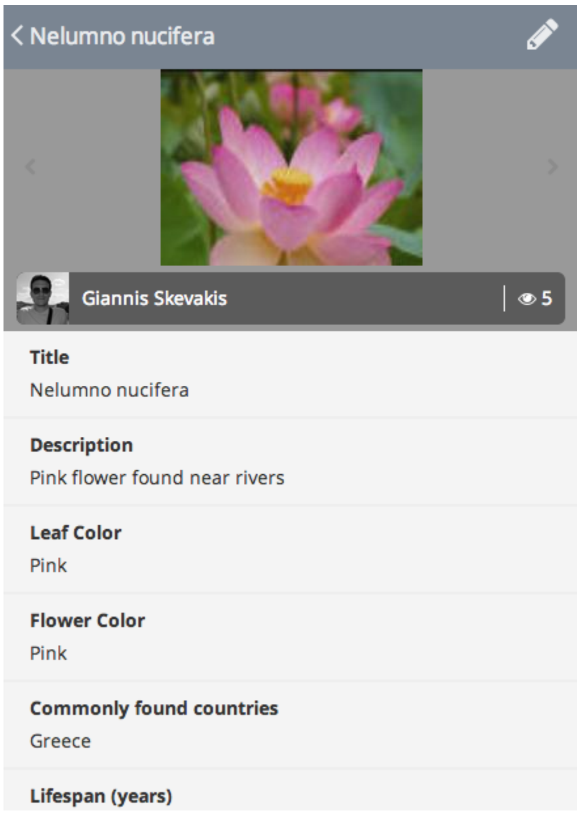
Euphorbia acanthothamnos
by Natural Europe
1 days ago

Coridothymus capitatus
by Natural Europe
1 days ago

192.168.1.10/mobile/#object/535d4a4fe4b03df17edb8046

Figure 9.11: Mobile Interface: Object list

and the number of times that it has been viewed. Below we find the metadata of the object as contributed by the creator. At the bottom of the page there exist two links. The first one redirects to the list of the observations about this object(Figure 9.13a), and the other one redirects to the list of the multimedia files that have been created along with the observations (Figure 9.13b).


< Nelumno nucifera

Giannis Skevakis | 5

Title
Nelumno nucifera

Description
Pink flower found near rivers

Leaf Color
Pink

Flower Color
Pink

Commonly found countries
Greece

Lifespan (years)

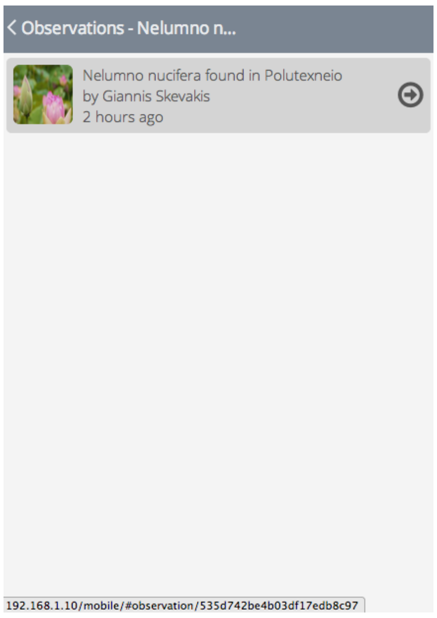
Figure 9.12: Mobile Interface: Object review

Browse and Review Observations

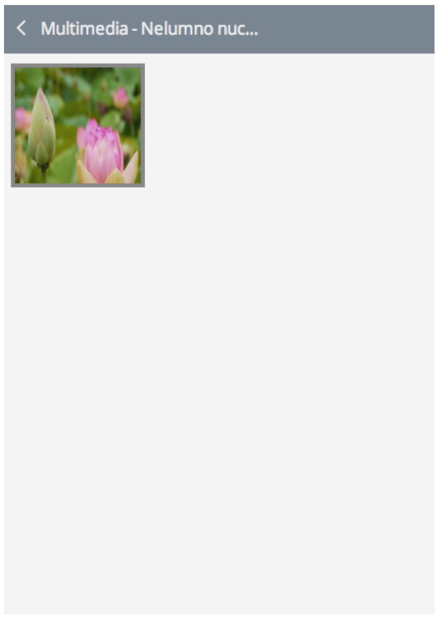
In order for the user to browse the observations of a specific type, or search for a specific one, he has to choose the right type from the menu. Then he is presented with the main screen of the type as already presented in Figure 9.10, which contains the latest objects and observations for the type. He can then choose to see all of the observations and he is transfered to the observation list (Figure 9.14).

From this screen he can use the search functionality to search for a specific observation, or scroll through the list. This screen implements infinite scroll as well, as described above.

Having found the observation that he wants to review, he can then select it and he is shown the details of the specific observation as shown in Figure 9.15. On the top of the screen we can


Observations - Nelumno n...
Nelumno nucifera found in Polutexneio
by Giannis Skevakis
2 hours ago
192.168.1.10/mobile/#observation/535d742be4b03df17edb8c97

(a) Observations concerning a specific object



(b) Multimedia concerning a specific object

Figure 9.13: Mobile Interface: Observations and Multimedia about object

see the multimedia files that have been created with the observation. Just below there is the details about the user that created the observation and the number of times that it has been viewed. Below we find the metadata of the observation as contributed by the creator. We can notice the different metadata fields that have been filled, as well as the map showing the geographic location that the observation was made.

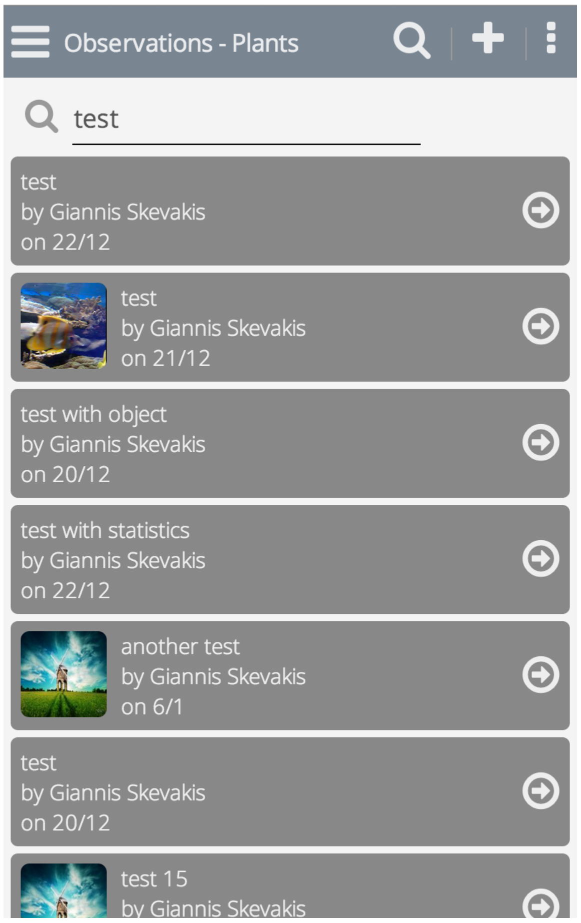
Create new Object

In order for the user to create a new object, he needs to start by choosing an object type from the menu. Then, from the main screen of the type, he can click on the '+' sign on the top bar, which presents him with the new observation interface, as shown in Figure 9.16a

Create new Observation

In order for the user to create a new observation, he needs to start by choosing an object type from the menu. Then, from the main screen of the type, he can click on the '+' sign on the top bar, which presents him with the new observation interface, as shown in Figure 9.16b

Each object and observation can contain one or more multimedia objects. When the user clicks on the plus button, he is presented with a multimedia upload screen, depending on the device and which of our applications he is using. That is, if he is using the native mobile application, he is



test
by Giannis Skevakis
on 22/12

test
by Giannis Skevakis
on 21/12

test with object
by Giannis Skevakis
on 20/12

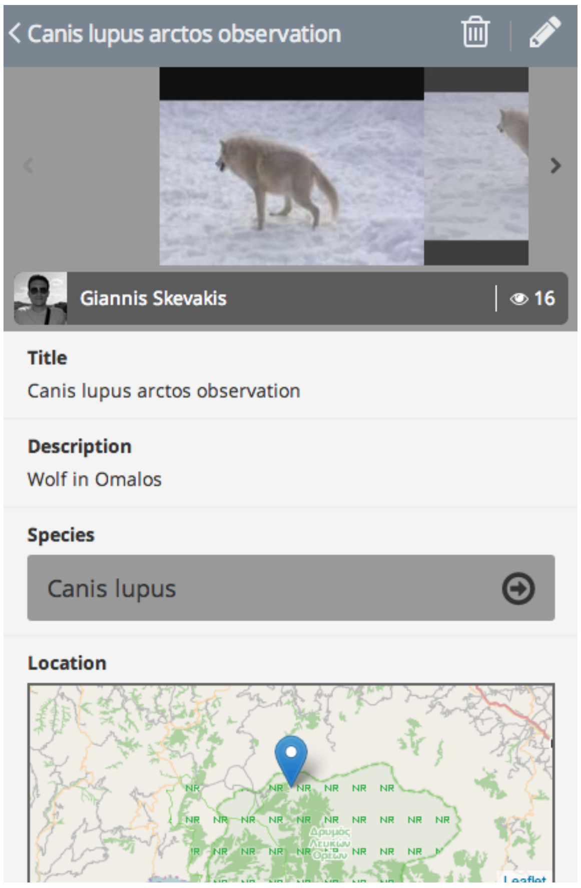
test with statistics
by Giannis Skevakis
on 22/12

another test
by Giannis Skevakis
on 6/1

test
by Giannis Skevakis
on 20/12

test 15
by Giannis Skevakis

Figure 9.14: Mobile Interface: Observation list



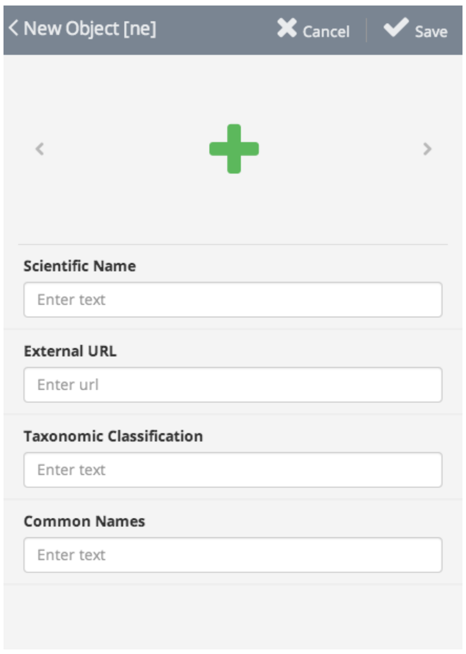
Title
Canis lupus arctos observation

Description
Wolf in Omalos

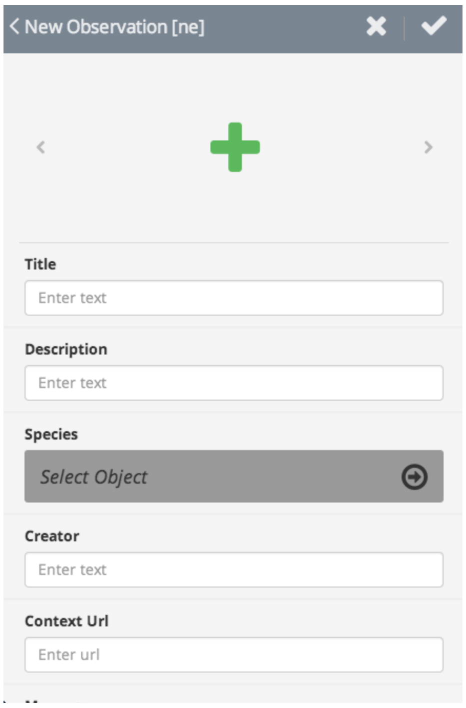
Species
Canis lupus

Location

Figure 9.15: Mobile Interface: Observation review



(a) Create Object



(b) Create Observation

Figure 9.16: Mobile Interface: Create Object and Observation.

presented with the screen in Figure 9.17a. In the case that he is using the mobile web application, he is presented with the screen in Figure 9.17b.

Browse Own Observations

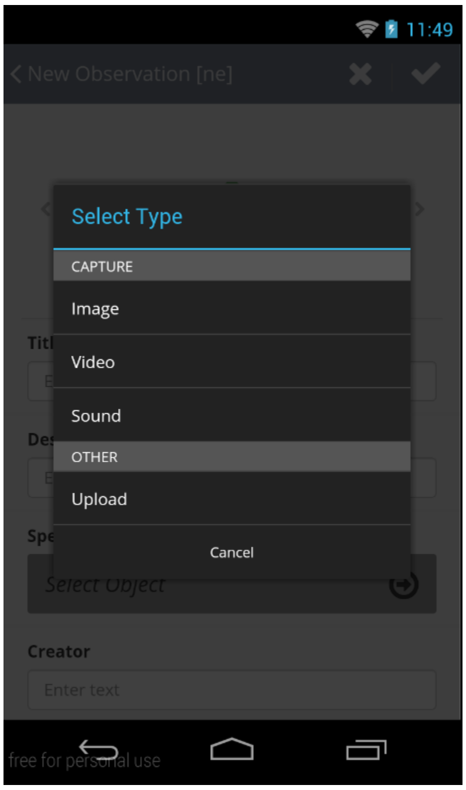
If the user wants to review the observations that he has submitted to the system, he needs to follow the 'My Observations' link of the menu. He is then presented with the list of his observations (Figure 9.18), and he can choose to review any of them.

Update Profile

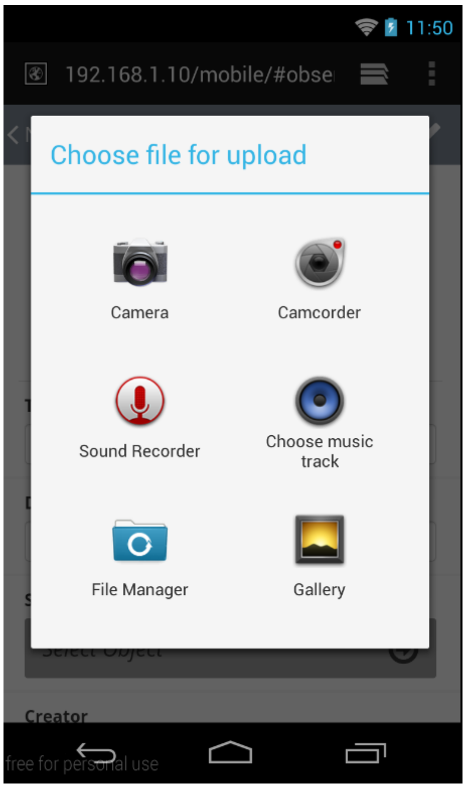
In the case that the user wants to edit his profile or upload a new profile picture, he needs to follow the 'Profile' link of the menu. He is then presented with the pre-filled form of his profile (Figure 9.19) and he can edit any information that he wants.

Review User's Profile

If the user wants to see the profile of another user of the system, he needs to follow the links that exist in the object and observation details pages. He is then presented with the user profile interface

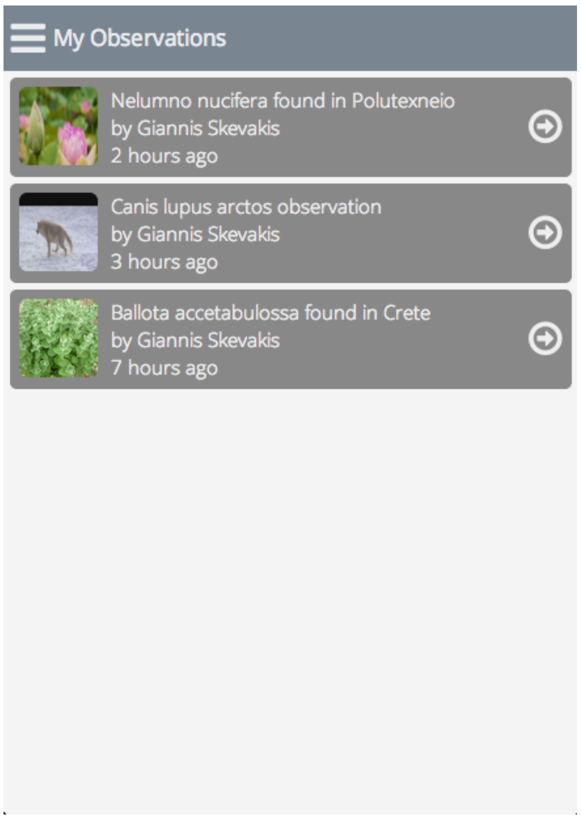


(a) Capture Multimedia



(b) Upload Multimedia

Figure 9.17: Mobile Interface: Upload/Capture multimedia for Object / Observation.

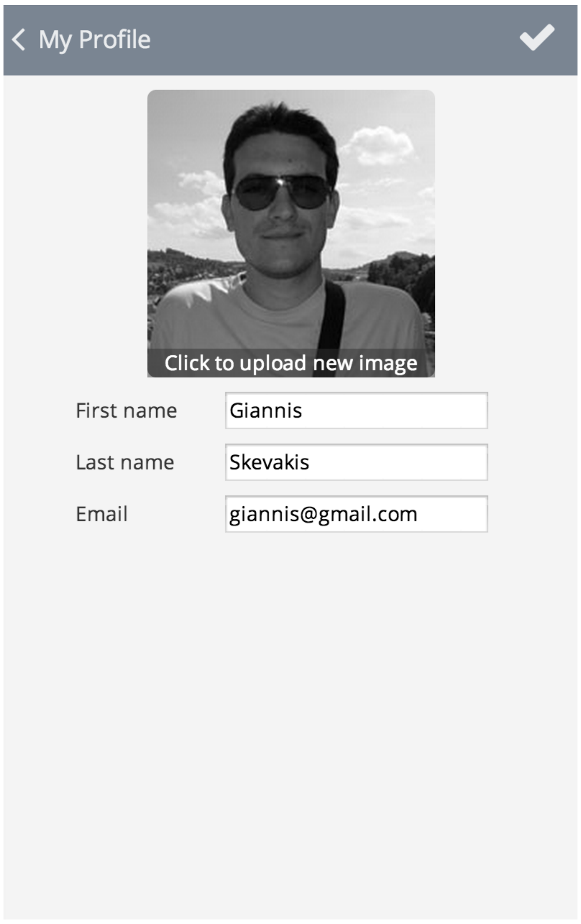

My Observations

Nelumno nucifera found in Polutexneio
by Giannis Skevakis
2 hours ago

Canis lupus arctos observation
by Giannis Skevakis
3 hours ago

Ballota accetabulossa found in Crete
by Giannis Skevakis
7 hours ago

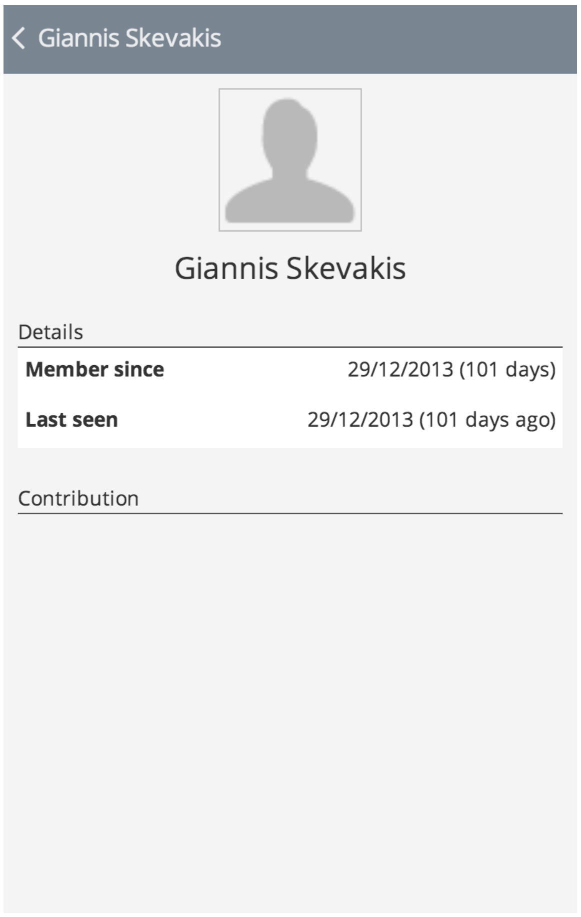
Figure 9.18: Mobile Interface: User's observations



First name	Giannis
Last name	Skevakis
Email	giannis@gmail.com

Figure 9.19: Mobile Interface: User's profile edit

(Figure 9.20), where he can see some details about the specific user.



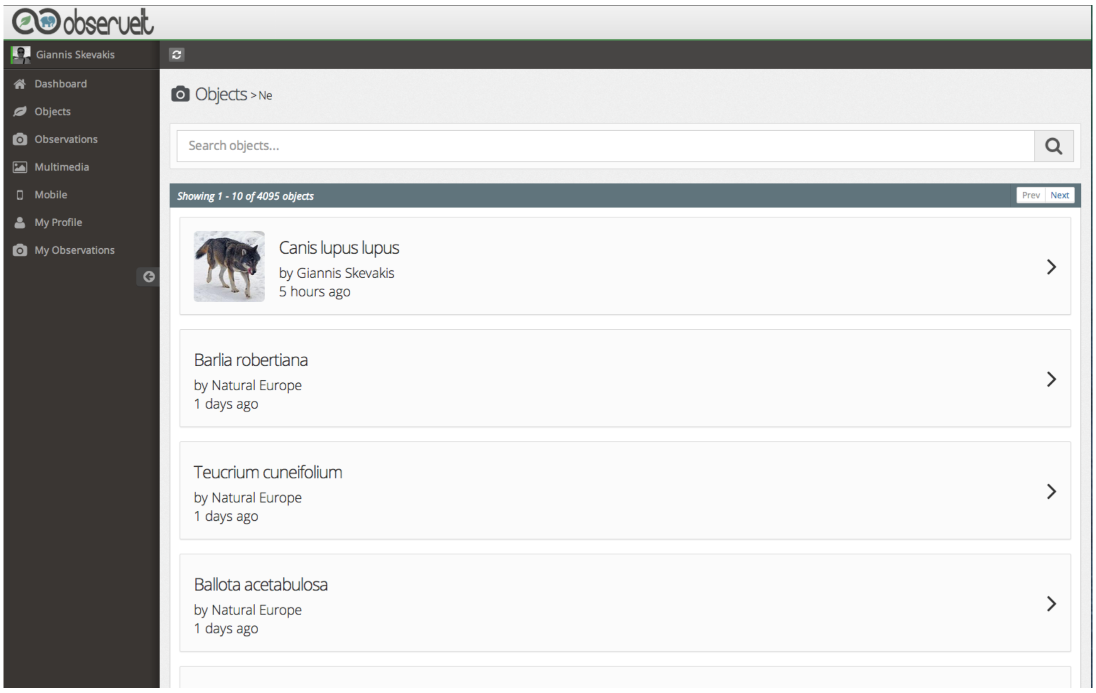
Member since	29/12/2013 (101 days)
Last seen	29/12/2013 (101 days ago)

Figure 9.20: Mobile Interface: User details

9.2.2 Desktop

This section refers to the interfaces presented to the user when using the web interface.

Browse and Review Objects



Showing 1 - 10 of 4095 objects

Canis lupus lupus
by Giannis Skevakis
5 hours ago

Barlia robertiana
by Natural Europe
1 days ago

Teucrium cuneifolium
by Natural Europe
1 days ago

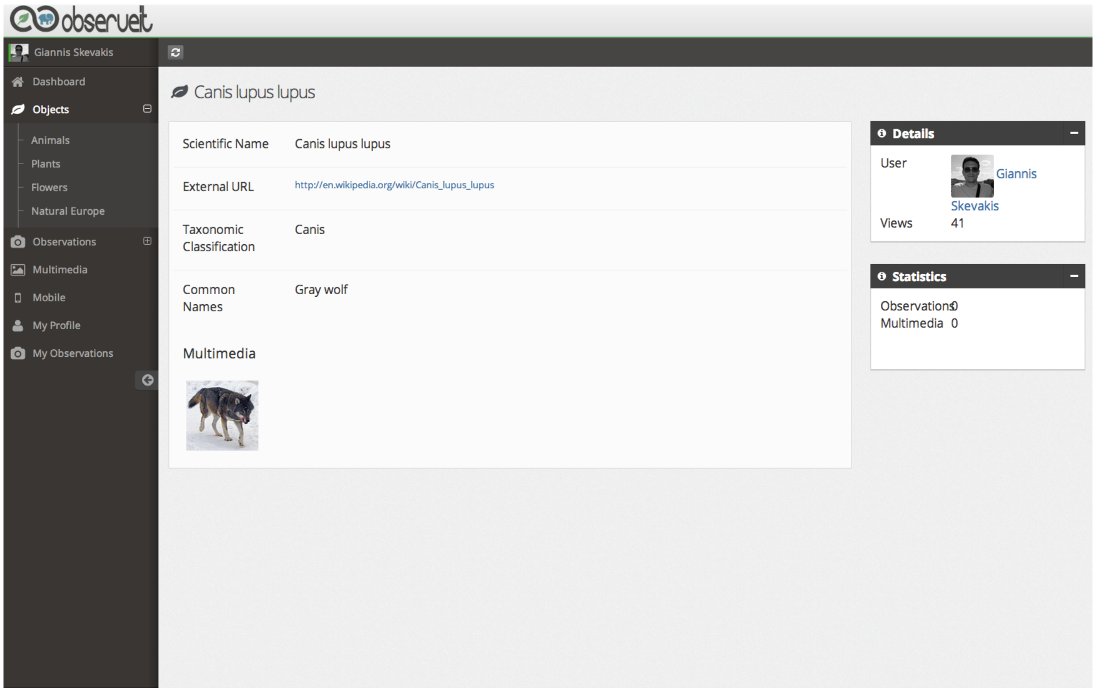
Ballota acetabulosa
by Natural Europe
1 days ago

Figure 9.21: Desktop Interface: Object list

In order for the user to browse the objects of a specific type, or search for a specific one, he has to choose the right type from the menu on the left of the interface. Then he is presented with the object list for the type, as shown in Figure 9.21.

From this screen he can use the search functionality to search for a specific object, or browse through the list. The interface contains a search bar, as well as supports paging functionality, reducing the amount of objects that need to be retrieved for each request.

Having found the object that he wants to review, he can then select it and he is shown the details of the specific object as shown in Figure 9.22. On the main part of the screen we can see the metadata and the multimedia files that follow the object. On the right side, there are the details about the user that created the object and the number of times that it has been viewed. Below we find the number of observations and objects that have been contributed about this object, which redirect to the respective screens (Figure 9.23,Figure 9.24).



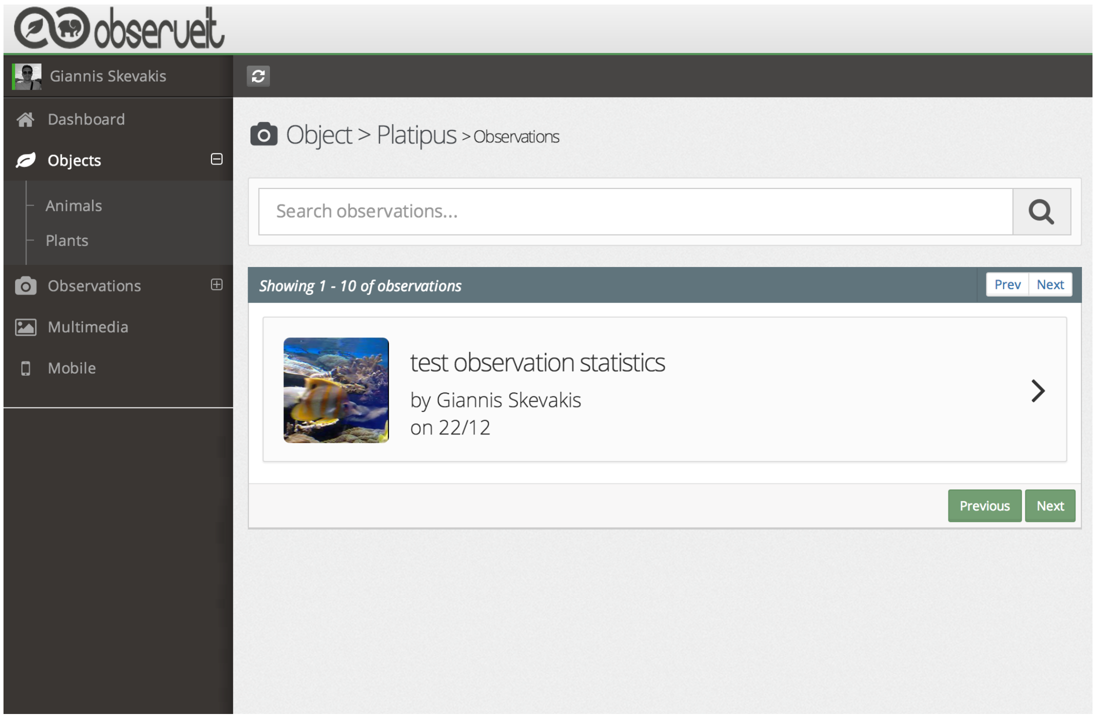
Scientific Name	Canis lupus lupus
External URL	http://en.wikipedia.org/wiki/Canis_lupus_lupus
Taxonomic Classification	Canis
Common Names	Gray wolf

Multimedia

User	Giannis Skevakis
Views	41

Observations	0
Multimedia	0

Figure 9.22: Desktop Interface: Object review



test observation statistics
by Giannis Skevakis
on 22/12

Figure 9.23: Desktop Interface: Observations concerning a specific object

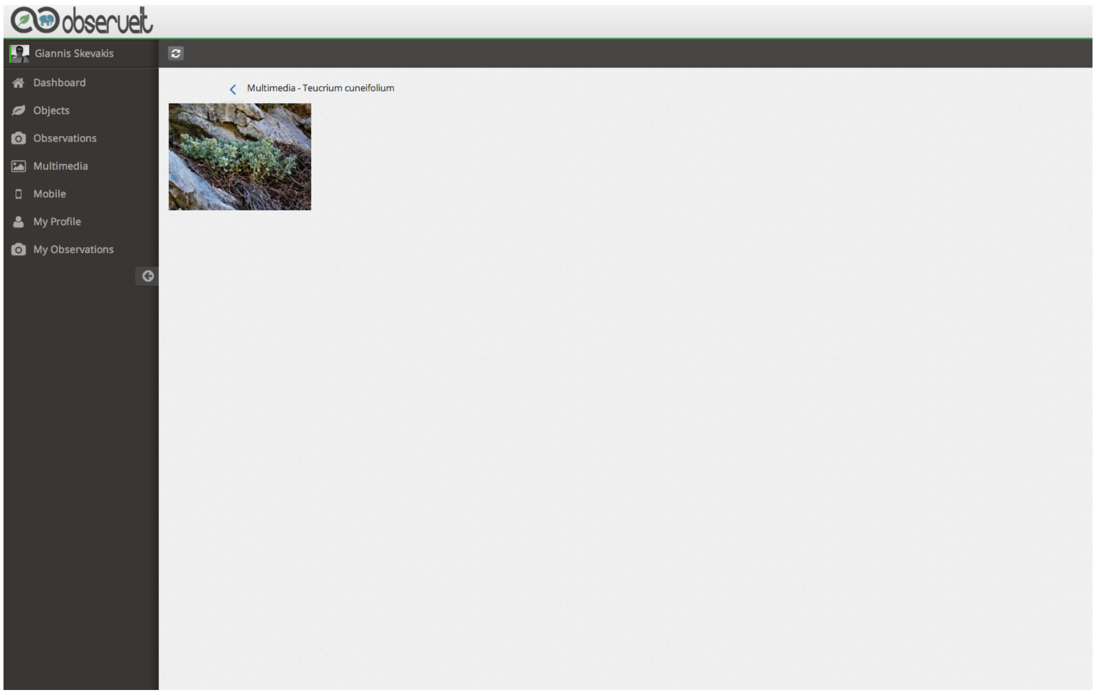
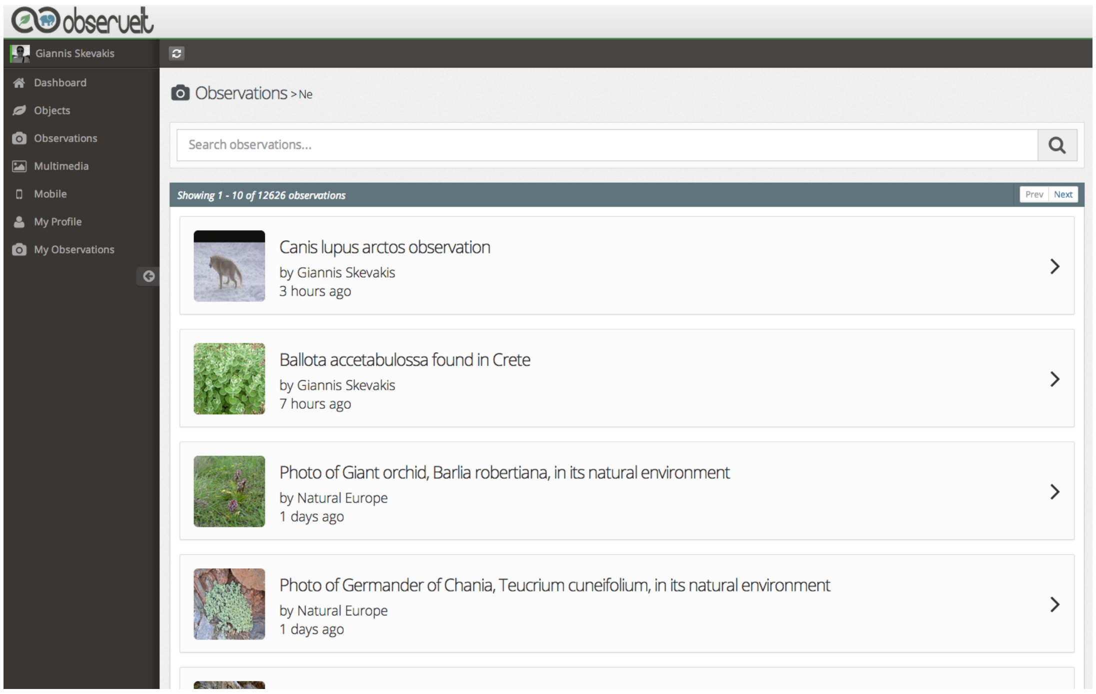


Figure 9.24: Desktop Interface: Multimedia concerning a specific object

Browse and Review Observations



Showing 1 - 10 of 12626 observations

Canis lupus arctos observation
by Giannis Skevakis
3 hours ago

Ballota accetabulossa found in Crete
by Giannis Skevakis
7 hours ago

Photo of Giant orchid, Barlia robertiana, in its natural environment
by Natural Europe
1 days ago

Photo of Germander of Chania, Teucrium cuneifolium, in its natural environment
by Natural Europe
1 days ago

Figure 9.25: Desktop Interface: Observation list

In order for the user to browse the observations of a specific type, or search for a specific one, he has to choose the right type from the menu on the left of the interface. Then he is presented with the observation list for the type, as shown in Figure 9.25.

From this screen he can use the search functionality to search for a specific observation, or browse through the list. This screen implements paging as well, as described above.

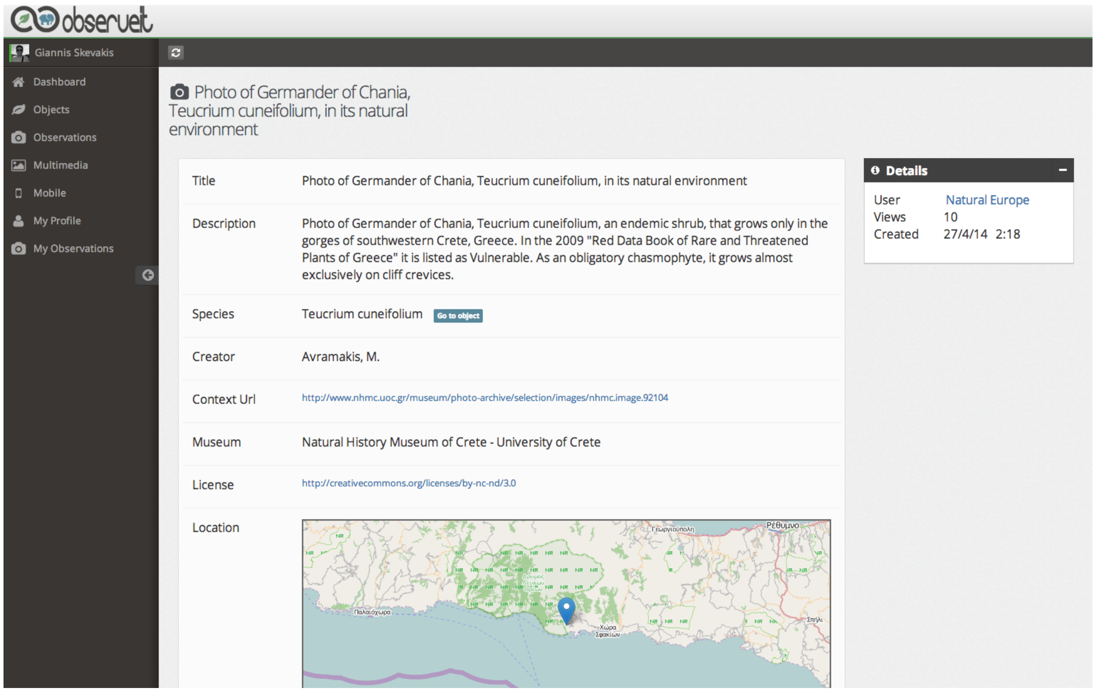


Photo of Germander of Chania, Teucrium cuneifolium, in its natural environment

Title	Photo of Germander of Chania, Teucrium cuneifolium, in its natural environment
Description	Photo of Germander of Chania, Teucrium cuneifolium, an endemic shrub, that grows only in the gorges of southwestern Crete, Greece. In the 2009 "Red Data Book of Rare and Threatened Plants of Greece" it is listed as Vulnerable. As an obligatory chasmophyte, it grows almost exclusively on cliff crevices.
Species	Teucrium cuneifolium
Creator	Avramakis, M.
Context Url	http://www.nhmc.uoc.gr/museum/photo-archive/selection/images/nhmc.image.92104
Museum	Natural History Museum of Crete - University of Crete
License	http://creativecommons.org/licenses/by-nc-nd/3.0
Location	

Details

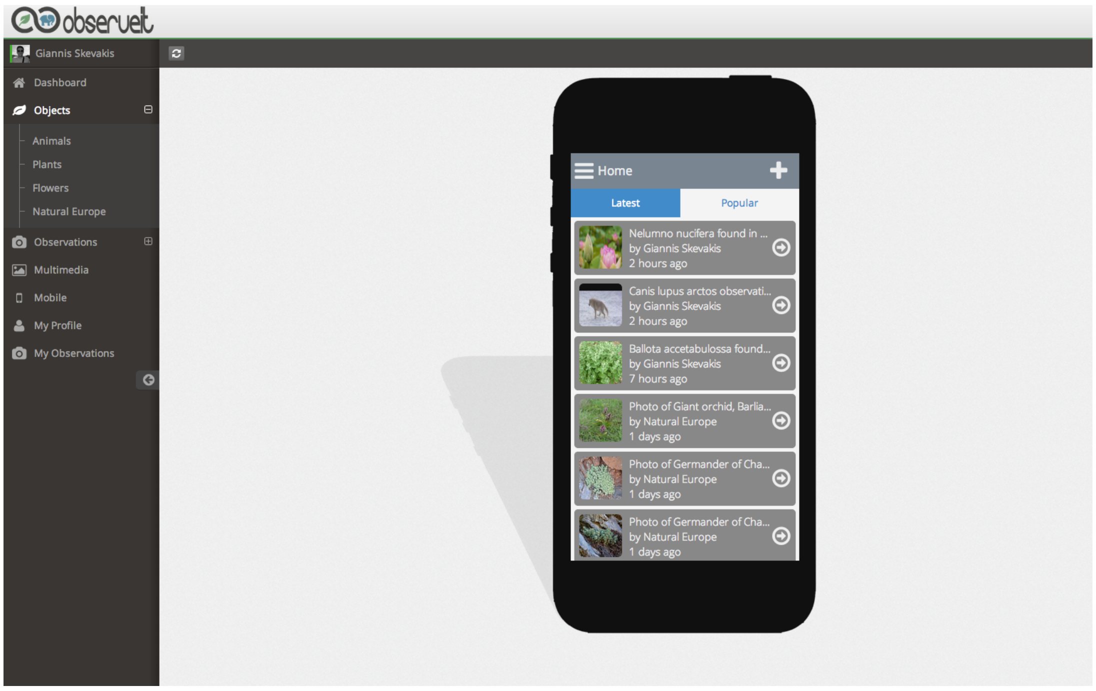
User	Natural Europe
Views	10
Created	27/4/14 2:18

Figure 9.26: Desktop Interface: Observation review

Having found the observation that he wants to review, he can then select it and he is shown the details of the specific observation as shown in Figure 9.26. On the top of the screen we can see the multimedia files that have been created with the observation. Just below there is the details about the user that created the observation and the number of times that it has been viewed. Below we find the metadata of the observation as contributed by the creator. We can notice the different metadata fields that have been filled, as well as the map showing the geographic location that the observation was made.

Review Mobile Site

The system provides users with the ability to see the mobile application from inside the desktop application, as shown in Figure 9.27.



Nelumno nucifera found in ...
by Giannis Skevakis
2 hours ago

Canis lupus arctos observati...
by Giannis Skevakis
2 hours ago

Ballota accetabulossa found...
by Giannis Skevakis
7 hours ago

Photo of Giant orchid, Barlia...
by Natural Europe
1 days ago

Photo of Germander of Cha...
by Natural Europe
1 days ago

Photo of Germander of Cha...
by Natural Europe
1 days ago

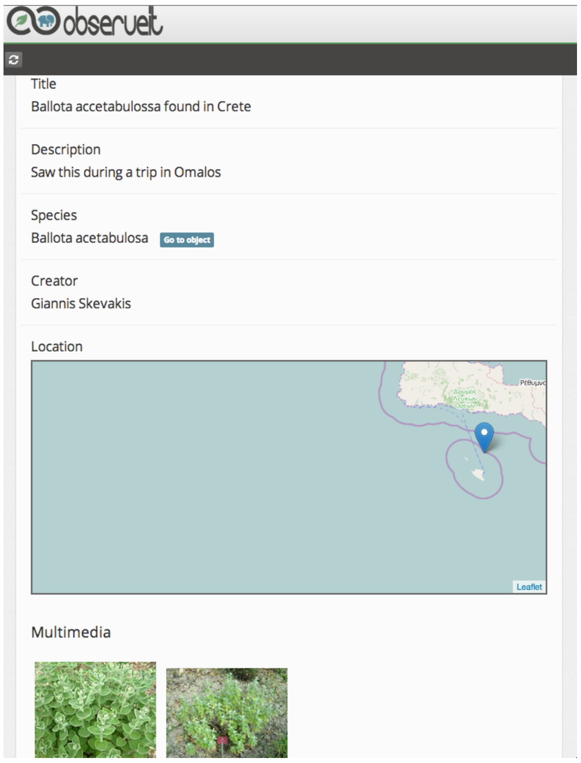
Figure 9.27: Desktop mobile review screen

Responsive Interface

The whole system and all the interfaces have been designed as responsive, meaning that the visual effects change depending on the browser and the device that the system uses. This makes a huge impact on the usability [56] of the system, especially when the user login in from a mobile browser. An example to this can be seen in Figure 9.26, which contains a screenshot of the interface presenting the metadata of an observation when viewed by a mobile browser. This is the same page as in Figure 9.28, so we can immediately notice the changes that happen in the user interface, thus making it responsive.

Summary

In this chapter we presented the methodology followed for designing the interfaces of our applications, as well as the user interfaces of the final products. We described the initial phase of prototyping the interfaces and creating the storyboards, and presented the interfaces as seen by real users of our systems.


Title
Ballota accetabulossa found in Crete

Description
Saw this during a trip in Omalos

Species
Ballota acetabulosa

Creator
Giannis Skevakis

Location

Multimedia

Figure 9.28: Desktop responsive example

Chapter 10

Migration of the Natural Europe data

This chapter presents the process and the software that we have developed in order to achieve the migration of the data collected from the Natural Europe project to our infrastructure. As has already been stated, the Natural Europe project has influenced the development of our framework. The data that it holds are in the context of biodiversity and contain metadata descriptions for both species and observations. The software that handles the transition has been build with extensibility in mind, so as to allow further handling and transition of data from other sources and other formats.
The steps that need to be taken for any migration of existing data are the following:

- Define the Application Profile based on the model of the existing data.
- Traverse the items of the source one by one.
- Translate each record to our framework's model based on the new Application Profile and possibly enrich some fields.
- Store the objects, observations and multimedia using the web services provided by our infrastructure.

10.1 Meta-Model definition

The definition of the Meta-Model based on the Natural Europe data is the first step in the transition process. The metadata about the Natural Europe cultural heritage objects follow the Natural Europe CHO Application Profile (NE CHO-AP) which has been described in brief at Section 4.1.1, so we needed to create an Application Profile with most of the elements that appear in NE CHO-AP.

In order to define the Application Profile, we need to identify the types of objects that we need to support. The fact that the NE CHO AP defines general use metadata that apply to the whole biodiversity context and the lack of distinction between plants, animals, etc., forces us to define one Application Profile for all the objects.

Table 10.1: Application Profile for the Objects based on the NE CHO AP. For simplicity we did not include the order and multivalued attributes

key	label	description	type	required
`label`	"Scientific Name"	"The scientific name of the species"	text	true
`url`	"Url"	"A URL where the user can get more information about the species"	url	false
`taxonomy`	"Taxonomic Classification"	"The taxonomy of the species"	text	false
`common_names`	"Common Names"	"The common names of the species"	text	false

Table 10.2: Application Profile for the Observations based on the NE CHO AP. For simplicity we did not include the order and multivalued attributes

key	label	description	type	required
`label`	"Title"	"The title of the observation"	text	true
`description`	"Description"	"A small description about the observation"	text	false
`creator`	"Creator"	"The creator of the observation"	text	false
`url`	"Context URL"	"A URL where the user can get more information about the observation"	url	false
`museum`	"Museum"	"The name of the museum that created the observation"	text	true
`license`	"License"	"A URL defining the license of the observation"	url	true
`location`	"Location"	"The gps coordinates of the observation"	location	true

10.2 Architecture

The architecture of the system that has been developed to support the transition of the data is presented in Figure 10.1. It is comprised of four different modules that were developed with the purpose of facilitating the migration of the data of Natural Europe. The process as well as the components are described below in detail and can be extended to support transition of data in other formats as well.

The *Access* module is responsible for accessing and retrieving the initial metadata records from the Natural Europe Cultural Environment (see Section 4.1) installations. The NE Service Client that it contains is able to communicate with the Access Services on the NECE nodes and retrieve the metadata for all the cultural heritage objects. Each of the object that it is received is passed to

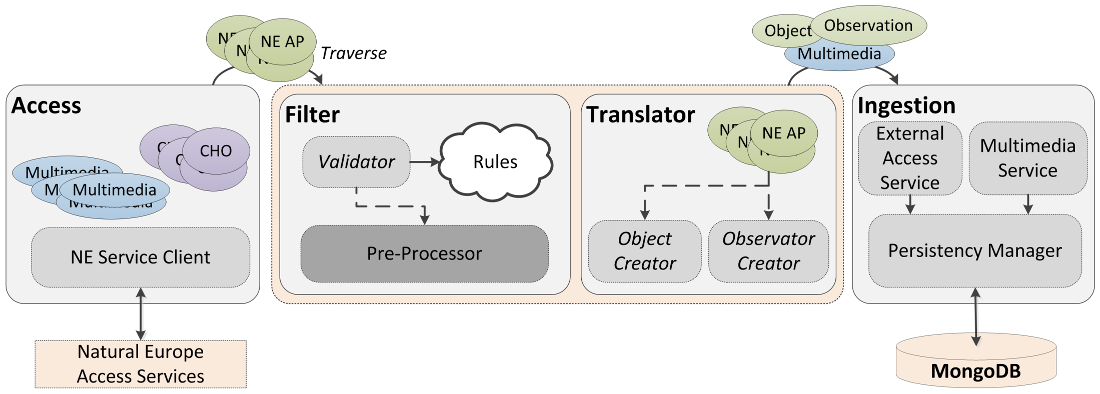


Figure 10.1: Overview of the Natural Europe Data migration process.

the Filter module for further processing. Additionally, the multimedia files that follow the objects are kept until they are later requested.

The *Filter* module is responsible for filtering all the records and choosing which of them will be eventually ingested by our systems. It contains a Validator component and a set of rules that define which records are suitable. The data that does not pass the validation phase is discarded, while the rest of the data is passed through a Pre-Processor and further moved to the Translator module.

The *Translator* module handles the conversion of the data from the NE CHO AP format to our model. This includes the separation of the metadata records to Objects and Observations, a procedure that is undertaken by two distinct components. Furthermore, the module holds an enrichment component which The final metadata are passed to the Ingestion module, along with their multimedia.

The *Ingestion* module is the final step in the process of migrating the data, and is responsible for persisting the data to our infrastructure. It connects to the External Access Services as well as the Multimedia Services of our system and passed the data. These modules are connected with the Persistency Manager which stores the final data to the database.

10.3 Results

We have used the previously defined methodology and system in all six instances of the NECE. The data from these instances has created over 2000 new objects and 300 observations. The detailed numbers of this migration can be seen in Table 10.3.

Table 10.3: The number of CHOs annotated by each NHM using MMAT.

Natural History Museums (NHMs)	CHOs	Objects	Observations	Multimedia
Natural History Museum of Crete (NHMC)	4,010	1,204	4,010	3,853
National Museum of Natural History of Lisbon (MNHNL)	2,686	1,315	2,684	1,813
Jura-Museum Eichstätt (JME)	1,658	426	1,658	1,352
Arctic Center (AC)	480	9	480	479
Hungarian Natural History Museum (HNHM)	4,244	1,315	4,243	2,361
Estonian Museum of Natural History (TNHM)	1,972	511	1,972	1,814
TOTAL	**15,050**	**4,780**	**15,047**	**11,672**

Summary

We have presented the process that we followed and the software that we have developed in order to achieve the transition of the data collected from the Natural Europe project to our infrastructure. The methodology can be extended so as to allow further handling and transition of data from other sources and other formats.

Chapter 11

Conclusion & Future Work

We presented the design and implementation of a framework for the management of biodiversity observations captured by users roaming in the nature. The main objective of this framework is to alleviate the need for experts capturing biodiversity information, and propagate the collection of information to simple users wandering in the nature. The collection of the observational data is performed using mobile devices that most of the people have available with them, like mobile phones and tablet devices. Additionally, we describes the model that we have defined, allowing the personalization of the metadata that follow the observations. This provides our framework with the freedom and extensibility needed so as to be implemented for various domains other than biodiversity, e.g. biodiversity observations about species, observations about natural disasters like earthquakes, observations about car accidents.

Our future work will include the following:

- Enable the use of social media for collaborative species identification and occurrence by providing the appropriate methodologies and tools [29]
- Implement procedures that will help users identify the species better and faster. This can use the metadata provided about the objects of the application to create questions to the user in order to help him enrich his knowledge and reach the correct object that he is observing faster.
- Capture the movement of the users when creating observations. This will allow the creation of the paths that the user followed during his trip and the playback for later user.
- Enrich attribute types with more attributes, and create complex attributes that contain more

than one simple attributes. For example, create attributes that contain the height in multiple metrics with the user defining the unit that he uses using a dropdown list that follows the value.

- Include validation rules in the Application Profile, which check the validity of the metadata provided.
- Connect the systems to Natural Europe and other systems, allowing the exchange of the observational data.
- Create visualization interfaces with faceted search functionality, enabling users to filter the information that they search for.

Bibliography

[1] Backbone.js. `http://backbonejs.org/`.

[2] BSON. `http://bsonspec.org`.

[3] Elasticsearch. `http://www.elasticsearch.org/`.

[4] Europeana Semantic Elements Specification V.3.4.1. `http://pro.europeana.eu/documents/900548/dc80802e-6efb-4127-a98e-c27c95396d57`.

[5] Global Biodiversity Information Facility. `http://www.gbif.org/`.

[6] jQuery. `http://jquery.com/`.

[7] MongoDB. `http://www.mongodb.org/`.

[8] Redis. `http://redis.io/`.

[9] Spring. `http://spring.io/`.

[10] The Natural Europe Project. `http://www.natural-europe.eu/`.

[11] Xuggler. `http://www.xuggle.com/xuggler/`.

[12] ISO 14721:2003 Open Archival Information System (OAIS) Reference Model. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=24683`, 2003.

[13] V. Abramova and J. Bernardino. NoSQL Databases: MongoDB vs Cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 14–22, New York, NY, USA, 2013. ACM.

[14] G. Alonso. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[15] D. Alur, D. Malks, and J. Crupi. *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall, 2 edition, 2003.

[16] W. Berendsohn, M. Döring, M. Gebhardt, and A. Güntsch. BioCase - A Biological Collection Access Service for Europe. Technical report, 2002.

[17] W. G. Berendsohn. ABCD Schema - Task Group on Access to Biological Collection Data. Technical report, sep 2007.

[18] T. Berners-Lee and D. Connolly. RFC 1866 – Hypertext Markup Language – 2.0. `http://www.faqs.org/rfcs/rfc1630.html`, November 1995.

[19] R. Bonney, C. B. Cooper, J. Dickinson, S. Kelling, T. Phillips, K. V. Rosenberg, and J. Shirk. Citizen science: a developing tool for expanding science knowledge and scientific literacy. *BioScience*, 59(11):977–984, 2009.

[20] D. C. Brabham. Crowdsourcing as a Model for Problem Solving: An Introduction and Cases. *Convergence: The International Journal of Research into New Media Technologies*, 14(1):75–90, 2008.

[21] D. C. Brabham. Moving the crowd at iStockphoto: The composition of the crowd and motivations for participation in a crowdsourcing application. *First Monday*, 13(6), 2008.

[22] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[23] K. Chodorow and M. Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010.

[24] J. Chone. Asynchronous JavaScript Programming. The Power Of $.Deferred for HTML5 Application - HTML5 Rocks. 2012.

[25] S. Coast. How OpenStreetMap Is Changing the World. In K. Tanaka, P. Fröhlich, and K.-S. Kim, editors, *W2GIS*, volume 6574 of *Lecture Notes in Computer Science*, page 4. Springer, 2011.

[26] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2001.

[27] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. 2004.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[29] D.-P. Deng, T.-R. Chuang, K.-T. Shao, G.-S. Mai, T.-E. Lin, R. Lemmens, C.-H. Hsu, H.-H. Lin, and M.-J. Kraak. Using Social Media for Collaborative Species Identification and Occurrence: Issues, Methods, and Tools. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Crowdsourced and Volunteered Geographic Information*, GEOCROWD '12, pages 22–29, New York, NY, USA, 2012. ACM.

[30] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, 1992.

[31] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[32] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[33] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616 - Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html, 1999. [Online; accessed 25-July-2012].

[34] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California, 2000.

[35] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[36] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, erste auflage edition, 2003.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[38] J. J. Garrett. Ajax: A New Approach to Web Applications. 2005.

[39] O. Gierke. *Spring Data JPA - Reference Documentation*, 2012.

[40] R. Hecht and S. Jablonski. NoSQL Evaluation: A Use Case Oriented Survey. In *Proceedings of the 2011 International Conference on Cloud and Service Computing*, CSC '11, pages 336–341, Washington, DC, USA, 2011. IEEE Computer Society.

[41] H. Heitkötter, T. A. Majchrzak, B. Ruland, and T. Weber. Evaluating Frameworks for Creating Mobile Web Apps. In K.-H. Krempels and A. Stocker, editors, *WEBIST*, pages 209–221. SciTePress, 2013.

[42] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrve. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation, April 2004.

[43] J. Howe. The Rise of Crowdsourcing. *Wired Magazine*, 14(6), 06 2006.

[44] J. Howe. *Crowdsourcing: Why the Power of the Crowd is Driving the Future of Business*. Crown Business, New York, 2008.

[45] S. Kelling, J. Gerbracht, D. Fink, C. Lagoze, W.-K. Wong, J. Yu, T. Damoulas, and C. P. Gomes. eBird: A Human/Computer Learning Network for Biodiversity Conservation and Research. In M. P. J. Fromherz and H. Muñoz-Avila, editors, *IAAI*. AAAI, 2012.

[46] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, Feb. 2010.

[47] X. Li, B. Dong, L. Xiao, L. Ruan, and D. Liu. HCCache: A Hybrid Client-Side Cache Management Scheme for I/O-intensive Workloads in Network-Based File Systems. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, pages 467–473, Dec 2012.

[48] K. Makris, G. Skevakis, V. Kalokyri, P. Arapi, and S. Christodoulakis. Metadata Management and Interoperability Support for Natural History Museums. In *Proceedings of the 17th International Conference on Theory and Practice of Digital Libraries*, volume 8092 of *TPDL '13*, pages 120–131. Springer, 2013.

[49] K. Makris, G. Skevakis, V. Kalokyri, P. Arapi, S. Christodoulakis, J. Stoitsis, N. Manolis, and S. L. Rojas. Federating Natural History Museums in Natural Europe. In *Proceedings of the 7th Metadata and Semantics Research Conference*, volume 390 of *MTSR '13*, pages 361–372. Springer, November 2013.

[50] K. Makris, G. Skevakis, V. Kalokyri, N. Gioldasis, F. G. Kazasis, and S. Christodoulakis. Bringing Environmental Culture Content into the Europeana.eu Portal: The Natural Europe

Digital Libraries Federation Infrastructure. In *Proceedings of the 5th Metadata and Semantics Research Conference*, volume 240 of *MTSR '11*, pages 400–411. Springer, 2011.

[51] B. McLaughlin. What is Node.js? *O'Reilly Radar*, 2011.

[52] A. Miles and S. Bechhofer. SKOS Simple Knowledge Organization System Reference. `http://www.w3.org/TR/skos-reference/`, 2009.

[53] V. Mirgorod. *Backbone.js Cookbook*. Packt Publishing, Aug. 2013.

[54] M. D. Network. Introduction to Object-Oriented JavaScript.

[55] M. W. Newman and J. A. Landay. Sitemaps, Storyboards, and Specifications: A Sketch of Web Site Design Practice. In *Symposium on Designing Interactive Systems*, pages 263–274, 2000.

[56] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, San Francisco, Calif., 1994.

[57] A. Osmani. *Developing Backbone.js Applications*. O'Reilly Media, May 2013.

[58] Z. Parker, S. Poe, and S. V. Vrbsky. Comparing NoSQL MongoDB to an SQL DB. In *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[59] T. J. Parr. Enforcing Strict Model-view Separation in Template Engines. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA, 2004. ACM.

[60] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, Proceedings of the 17th international conference on World Wide Web, pages 805–814, New York, 2008. ACM.

[61] J. Pokorny. NoSQL Databases: A Step to Database Scalability in Web Environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[62] T. Reenskaug. Models - Views - Controllers. Technical report, Technical Note, Xerox Parc, 1979.

[63] T. Reenskaug. The Model-View-Controller (MVC) Its Past and Present, 2003.

[64] P. Ritchie. The Security Risks of AJAX/Web 2.0 Applications. *Network Security*, 2007(3):4–8, March 2007.

[65] G. Skevakis, K. Makris, V. Kalokyri, P. Arapi, and S. Christodoulakis. Metadata management, interoperability and Linked Data publishing support for Natural History Museums. *International Journal on Digital Libraries*, pages 1–14.

[66] J. F. Smart. *Jenkins - The Definitive Guide: Continuos Integration for the Masses: also Covers Hudson.* O'Reilly, 2011.

[67] M. Stonebraker. SQL databases v. NoSQL databases. *Commun. ACM*, 53(4):10–11, 2010.

[68] M. Tatsubori and T. Suzumura. HTML Templates That Fly: A Template Engine Approach to Automated Offloading from Server to Client. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 951–960, New York, NY, USA, 2009. ACM.

[69] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[70] C. Tsinaraki, G. Skevakis, I. Trochatou, and S. Christodoulakis. MoM-NOCS: Management of Mobile Multimedia Nature Observations Using Crowd Sourcing. In *Proceedings of International Conference on Advances in Mobile Computing &#38; Multimedia*, MoMM '13, pages 395:395–395:404, New York, NY, USA, 2013. ACM.

[71] A. van Kesteren, J. Aubourg, J. Song, and H. R. M. Steen. XMLHttpRequest Level 1. World Wide Web Consortium, Working Draft WD-XMLHttpRequest2-20080930, January 2014.

[72] W. Wei, T. Enmin, and F. bing. A Data Distribution Platform Based on Event-Driven Mechanism. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 1395–1399, Dec 2011.

[73] A. Wiggins. eBirding: Technology Adoption and the Transformation of Leisure into Science. In *Proceedings of the 2011 iConference*, iConference '11, pages 798–799, New York, NY, USA, 2011. ACM.

[74] C. Wood, B. Sullivan, M. Iliff, D. Fink, and S. Kelling. eBird: engaging birders in science and conservation. *PLoS Biol.*, 9(12):e1001220, Dec. 2011.

Appendix A

Model Implementation Samples

This appendix provides samples of the implementation of the model, as persisted in the database, transfered through the services and used by the components. The database contains 9 main collections: (*a*) *"users"* collection holds the details for the users of the system who have an account, (*b*) *"object"* collection holds all the objects, (*c*) *"observations"* collection holds all the observations, (*d*) *"profiles"* collection holds the application profiles used by the system, (*e*) *"multimedia"* collection holds the metadata concerning the multimedia objects, (*f*) *"thumbnails"* collection contains the thumbnails generated by the system, (*g*) *"statistics"* collection holds the the number of views concerning the objects and the observations.

Apart from these, there are some additional collections holding pieces of data, like the collections that contain the binary data for the multimedia objects and various collections that hold calculated data, reducing the number of queries. In the following sections we will present sample data for the most important collections.

Users

```
{
    "_id" : ObjectId("52a0bc5ee4b0cb13e882115d"),
    "_class" : "com.skevakis.observeit.core.model.User",
    "firstName" : "Giannis",
    "lastName" : "Skevakis",
    "email" : "gskevakis@gmail.com",
    "password" : "$2a$10$ySKcjRa0e8tySpsgkYZZ8",
    "joinedOn" : ISODate("2013-12-28T23:03:19.841Z"),
    "role" : "ADMIN"
}
```

Objects

```
{
    "_class" : "com.skevakis.observeit.core.model.ObservationObject",
    "_id" : ObjectId("52b4a398e4b041056bbc442b"),
    "attributes" : {
        "description" : "dnk",
        "number" : NumberLong(4)
    },
    "creationDate" : ISODate("2013-12-20T20:07:52.150Z"),
    "label" : "canis lupus",
    "multimedia" : [
        ObjectId("52b4a38ee4b041056bbc4428")
    ],
    "type" : "plants",
    "userId" : ObjectId("52b98c0ee4b013c9e11dccb5")
}
```

Observations

```
{
    "_class" : "com.skevakis.observeit.core.model.Observation",
    "_id" : ObjectId("52b4ad06e4b039de5f0f0f4a"),
    "attributes" : {
        "description" : "2",
        "number" : NumberLong(22),
        "object" : "52b4a398e4b041056bbc442b"
    },
    "creationDate" : ISODate("2013-12-20T20:48:06.966Z"),
    "label" : "observation of canis lupus",
    "location" : [
        37.983715,
        23.72931
    ],
    "type" : "plants",
    "userId" : ObjectId("52b98c0ee4b013c9e11dccb5")
}
```

Profiles

```
{
    "_id" : ObjectId("52b4a352a6d2221cfd619b44"),
    "id" : "52a2d730e4b070e8e47573e3",
    "type" : "animals",
    "objectAttributes" : [
        {
            "key" : "label",
            "label" : "Title",
            "description" : "observation label",
            "type" : "text",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "list",
            "label" : "List",
            "description" : "observation list",
            "type" : "list",
            "multiplicity" : false,
            "required" : false,
            "options" : {
                "listOptions" : [
                    {
                        "order" : 0,
                        "label" : "First Option",
                        "value" : "first"
                    },
                    {
                        "order" : 1,
                        "label" : "Second Option",
                        "value" : "second"
                    },
                    {
                        "order" : 2,
                        "label" : "Third Option",
                        "value" : "third"
                    },
                    {
                        "order" : 3,
                        "label" : "Fourth Option",
                        "value" : "fourth"
                    }
                ]
            }
        },
        {
            "key" : "date",
            "label" : "Date",
            "description" : "observation date",
            "type" : "date",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "description",
            "label" : "Description",
            "description" : "observation description",
            "type" : "text",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "number",
            "label" : "Number",
            "description" : "observation number",
            "type" : "number",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "location",
            "label" : "Location",
            "description" : "description about location",
            "type" : "location",
            "multiplicity" : false,
            "required" : false
        }
    ],
    "observationAttributes" : [
        {
            "key" : "label",
            "label" : "Title",
            "description" : "observation title",
            "type" : "text",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "object",
            "label" : "Animal",
            "description" : "animal field description",
            "type" : "object",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "list",
            "label" : "List",
            "description" : "observation list",
            "type" : "list",
            "multiplicity" : false,
            "required" : false,
            "options" : {
                "listOptions" : [
                    {
                        "order" : 0,
                        "label" : "First Option",
                        "value" : "first"
                    },
                    {
                        "order" : 0,
                        "label" : "Second Option",
                        "value" : "second"
                    },
                    {
                        "order" : 0,
                        "label" : "Third Option",
                        "value" : "third"
                    },
                    {
                        "order" : 0,
                        "label" : "Fourth Option",
                        "value" : "fourth"
                    }
                ]
            }
        },
        {
            "key" : "date",
            "label" : "Date",
            "description" : "observation date",
            "type" : "date",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "username",
            "label" : "Your name",
            "description" : "observation description",
            "type" : "text",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "number",
            "label" : "Number",
            "description" : "observation number",
            "type" : "number",
            "multiplicity" : false,
            "required" : false
        },
        {
            "key" : "location",
            "label" : "Location",
            "description" : "description about location",
            "type" : "location",
            "multiplicity" : false,
            "required" : false
        }
    ]
}
```

Multimedia

```
{
    "_class" : "com.skevakis.observeit.core.model.Video",
    "_id" : ObjectId("52b55357e4b0dd7e997b5bc3"),
    "duration" : NumberLong(28),
    "fileId" : ObjectId("52b55356e4b0dd7e997b5bb7"),
    "height" : 360,
    "objectId" : ObjectId("52b4a398e4b041056bbc442b"),
    "type" : "video",
    "width" : 202
}
```

Statistics

```
{
    "_class" : "com.skevakis.observeit.core.model.Statistics",
    "_id" : ObjectId("52b70e68e4b05263c23412d7"),
    "views" : 29
}
```

Appendix B

Restful Web Services

This appendix describes a list of the web services that are provided by the system. All the services are based on Representational state transfer (REST) [34]. The data format supported is JSON, although the services can be easily extended to support other formats as well. The services have been categorized in six categories:

- *"User"*: describes services that are relevant to the user management.
- *"Object"*: contains services that handle the objects of the system.
- *"Observation"*: contains services that handle the observations.
- *"Application Profile"*: describes services concerning the application profiles.
- *"Multimedia"*: describes the services about the the multimedia.
- *"Statistics"*: contains services that expose the statistics of the objects and the observations.

The services for each category are documented below.

B.1 User

Table B.1: RESTful Service: Get all users.

URL	/user
Method	GET
Description	Returns the user list.
Example	

```
GET /user


[
  {
    "id":"52b98c0ee4b013c9e11dccb5",
    "firstName":"Giannis",
    "lastName":"Skevakis",
    "email":"giannis@gmail.com",
    "joinedOn":1388271799841,
    "lastSeen":1389202911470,
    "imageId":"52cd8e0d74d245d150395d4c",
    "role":"ADMIN"
  },
  {
    "id":"52bf58b774d27e9323764752",
    "firstName":"Giannis",
    "lastName":"Skevakis",
    "email":"giannis.86@gmail.com",
    "joinedOn":1388271799841,
    "role":"USER"
  }
]
```

Table B.2: RESTful Service: Get user's details.

URL	/user/{id}
Method	GET
Parameters	*id*: the user's id
Description	Returns the details of the user.
Example	

```
GET /user/52b98c0ee4b013c9e11dccb5


{
  "id":"52b98c0ee4b013c9e11dccb5",
  "firstName":"Giannis",
  "lastName":"Skevakis",
  "email":"giannis@gmail.com",
  "joinedOn":1388271799841,
  "lastSeen":1397168064393,
  "imageId":"52cd8e0d74d245d150395d4c",
  "role":"ADMIN",
  "statistics":{
    "id":"52b98c0ee4b013c9e11dccb5",
    "observations":23,
    "multimedia":12
  }
}
```

Table B.3: RESTful Service: Update user's details.

URL	/user/update/{id}
Method	POST
Parameters	*firstName*: the new user's first name *lastName*: the new last name *email*: the new email *imageId*: the new user's image
Description	Updates the user's details and returns the new data.
Example	

```
POST /user/52b98c0ee4b013c9e11dccb5


{
  "id":"52b98c0ee4b013c9e11dccb5",
  "firstName":"Giannis",
  "lastName":"Skevakis",
  "email":"giannis@gmail.com",
  "joinedOn":1388271799841,
  "lastSeen":1397168064393,
  "imageId":"52cd8e0d74d245d150395d4c",
  "role":"ADMIN",
  "statistics":{
    "id":"52b98c0ee4b013c9e11dccb5",
    "observations":23,
    "multimedia":12
  }
}
```

Table B.4: RESTful Service: Delete user.

URL	/user/{id}
Method	DELETE
Parameters	*id*: the user's id
Description	Deletes the user.
Example	

```
DELETE /user/52b98c0ee4b013c9e11dccb5
```

B.2 Object

Table B.5: RESTful Service: Create object.

URL	/object
Method	POST
Description	Create a new object and return it.
Example	

```
POST /object

{
  "id":"52b70bc6e4b032a73a7c1aa5",
  "type":"animals",
  "createdOn":1387727814194,
  "label":"platipus",
  "user":"52b98c0ee4b013c9e11dccb5",
  "multimedia":[
    "52b70bc0e4b032a73a7c1aa2"
  ]
}
```

Table B.6: RESTful Service: Get objects.

URL	/object
Method	GET
Parameters	*size*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the objects in the system, sorted by creation date and with paging.

Example

```
GET /object?size=10&page=0


{
  "content":[
    {
      "id":"52b70bc6e4b032a73a7c1aa5",
      "type":"animals",
      "createdOn":1387727814194,
      "label":"platipus",
      "user":"52b98c0ee4b013c9e11dccb5",
      "multimedia":[
        "52b70bc0e4b032a73a7c1aa2"
      ]
    },
    {
      "id":"52b58fc7e4b0d89d2f5327d8",
      "type":"animals",
      "createdOn":1387630535927,
      "label":"test object animal",
      "user":"52b98c0ee4b013c9e11dccb5"
    },
    ...
  ],
  "numberOfElements":5,
  "lastPage":true,
  "firstPage":true,
  "totalPages":1,
  "sort":[
    {
      "direction":"DESC",
      "property":"creationDate",
      "ignoreCase":false,
      "ascending":false
    }
  ],
  "totalElements":5,
  "size":10,
  "number":0
}
```

Table B.7: RESTful Service: Get object.

URL	/object/{id}
Method	GET
Parameters	*id*: the object's id
Description	Returns the object.
Example	

```
GET /object/52b70bc6e4b032a73a7c1aa5


{
  "id":"52b70bc6e4b032a73a7c1aa5",
  "type":"animals",
  "createdOn":1387727814194,
  "label":"platipus",
  "creator":{
    "id":"52b98c0ee4b013c9e11dccb5",
    "firstName":"Giannis",
    "lastName":"Skevakis",
    "email":"giannis@gmail.com",
    "joinedOn":1388271799841,
    "lastSeen":1389202911470,
    "imageId":"52cd8e0d74d245d150395d4c",
    "role":"ADMIN",
    "statistics":{
      "id":"52b98c0ee4b013c9e11dccb5",
      "observations":24,
      "multimedia":12
    }
  },
  "multimedia":[
    {
      "type":"image",
      "id":"52b70bc0e4b032a73a7c1aa2",
      "fileId":"52b70bc0e4b032a73a7c1a9f",
      "height":1200,
      "width":1920
    }
  ],
  "statistics":{
    "id":"52b70bc6e4b032a73a7c1aa5",
    "views":89,
    "likes":0,
    "observations":1,
    "multimedia":1
  }
}
```

Table B.8: RESTful Service: Get objects of a specific type.

URL	/object/type/{type}
Method	GET
Parameters	*type*: the type of objects from the application profiles *count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the objects of the type, sorted by creation date and with paging.

Example

```
GET /object/type/animals?size=10&page=0
```

```
{
  "content":[
    {
      "id":"52b70bc6e4b032a73a7c1aa5",
      "type":"animals",
      "createdOn":1387727814194,
      "label":"platipus",
      "user":"52b98c0ee4b013c9e11dccb5",
      "multimedia":[
        "52b70bc0e4b032a73a7c1aa2"
      ]
    },
    {
      "id":"52b58fc7e4b0d89d2f5327d8",
      "type":"animals",
      "createdOn":1387630535927,
      "label":"test object animal",
      "user":"52b98c0ee4b013c9e11dccb5"
    },
    ...
  ],
  "numberOfElements":5,
  "lastPage":true,
  "firstPage":true,
  "totalPages":1,
  "sort":[
    {
      "direction":"DESC",
      "property":"creationDate",
      "ignoreCase":false,
      "ascending":false
    }
  ],
  "totalElements":5,
  "size":10,
  "number":0
}
```

Table B.9: RESTful Service: Get latest objects.

URL	/object/latest
Method	GET
Parameters	*count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the latest created objects.
Example	

```
GET /object/latest?count=10&page=0
```

```
{
  "id":"52b70bc6e4b032a73a7c1aa5",
  "type":"animals",
  "createdOn":1387727814194,
  "label":"platipus",
  "creator":{
    "id":"52b98c0ee4b013c9e11dccb5",
    "firstName":"Giannis",
    "lastName":"Skevakis",
    "email":"giannis@gmail.com",
    "joinedOn":1388271799841,
    "lastSeen":1389202911470,
    "imageId":"52cd8e0d74d245d150395d4c",
    "role":"ADMIN",
    "statistics":{
      "id":"52b98c0ee4b013c9e11dccb5",
      "observations":24,
      "multimedia":12
    }
  },
  "multimedia":[
    {
      "type":"image",
      "id":"52b70bc0e4b032a73a7c1aa2",
      "fileId":"52b70bc0e4b032a73a7c1a9f",
      "height":1200,
      "width":1920
    }
  ],
  "statistics":{
    "id":"52b70bc6e4b032a73a7c1aa5",
    "views":89,
    "likes":0,
    "observations":1,
    "multimedia":1
  }
}
```

Table B.10: RESTful Service: Search objects.

URL	/object/search/{type}
Method	GET
Parameters	*type*: the type of objects from the application profiles *query*: the search query *count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the objects that match the query, sorted by creation date and with paging.
Example	

```
GET /object/search/animals?query=test&size=10&page=0


{
  "content":[
    {
      "id":"52b70bc6e4b032a73a7c1aa5",
      "type":"animals",
      "createdOn":1387727814194,
      "label":"platipus",
      "user":"52b98c0ee4b013c9e11dccb5",
      "multimedia":[
        "52b70bc0e4b032a73a7c1aa2"
      ]
    },
    {
      "id":"52b58fc7e4b0d89d2f5327d8",
      "type":"animals",
      "createdOn":1387630535927,
      "label":"test object animal",
      "user":"52b98c0ee4b013c9e11dccb5"
    },
    ...
  ],
  "numberOfElements":5,
  "lastPage":true,
  "firstPage":true,
  "totalPages":1,
  "sort":[
    {
      "direction":"DESC",
      "property":"creationDate",
      "ignoreCase":false,
      "ascending":false
    }
  ],
  "totalElements":5,
  "size":10,
  "number":0
}
```

Table B.11: RESTful Service: Delete object.

URL	/object/{id}
Method	DELETE
Parameters	*id*: the object's id
Description	Deletes the object.
Example	

```
DELETE /object/52b98c0ee4b013c9e11dccb5
```

B.3 Observation

Table B.12: RESTful Service: Create observation.

URL	/observation
Method	POST
Description	Create a new observation and return it.
Example	

```
POST /observation


{
  "id":"52ca6f1e74d2423694ed75d6",
  "label":"another test",
  "creator":"Giannis Skevakis",
  "thumbnailId":"52ca6f1374d2423694ed75bd",
  "createdOn":1388998430171
}
```

Table B.13: RESTful Service: Get observation.

URL	/observation/{id}
Method	GET
Parameters	*id*: the observation's id
Description	Returns the observation.
Example	

```
GET /observation/52b70bc6e4b032a73a7c1aa5


{
  "id":"52ca6f1e74d2423694ed75d6",
  "type":"plants",
  "createdOn":1388998430171,
  "label":"another test",
  "location":{
    "longitude":23.729309999999998,
    "latitude":37.983715
  },
  "creator":{
    "id":"52bf58b774d27e9323764752",
    "firstName":"Giannis",
    "lastName":"Skevakis",
    "email":"giannis.86@gmail.com",
    "joinedOn":1388271799841,
    "role":"USER"
  },
  "creatorId":"52bf58b774d27e9323764752",
  "multimedia":[
  {
    "type":"image",
    "id":"52ca6f1374d2423694ed75c0",
    "fileId":"52ca6f1374d2423694ed75bd",
    "height":1200,
    "width":1920
  },
  {
    "type":"video",
    "id":"52ca6f1774d2423694ed75cf",
    "fileId":"52ca6f1674d2423694ed75c3",
    "height":360,
    "width":202,
    "duration":28
  }
  ],
  "statistics":{
    "id":"52ca6f1e74d2423694ed75d6",
    "views":57,
    "likes":0
  }
}
```

Table B.14: RESTful Service: Get observations.

URL	/observation
Method	GET
Parameters	*size*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the observations in the system, sorted by creation date and with paging.
Example	

```
GET /observation?size=10&page=0


{
  "content":[
    {
      "id":"52ca6f1e74d2423694ed75d6",
      "type":"plants",
      "createdOn":1388998430171,
      "label":"another test",
      "location":{
        "longitude":23.729309999999998,
        "latitude":37.983715
      },
      "creatorId":"52bf58b774d27e9323764752",
      "multimedia":[
        "52ca6f1374d2423694ed75c0",
        "52ca6f1774d2423694ed75cf",
        "52ca6f1974d2423694ed75d5"
      ]
    },
    ...
  ],
  "size":10,
  "number":0,
  "sort":[
    {
      "direction":"DESC",
      "property":"creationDate",
      "ignoreCase":false,
      "ascending":false
    }
  ],
  "totalElements":27,
  "numberOfElements":10,
  "totalPages":3,
  "firstPage":true,
  "lastPage":false
}
```

Table B.15: RESTful Service: Get object's observations.

URL	/object/{id}/observations
Method	GET
Parameters	*id*: the object's id *count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the observations about a single object, sorted by creation date and with paging.

Example

```
GET /object/52b70bc6e4b032a73a7c1aa5/observations?count=10&page=0


{
  "content":[
    {
      "id":"52b70c66e4b032a73a7c1b0f",
      "type":"animals",
      "createdOn":1387727974904,
      "label":"test observation statistics",
      "location":{
        "longitude":23.729309999999998,
        "latitude":37.983715
      },
      "creator":{
        "id":"52b98c0ee4b013c9e11dccb5",
        "firstName":"Giannis",
        "lastName":"Skevakis",
        ...
      },
      "multimedia":[
        {
          "type":"video",
          "id":"52b70c4be4b032a73a7c1b0c",
          "fileId":"52b70c4ae4b032a73a7c1b00",
          "height":360,
          "width":202,
          "duration":28
        }
      ],
      "objectId":"52b70bc6e4b032a73a7c1aa5",
      "statistics":{
        "id":"52b70c66e4b032a73a7c1b0f",
        "views":79,
        "likes":0
      }
    }
  ],
  "size":10,
  "number":0,
  "totalElements":1,
  "totalPages":1
}
```

Table B.16: RESTful Service: Get user's observations.

URL	/user/{id}/observations
Method	GET
Parameters	*id*: the user's id *count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the observations of a single user, sorted by creation date and with paging.
Example	

```
GET /user/52b70bc6e4b032a73a7c1aa5/observations?count=10&page=0

{
  "content":[
    {
      "id":"52c7b801e4b021e06f0b85f9",
      "type":"plants",
      "createdOn":1388820481886,
      "label":"test with audio",
      "location":{
        "longitude":23.729309999999998,
        "latitude":37.983715
      },
      ...
    },
    {
      "id":"52c7174ae4b0d4610943ce82",
      "type":"plants",
      "createdOn":1388779338543,
      "label":"tst",
      "location":{
        "longitude":23.729309999999998,
        "latitude":37.983715
      },
      ...
    },
    ...
  ],
  "size":10,
  "number":0,
  "totalElements":24,
  "numberOfElements":10,
  "totalPages":3,
  "firstPage":true,
  "lastPage":false
}
```

Table B.17: RESTful Service: Get observations of a single type.

URL	/observation/type/{type}
Method	GET
Parameters	*type*: the type's name *count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the observations of a single type, sorted by creation date and with paging.
Example	

```
GET /observation/type/animals?count=10&page=0
```

```
{
  "content":[
    {
      "id":"52c2dfd5e4b083bd6b95e474",
      "type":"animals",
      "createdOn":1388502997420,
      "label":"test",
      "location":{
        "longitude":23.729309999999998,
        "latitude":37.983715
      },
      ...
    },
    {
      "id":"52c27a5ee4b07b2b9ff8c6a6",
      "type":"animals",
      "createdOn":1388477022759,
      "label":null,
      ...
    },
    ...
  ],
  "size":10,
  "number":0,
  "totalElements":6,
  "numberOfElements":6,
  "totalPages":1,
  "firstPage":true,
  "lastPage":true
}
```

Table B.18: RESTful Service: Get latest observations.

URL	/observation/latest
Method	GET
Parameters	*count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the latest created observations.
Example	

```
GET /observation/latest?count=10&page=0


{
  "content":[
    {
      "id":"52ca6f1e74d2423694ed75d6",
      "type":"plants",
      "createdOn":1388998430171,
      "label":"another test",
      "location":{
      "longitude":23.729309999999998,
      "latitude":37.983715
      },
      "creator":{
      "id":"52bf58b774d27e9323764752",
      "firstName":"Giannis",
      "lastName":"Skevakis",
      "email":"giannis.86@gmail.com",
      "joinedOn":1388271799841,
      "role":"USER"
      },
      ...
    },
    ...
  ],
  "size":10,
  "number":0,
  "sort":[
    {
      "direction":"DESC",
      "property":"creationDate",
      "ignoreCase":false,
      "ascending":false
    }
  ],
  "totalElements":27,
  "numberOfElements":10,
  "totalPages":3,
  "firstPage":true,
  "lastPage":false
}
```

Table B.19: RESTful Service: Get most popular observations.

URL	/observation/popular
Method	GET
Parameters	*count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the most popular observations.
Example	

```
GET /observation/popular?count=10&page=0


{
  "content":[
    {
      "id":"52ca6f1e74d2423694ed75d6",
      "type":"plants",
      "createdOn":1388998430171,
      "label":"another test",
      "location":{
      "longitude":23.729309999999998,
      "latitude":37.983715
      },
      "creator":{
      "id":"52bf58b774d27e9323764752",
      "firstName":"Giannis",
      "lastName":"Skevakis",
      "email":"giannis.86@gmail.com",
      "joinedOn":1388271799841,
      "role":"USER"
      },
      ...
    },
    ...
  ],
  "size":10,
  "number":0,
  "sort":[
    {
      "direction":"DESC",
      "property":"creationDate",
      "ignoreCase":false,
      "ascending":false
    }
  ],
  "totalElements":27,
  "numberOfElements":10,
  "totalPages":3,
  "firstPage":true,
  "lastPage":false
}
```

Table B.20: RESTful Service: Search observations.

URL	/observation/search/{type}
Method	GET
Parameters	*type*: the type of observations from the application profiles *query*: the search query *count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the observations that match the query, sorted by creation date and with paging.
Example	

```
GET /observation/search/animals?query=test&size=10&page=0


{
  "content":[
    {
      "id":"52ca6f1e74d2423694ed75d6",
      "type":"plants",
      "createdOn":1388998430171,
      "label":"another test",
      "location":{
      "longitude":23.729309999999998,
      "latitude":37.983715
      },
      "creator":{
      "id":"52bf58b774d27e9323764752",
      "firstName":"Giannis",
      "lastName":"Skevakis",
      "email":"giannis.86@gmail.com",
      "joinedOn":1388271799841,
      "role":"USER"
      },
      ...
    },
    ...
  ],
  "size":10,
  "number":0,
  "sort":[
    {
      "direction":"DESC",
      "property":"creationDate",
      "ignoreCase":false,
      "ascending":false
    }
  ],
  "totalElements":27,
  "numberOfElements":10,
  "totalPages":3,
  "firstPage":true,
  "lastPage":false
}
```

Table B.21: RESTful Service: Delete observation.

URL	/observation/{id}
Method	DELETE
Parameters	*id*: the observation's id
Description	Deletes the observation.
Example	
`DELETE /observation/52b98c0ee4b013c9e11dccb5`	

B.4 Application Profile

Table B.22: RESTful Service: Create application profile.

URL	/profile
Method	POST
Description	Creates a new application profile and returns it.
Example	

```
POST /profile

{
  "id":"52b4a352a6d2221cfd619b44",
  "type":"animals",
  "objectAttributes":[
    {
      "key":"label",
      "label":"Title",
      "description":"observation label",
      "type":"text",
      "multiplicity":false,
      "required":false
    },
    {
      "key":"description",
      "label":"Description",
      "description":"observation description",
      "type":"text",
      "multiplicity":false,
      "required":false
    }
  ],
  "observationAttributes":[
    {
      "key":"label",
      "label":"Title",
      "description":"observation title",
      "type":"text",
      "multiplicity":false,
      "required":false
    },
    {
      "key":"object",
      "label":"Animal",
      "description":"animal field description",
      "type":"object",
      ...
    },
  ]
}
```

Table B.23: RESTful Service: Get application profiles.

URL	/profile
Method	GET
Description	Get the application profiles.
Example	

```
GET /profile


[
  {
    "id":"52b4a352a6d2221cfd619b44",
    "type":"animals",
    "objectAttributes":[
      {
        "key":"label",
        "label":"Title",
        "description":"observation label",
        "type":"text",
        "multiplicity":false,
        "required":false
      },
      ...
    ],
    "observationAttributes":[
      {
        "key":"label",
        "label":"Title",
        "description":"observation title",
        "type":"text",
        "multiplicity":false,
        "required":false
      },
      ...
    ]
  },
  {
    "id":"52b4a37aa6d2221cfd619b45",
    "type":"plants",
    "objectAttributes":[
      ...
    ],
    "observationAttributes":[
      ...
    ]
  }
]
```

Table B.24: RESTful Service: Get application profile details.

URL	/profile/{type}
Method	GET
Parameters	*type*: the profile's type
Description	Returns the details of the application profile
Example	

```
GET /profile/animals

{
  "id":"52b4a352a6d2221cfd619b44",
  "type":"animals",
  "objectAttributes":[
    {
      "key":"label",
      "label":"Title",
      "description":"observation label",
      "type":"text",
      "multiplicity":false,
      "required":false
    },
    {
      "key":"description",
      "label":"Description",
      "description":"observation description",
      "type":"text",
      "multiplicity":false,
      "required":false
    }
  ],
  "observationAttributes":[
    {
      "key":"label",
      "label":"Title",
      "description":"observation title",
      "type":"text",
      "multiplicity":false,
      "required":false
    },
    {
      "key":"object",
      "label":"Animal",
      "description":"animal field description",
      "type":"object",
      "multiplicity":false,
      "required":false
    },
  ]
}
```

Table B.25: RESTful Service: Get application compact data.

URL	/details/{type}
Method	GET
Parameters	*type*: the profile's type
Description	Returns the details of the application profile.
Example	

```
GET /details/plants


{
  "name":"plants",
  "objectCount":2,
  "observationCount":22,
  "objects":[
    {
      "id":"52b57eace4b0d89d2f53270b",
      "label":"test paparouna",
      "creator":"Giannis Skevakis",
      "thumbnailId":"52b57e81e4b0d89d2f532703",
      "createdOn":1387626156392
    },
    {
      "id":"52b4a398e4b041056bbc442b",
      "label":"canis lupus",
      "creator":"Giannis Skevakis",
      "thumbnailId":"52b4a38ee4b041056bbc4425",
      "createdOn":1387570072150
    }
  ],
  "observations":[
    {
      "id":"52ca6f1e74d2423694ed75d6",
      "label":"another test",
      "creator":"Giannis Skevakis",
      "thumbnailId":"52ca6f1374d2423694ed75bd",
      "createdOn":1388998430171
    },
    {
      "id":"52c7174ae4b0d4610943ce82",
      "label":"tst",
      "creator":"Giannis Skevakis",
      "thumbnailId":"52c71746e4b0d4610943ce7c",
      "createdOn":1388779338543
    }
  ]
}
```

Table B.26: RESTful Service: Delete application profile.

URL	/profile/{type}
Method	DELETE
Parameters	*type*: the application profile's type
Description	Deletes the application profile.
Example	
`DELETE /profile/animals`	

B.5 Multimedia

Table B.27: RESTful Service: Upload new multimedia.

URL	/observation/{image | video | audio}
Method	POST
Description	Upload new multimedia file and return it's data.
Example	

```
POST /observation/{image | video | audio}
```

Table B.28: RESTful Service: Get observation.

URL	/observation/{id}
Method	GET
Parameters	*id*: the observation's id
Description	Returns the observation.
Example	

```
GET /observation/52b70bc6e4b032a73a7c1aa5
```

```
{
  "id":"52ca6f1e74d2423694ed75d6",
  "type":"plants",
  "createdOn":1388998430171,
  "label":"another test",
  "location":{
    "longitude":23.729309999999998,
    "latitude":37.983715
  },
  "creator":{
    "id":"52bf58b774d27e9323764752",
    "firstName":"Giannis",
    "lastName":"Skevakis",
    "email":"giannis.86@gmail.com",
    "joinedOn":1388271799841,
    "role":"USER"
  },
  "creatorId":"52bf58b774d27e9323764752",
  "multimedia":[
  {
    "type":"image",
    "id":"52ca6f1374d2423694ed75c0",
    "fileId":"52ca6f1374d2423694ed75bd",
    "height":1200,
    "width":1920
  },
  {
    "type":"video",
    "id":"52ca6f1774d2423694ed75cf",
    "fileId":"52ca6f1674d2423694ed75c3",
    "height":360,
    "width":202,
    "duration":28
  }
  ],
  "statistics":{
    "id":"52ca6f1e74d2423694ed75d6",
    "views":57,
    "likes":0
  }
}
```

Table B.29: RESTful Service: Get object's multimedia.

URL	/object/{id}/multimedia
Method	GET
Parameters	*id*: the object's id *count*: the number of elements for each page (default: 10) *page*: the page number (default: 0)
Description	Returns the multimedia about a single object, sorted by creation date and with paging.
Example	

```
GET /object/52b70bc6e4b032a73a7c1aa5/multimedia?count=10&page=0


{
  "content":[
    {
      "id":"52b70c4be4b032a73a7c1b0c",
      "objectId":"52b70bc6e4b032a73a7c1aa5",
      "fileId":"52b70c4ae4b032a73a7c1b00",
      "height":360,
      "width":202,
      "duration":28
    }
  ],
  "size":10,
  "number":0,
  "totalElements":1,
  "numberOfElements":1,
  "totalPages":1,
  "firstPage":true,
  "lastPage":true
}
```

Table B.30: RESTful Service: Delete multimedia.

URL	/multimedia/{id}
Method	DELETE
Parameters	*id*: the multimedia's id
Description	Deletes the multimedia.
Example	

```
DELETE /multimedia/52b98c0ee4b013c9e11dccb5
```

B.6 Statistics

Table B.31: RESTful Service: Get user statistics.

URL	/stats/user/{id}
Method	GET
Parameters	*id*: the user's id
Description	Returns the statistics of the user.
Example	

```
GET /stats/user/52b98c0ee4b013c9e11dccb5
```

```
{
  "id":"52b98c0ee4b013c9e11dccb5",
  "observations":23,
  "multimedia":12
}
```

Table B.32: RESTful Service: Get observation statistics.

URL	/stats/observation/{id}
Method	GET
Parameters	*id*: the observation's id.
Description	Returns the statistics of the observation.
Example	

```
GET /stats/observation/52b98c0ee4b013c9e11dccb5
```

```
{
  "id":"52b98c0ee4b013c9e11dccb5",
  "views":235,
  "likes":12
}
```

Table B.33: RESTful Service: Get object statistics.

URL	/stats/object/{id}
Method	GET
Parameters	*id*: the object's id.
Description	Returns the statistics of the object.
Example	

```
GET /stats/object/52b98c0ee4b013c9e11dccb5

{
  "id":"52b98c0ee4b013c9e11dccb5",
  "views":235,
  "likes":12,
  "observations":90,
  "multimedia":121
}
```

Table B.34: RESTful Service: Get application profile type statistics.

URL	/stats/type/{type}
Method	GET
Parameters	*type*: the type's name.
Description	Returns the statistics of the application profile type.
Example	

```
GET /stats/type/animals

{
  "name":"animals",
  "objects":10,
  "observations":90
}
```