Design and Implementation of a Platform for Supporting Annotation of indoor objects and Navigation in complex interior spaces.

By

Petros I. Kontogiannis

A thesis submitted in partial fulfillment of the
requirements for the

Diploma of
Electronics & Computer Engineering

at the

School of Electronics & Computer Engineering
Technical University of Crete
May 2016

Dedicated to my family

The engineer who gave me my father, the gentle dynamism of my mother
and the rhythm of life that gives my sister...

Abstract

Recent international surveys present that humans spend over 75% of their time indoors, however their movements are rarely limited to streets routes and mainly take place within buildings of different size and shape, since they are able to roam almost in any free space of a building. Furthermore, the complexity of buildings is growing constantly and visitors want to feel familiar inside buildings and they want to be able to search, find and finally reach their desired destination through a sequence of simple instructions. They also want to know more information about the objects that they are located front of them, in order to acquire a complete view. To this end, any tool supporting the creation of floor plans, management of their content and generation of navigation instructions is considered invaluable both for visitors of buildings and owners. In this thesis we discuss the design and implementation of a platform for the design, creation and management of interior spaces. More specifically, we present an editor tool (desktop application) supporting the design, creation and management of floor plans, as well as floor plan editing, definition of primitives structures and give annotation to each containing object. Furthermore, we present the mobile application that we have developed, supporting the navigation inside the buildings which have been created using the editor tool. The mobile application is compatible with most state-of-the-art mobile devices and platforms, while both tools have been designed with flexibility and extensibility in mind. To this end, they can be used in various domains including universities, museums, shopping malls, as well as every large building. Finally, is worth mentioning that our applications have been evaluated for their usability by professionals and naive users.

Acknowledgements

I would like to express my gratitude to my supervisor, Prof Stavros Christodoulakis, for his encouragement and his continuous guidance and support not only throughout my thesis but also my studies. I would also like to thank him for the important experiences he offered me during my stay at the Laboratory of Distributed Multimedia Information Systems and Applications (TUC/MUSIC), as well as for the positive influence in broadening my horizons. But the most important thing is that Mr. Christodoulakis was the main source of inspiration for me and made me try daily for the best.

Special thanks go to Nektarios Gioldasis for his support, his valuable advices, as well as our long fruitful discussions. I would also like to thank you Prof. Antonios Deligiannakis and Prof. Minos Garofalakis for serving on my thesis committee.

I would like to thank you all my colleagues in the TUC/MUSIC Laboratory for their support and for the pleasant environment they provided. My sincerest gratitude also goes to all my friends for the memorable moments we lived together during our studies.

Finally, this thesis would not have possible without the support and encouragement of MY PARENTS. My mother Maria and my father Isidoros were and are always there whenever I need them. Every hour, every day, every moment are there and constantly support me everywhere. At this point, I want to use Greek language in order to express exactly what I want.

Thank you from the bottom of my heart for your never ending love and care.

I am grateful to you...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Inspired by the achievements of satellite-based location services in outdoor applications the challenge has shifted to the provision of such services for the indoor environment. However, the ability to locate objects and people indoors remains a substantial challenge.

Humans spend over 75% of their time indoors, therefore their movements are rarely restricted to specific streets routes and mainly take place inside buildings of different size and shape. Considering the growing complexity of new buildings, the interest has shifted to providing navigation guidance for indoor spaces. Unfortunately, due to a number of GPS limitations, obtaining accurate indoor tracking or positioning is rarely possible, making indoor mobility, navigation, and movement analysis a really challenging topic.

In general, the design requirements of indoor navigation systems are different from outdoor navigation systems in several ways. First of all, the coordinate system and the precision measurement, supported by the indoor location infrastructure, often differs from GPS. Secondly, navigation within a building is different from navigation in the road system. Unlike cars, which are restricted by the road network, mobile users are able to roam almost in any free space of a building due to the lack of well-defined pathways (roads, cyclic nodes, etc.) in it. Consequently, the techniques used for charting a path between any indoor origin and destination are different from those used in charting paths on the road network. In more details, the road system can be represented as a huge graph of vertices (nodes) and edges (pathways). Finally, it is so important that in case of outdoor navigation, there are many commercial and public digitalized maps and Geographical Information System databases for the road network whereas maps of buildings and campuses are not. In this way, it is easy to take all these information and follow the appropriate standards in order to implement your own GPS navigation system for outdoor environment.

The implementation of indoor navigation systems is by far more complex. In indoor environments there is a great variety of different structures and the CAD building floor plans are not represented in a form that can be easily processed. This is the reason why an implementation of indoor navigation systems should be conducted on a custom basis. Such a deployment model requires a set of tools to automate the process of creating indoor maps. Through this process, the owners of the buildings should be able to create and design indoor maps with specific characteristic according to their needs. These indoor maps may contain multiple structures, such as rooms, stairs, small objects, etc. In each case, the creator of building should be able to choose a structure from a set of primitive structures or define his desired structure with specific characteristics. For example in case of a museum, such structures may as well be an exhibit room, an elevator or a statue. All of these primitive structures can be described using a specific set of characteristics, such as the subject of an exhibit, the material of an exhibit, etc. Such characteristics might be of major importance for the visitor of the building.

Here is a use case scenario demonstrating the challenges that a visitor might face while navigating inside a building: Imagine that it is your first time visiting the Museum of Pergamus in Berlin or the Acropolis museum in Athens and you have a strong urge to see a popular exhibit. Although the museum is filled with numerous magnificent works of art, the museum is about to close and you would like to head directly to admire your favorite exhibit, the main reason why you visited this museum. After entering the museum, you arrive at the first hall and eagerly look for signs that lead you to your favorite exhibit. But there is not anything helpful for you. Then you try to ask a museum attendant for help in order to reach your destination but she only gives a long sequence of directions so complicated that you cannot possibly hope to follow them. After wandering around for a while, you find a pamphlet or a panel with floor map of museum that indicates the location of your favorite exhibit. Holding the pamphlet in your hand, you can feel hopeful and excited that you will soon see the desired exhibit. But soon, those feelings turn into feelings of defeat and frustration. As you walk through different rooms, you have troubles locating yourself on the map; the landmarks surrounding you, do not seem to correspond to those indicated on the map. Moreover, you discover that a variety of rooms are closed for the day, and the path that you are currently taking will not get you to your eventual goal.

This is a common scenario that can happen whenever people try to find a particular place, person or object inside an unfamiliar or complex environment. Such complex environments can be airports, museums, conference centers, exhibitions, corporate and school campuses, hospitals, shopping malls, stadiums, train station, public buildings, supermarkets, and urban areas. Such examples trigger the need for a system that assists the navigation of users in real time by providing a sequence of evolving directions in order to reach the desired destination. Such a system should take into account not only the user's position on the map but also the objects spatial relations. For example, if you know that your favorite painting is near to a huge statue and opposite another, you can find it more easily. The relations between objects are very useful and provide a better and easier navigation inside a building. Additionally, as we have already mentioned, it is very important for a visitor to obtain more information about the object that he wants to find or the object that is in front of him. In case of a painting, the visitor wants to know more information about it, such as its material, culture, artist, etc in order to acquire a overall view about the object. In this way, we can enable the user to search and find the desired objects depending on a specific set of characteristics. For example, a museum visitor may want to find artworks related to the impressionism or Ancient Greek statues.

In this thesis, we present the design and implementation of a platform for supporting annotations of indoor objects and navigation in complex interior spaces. The structures that we support, follow a flexible model, allowing our framework to support a great variety of buildings. More specifically, we present our editor tool (desktop application) supporting the creation, design and management of floor maps, as well as the creation and definition of primitive structures and objects. Furthermore, we present the mobile application that we developed for supporting indoor navigation in buildings that have been created using our editor tool. Our application is compatible with most state-of-the-art mobile devices/platforms and offers among others: (*a*) search functionality allowing users to track certain objects based on specific characteristics, (*b*) indoor navigation assistance by providing directions, and (*c*) QR code scanning as a supplementary method for identifying the user location inside the building.

Both the editor and the mobile application have been designed with flexibility and extensibility in mind and can be used in various domains, including museums, supermarkets, universities, shopping malls, as well as in every large building infrastructure.

The aim of this thesis is to: (*a*) define the model needed to represent building design created by both professional and novices users, (*b*) design and implement an infrastructure supporting the model, (*c*) design and implement an editor tool (desktop application) supporting the creation and management of building conforming to the model, and (*d*) design and implement a mobile application supporting the navigation and information retrieval inside buildings which have been created by our editor tool.

1.1 Summary of Contributions

In this thesis, we present a framework for the creation and management of Interior building spaces.

In more details, we propose a model for describing objects of interior spaces according to the kind of building and the user needs. The model refers to the overall information concerning the creation of a building. The aim of constructing the model was to support as many as possible types of building. To this end, we tried to parametrize it depending on different contexts, by defining information that constitutes each type of building.

Additionally to the 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 to external systems. The client applications are the graphical user interfaces that the users interact with during the management of building and its objects (editor tool - desktop application) and retrieving information about them (mobile application). The whole technical platform has been built using state-of-the-art web technologies, and the provided tools include a native mobile application compatible with all the major mobile platforms.

Our novel infrastructure supports the modern way of creating buildings and its objects and offers strong flexibility about the different structures (types of objects), giving you capabilities of creation and definition of structures and give them a set of characteristics and attributes. By doing so, we achieve the better and faster familiarization of visitors with the space of building using mobile devices and their tools (camera, map, compass etc.). This infrastructure can be applied in a variety of buildings such as hospitals, museums, airports, universities, retails, stadiums, shopping malls, hotels, libraries and other large buildings in order to cover the needs and objectives of these buildings and their visitors.

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 for the building construction. In the second part, we describe the infrastructure that we developed in order to support the model and the third part we present the user 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 (AngularJS), web applications/services (Spring Framework) and modern database systems (Mon-

goDB).

- chapter 4 describes the functional specifications of the system that has been developed for the management of the whole different types of buildings.
- chapter 5 specifies the model for describing building and its indoor structures. This model refers to the overall information concerning the creation and design of buildings.
- chapter 6 describes the architecture that has been designed and implemented for the infrastructure.
- chapter 7 describes the implementation details of some of the components in more details, providing some more insight on how specific parts of the system have been implemented.
- chapter 8 presents the user interfaces that have been developed for the interaction with the user.
- chapter 9 summarizes and reviews the presented work and sketches some perspectives for future extensions.

Chapter 2

Related Work

In this chapter, we present tools and applications that are considered relevant to our work. For each of the systems described below, we provide the objective that they aim to reach, present their capabilities and discuss their functionality. Additionally, we compare them to the tools and services that we developed, focusing on the strengths and weaknesses they possess. To the extend of our knowledge, there is no platform or tool that combines the creation of building and its objects, the automatic construction of navigation and instructions directly available in any mobile device and tablet, as well as the dynamic creation and definition of structures and give semantic of containing objects.

2.1 Google Indoor Maps

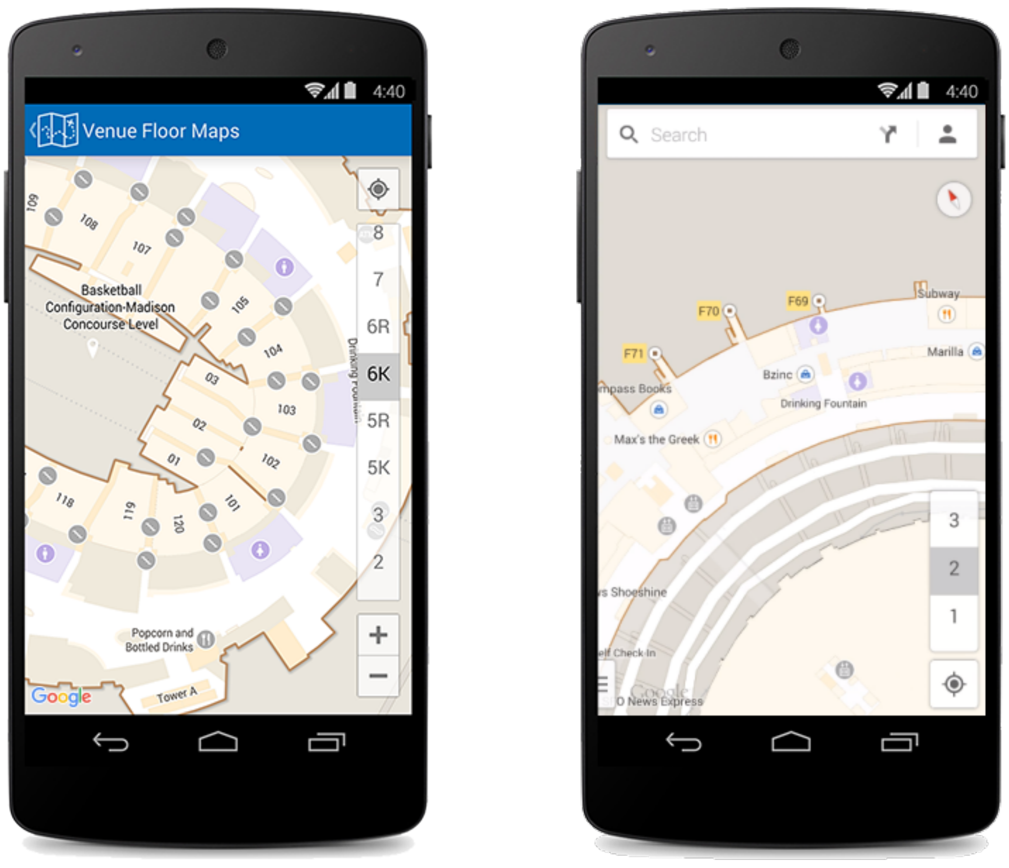
Google Indoor Maps[1] is an Indoor Positioning System developed by Google. More specifically, GIM is a web platform that offers the capability to preview a floor map, specifically an image, which is uploaded by a user.

Administrator users can draw an image with the floor plan of each floor of building and set it up to the area of building on Google Maps. In this way an administrator can "compose" a building consisting of images. Each image should have been placed on Google Maps in the desired orientation and the appropriate level and then should be approved by Google Administrators.

On the other side, a simple user can view and retrieve their position on this image using GPS functionality of his mobile device. By using GPS, Google achieves a limited accuracy, about 3-5 meters. Figure 2.1 presents the Google Indoor Maps mobile application in action, displaying two of the floor plans of Madison Square Garden Arena, in the New York area.

[1] `https://www.google.com/maps/about/partners/indoormaps/`

The main purpose of GIM is to help the visitors of a building to navigate inside it in order to save effort and time while trying to reach the desired destination.



(a) The floor plan of Madison Square Garden Arena. **(b)** The floor plan of San Francisco International Airport.

Figure 2.1: Google Indoor Maps.

Compared to our application, GIM is more limited because it does not provide the editor tool to the user in order to draw and design the desired floor plans. Also Google should approve the uploaded images in order to publish them to the users. On the contrary, we provide an editor tool and a mobile application which are able to support the creation and design of an extended variety of buildings and navigation inside them. Using our tools, the user is able to create his own objects and their specific types and of course to give semantic to them in order to give the ability to the simple users to search, retrieve and navigate at specific location. Furthermore our system provides a sequence of instructions to help the mobile user to reach his destination in a simple and fast way.

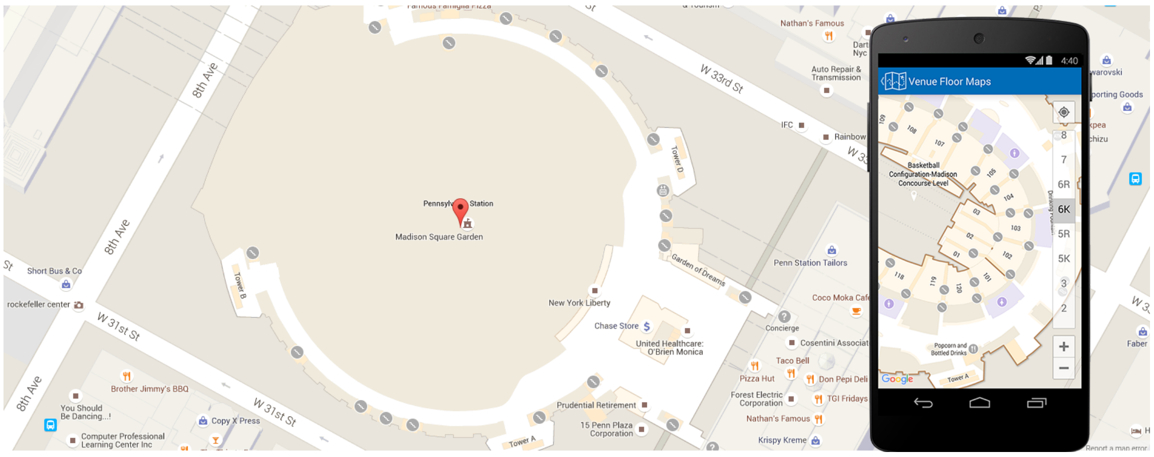


Figure 2.2: The floor plan of Madison Square Garden Arena - Google Maps

2.2 Nokia HAIP

Nokia[2] has its own indoor location technology called HAIP (High Accuracy Indoor Positioning) based on BlueTooth Low Energy beacons (BLE). With this combination of software and hardware the user can locate his position inside a building and is able to retrieve a 3D map of each floor of building. In this case the owner of a building should send all relevant information, such as floor plans, photographies of spaces etc, to the Nokia administrators in order to design and represent the interior spaces in form of a 3D Map. In this way, the user is able to identify his position in relation to the objects which are located around and navigate inside building during his wondering. The system previews a very accurate visualization of floor and containing objects in real time and the user is able to locate himself. The most impressive feature of Nokia technology is the accuracy since it is possible to locate the mobile devices in accuracy of 50 cm. Figure 2.3 presents a screenshot of the Nokia HAIP user interface, displaying a 3D map of a conference center.

Compared to our work, Nokia HAIP is limited because it does not offer the editor tool and therefore all of the implemented floor should be generated with programming language from a team of administrators. Additionally, it does not take into account the user's instructions in order to navigate him inside the building. The last but not least is that our floor plan editor gives the user the capability to give semantic at the object of the floor and customize their type and their attributes. In this way, the objects and their positions can be retrieved from our mobile application in case the user wants to navigate there through a sequence of instructions.

[2]`http://www.nokia.com/el_gr`

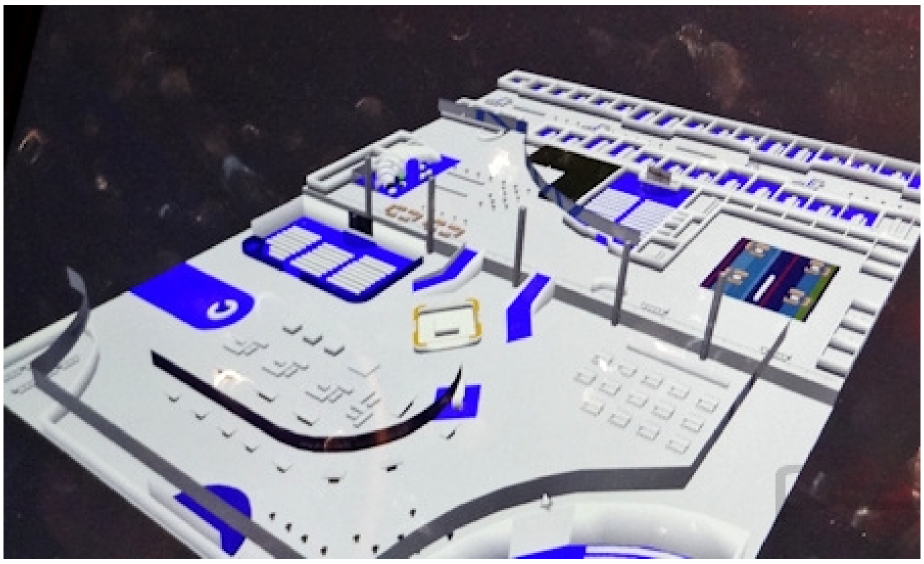


Figure 2.3: Nokia HAIP user interface presents the interior of Conference Center.

2.3 Bing Indoor Maps

Bing Indoor Maps[3] is a very similar system with Google Indoor Maps and offers about the same capabilities. Like Google, and Bing Maps allow to the user, who wants to design a building, to upload a set of images and information about the building which give the required information about the floor plans. In the next step, the administrators of Bing Maps create a composition of floor plans in the appropriate format in order to be previewed to the mobile user. Figure 2.4 presents a screenshot of the Bing Indoor Maps user interface, displaying a map of the first floor of the British Museum, in London.

Although Bing Indoor Maps technology uses the GPS and WiFi to locate the mobile devices inside buildings, the accuracy of the system is about to 2-3 meters, because the deviation of GPS is further in the interior spaces.

It is worth mentioning, that Bing Maps in contrast with Google, contains not only public buildings but also private buildings and in this way the users can design an extended variety of buildings. Figure 2.5 presents a screenshot of the Bing Indoor Maps user interface, displaying a map of the first floor of known department store, in the Moscow area.

On the contrary, the building editor of our application can be used by everyone who wants to design a building, either it is for a public building or not. Furthermore, our application provides further information about the spaces of building and not only the name of those. Using our editor tool the user is able to describe the desired spaces, as well as possible, with an extended list of

[3]`http://www.bing.com/maps/`

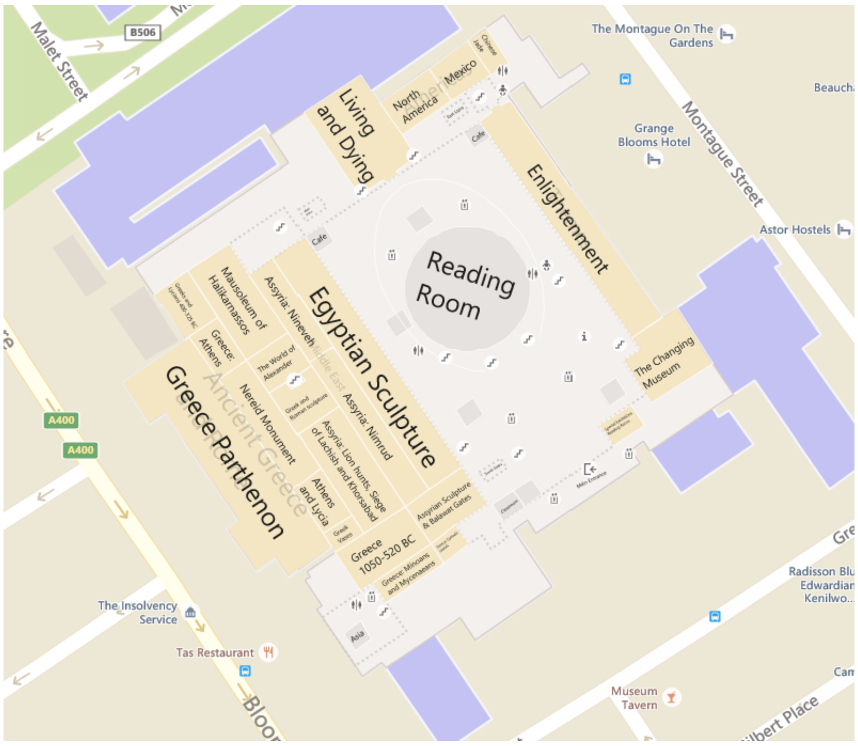


Figure 2.4: Bing Indoor Maps - Screenshot of the user interface presents the floor map of British Museum.

characteristics such as description, category, images etc. Finally, through our application the visitor of a building is able to receive navigation instructions in order to navigate inside the building and reach his destination.

2.4 WifiSlam

WifiSlam[4], is a mobile application software which focuses on fast navigation in interior spaces. First of all, an administrator set up into application the positions of WiFi sensors and then note them on image of floor plan. After that, the administrator user uploads images of floor plans to the system. On the other side the final user who wants to navigate into the building scan the appropriate image and then is able to locate him into this building, in real time. Through this application the final user can take his position with accuracy of 2 to 3 meters because the signal strength of WiFi is not constant during the time. In other words, the WiFi signal strength is changing constantly depending on the number of mobile devices which are connected with this. As opposed to our application, WiFiSlam does not offer an editor tool in order to create a floor plan and give semantic on containing objects. Using the scanned image, the user is able to locate himself but he is not able

[4]`https://angel.co/wifislam`

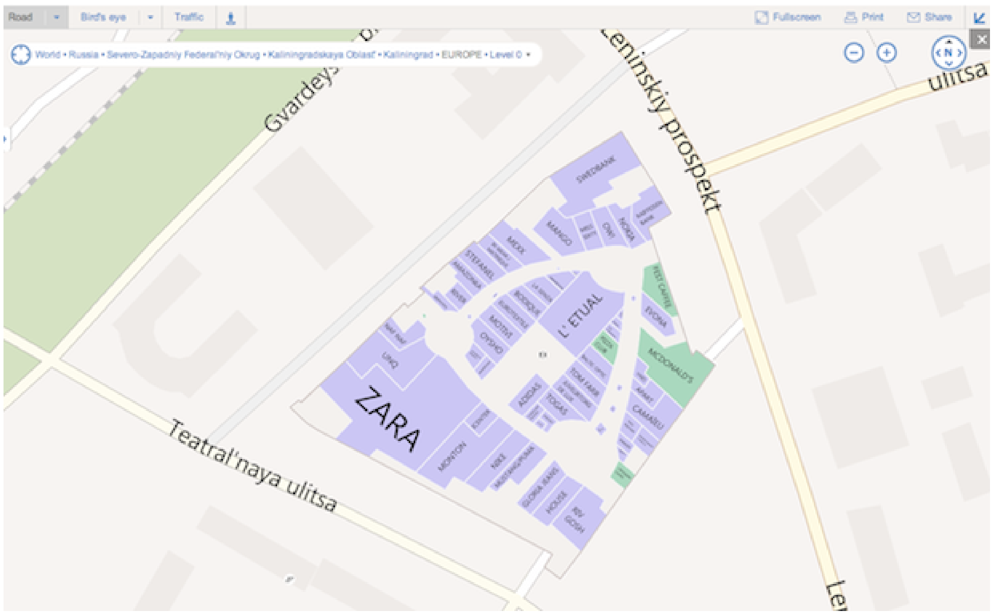


Figure 2.5: Bing Indoor Maps - Screenshot of the user interface presents the floor map of Shopping Mall.

to interact with the map in order to retrieve information about the objects. The last but not least difference with our application, is that the WiFiSlam software does not navigate the user inside the interior spaces, and as a result the user can not find the appropriate path to reach his destination.

2.5 WiFarer

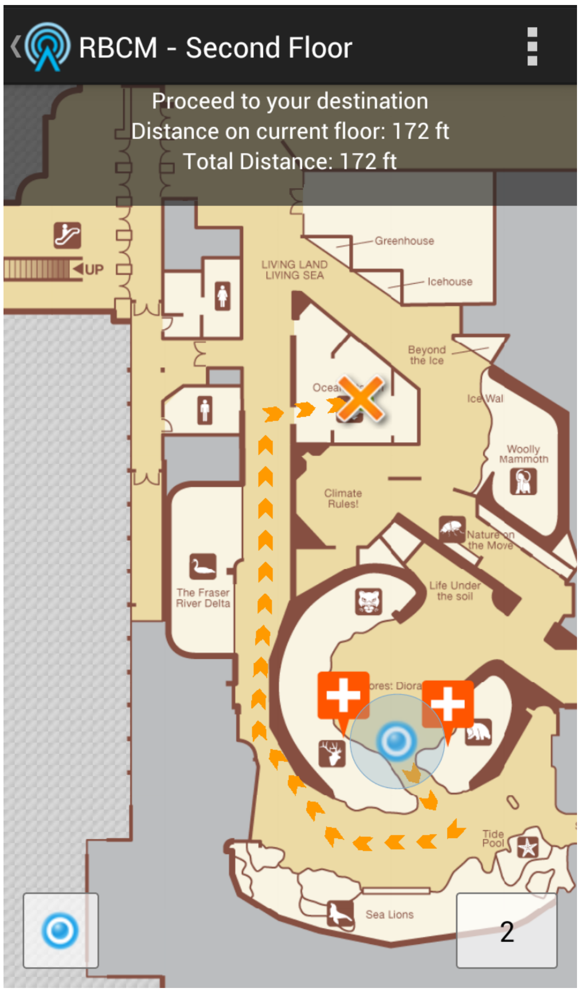
WiFarer[5], is a web platform for setting up a number of floor plans and navigate inside them. In the first step, the owner of a building can sign up into this platform and upload the floor plan images of his building accompanied with the appropriate information about the containing objects. After that, a team of specialists create a floor map, in the appropriate format, with all the required information depending on the user needs.

Furthermore, WiFarer provides an editor for the management of objects which are contained inside the building. In this way the owner of the building is able to manage (upload media, edit the title, description, etc.) the content of these objects according to his needs.

WiFarer provides the appropriate hardware of bluetooth or WiFi sensors in order to achieve the positioning of the final user. Similarly to our case, the WiFarer offers a mobile application in order to for the users to install it to their mobile devices and navigate inside the building.

Compared to our application and services, WiFarer does not offer the ability to the user, to change the floor plans using editor tools. Thus, if someone wants to change a floor map of a build-

[5]`http://www.wifarer.com/`


RBCM - Second Floor

Proceed to your destination
Distance on current floor: 172 ft
Total Distance: 172 ft

Figure 2.6: WiFarer - Screenshot of the mobile application during navigation.

ing, he should contact the administrator of the system and give them all the required information and files in order to set up the desired changes. Furthermore, unlike our application, the WiFarer does not offer the capability of extension concerning the primitive structures and also it is capable to preview information about an object without giving more information and semantic on that.

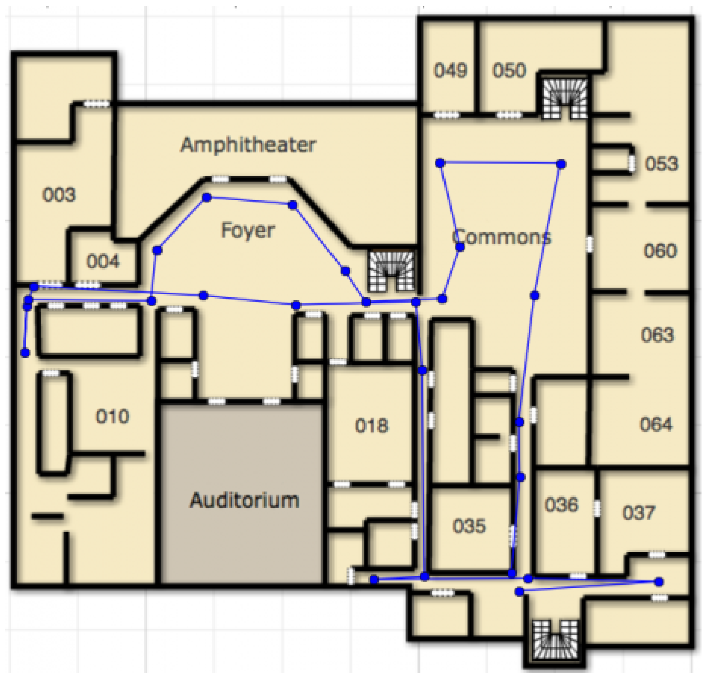


Figure 2.7: WiFarer - Screenshot of the floor plan of editor.

Apart from the above, the most important limitation of WiFarer is that their mobile application is designed only for iPhones. On the contrary, we provide our mobile application for the majority of mobile phones operating system including iOS, Android and Windows. Additionally, it is worth mentioning that our application is also available through the browser.

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 databases system used for persisting the data. Section 3.3 discusses PhoneGap, an open-source mobile development framework. Section 3.4 presents JavaScript, the scripting language used mainly for the implementation of client side logic and the interaction with the users, as well as the JavaScript libraries used. Section 3.5 describes Xuggle, the library used for analysing and manipulating video and audio files. Section 3.6 presents AngularJS, the JavaScript framework that we user to build our client side applications.

3.1 The Spring Framework

The Spring Framework [65, 66] 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 JavaBeans (EJB) model.

Modules

The Spring Framework consists of features organised into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP(Aspect Oriented Programming), Instrumentation, Messaging, and Test, as shown in the following diagram Figure 3.1.

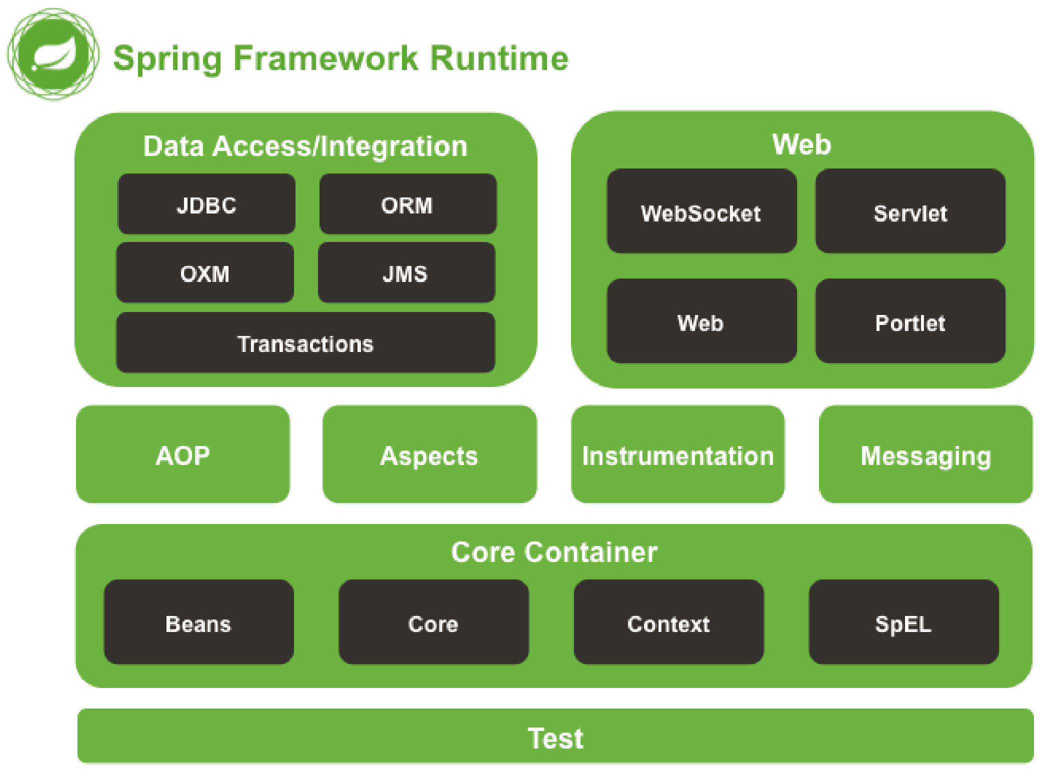


Figure 3.1: Overview of the Spring Framework

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 Inversion of Control (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 content 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 spring-orm module 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 such as JAXB, Castor, XMLBeans, JiBX and XStream.

- The Java Messaging Service module contains features for producing and consuming messages. Since Spring Framework 4.1, it provides integration with the spring-messaging module.

Web

The Web layer consists of the spring-web, spring-webmvc, spring-websocket, and spring-webmvc-portlet modules.

- The spring-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 an HTTP client and the web-related parts of Spring's remoting support.
- The spring-webmvc module (also known as the Web-Servlet module) contains Spring's model-view-controller (MVC) and REST Web Services implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.
- The spring-webmvc-portlet module (also known as the Web-Portlet module) provides the MVC implementation to be used in a Portlet environment and mirrors the functionality of the spring-webmvc module.

Test

The spring-test module supports the unit testing and integration 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 applications 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 life-cycle 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, or factory 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 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.[14][not in citation given] 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 that offers managed data access environments outside of an application server or container.

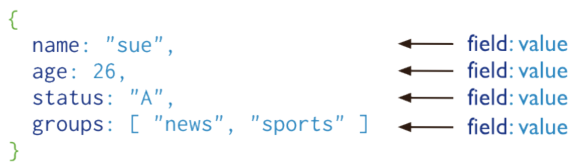
3.2 MongoDB

MongoDB [46, 20] 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 [17]), 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.

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.

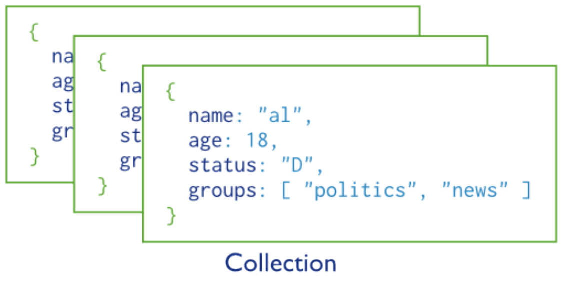
MongoDB stores data in the form of documents, which are JSON-like field and value pairs. Documents are analogous to structures in programming languages that associate keys with values, where keys may hold other pairs of keys and values (e.g. dictionaries, hashes, maps, and associative arrays). Formally, 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.



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

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

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.



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

Figure 3.3: A MongoDB collection of documents.

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

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 provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centers can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

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.

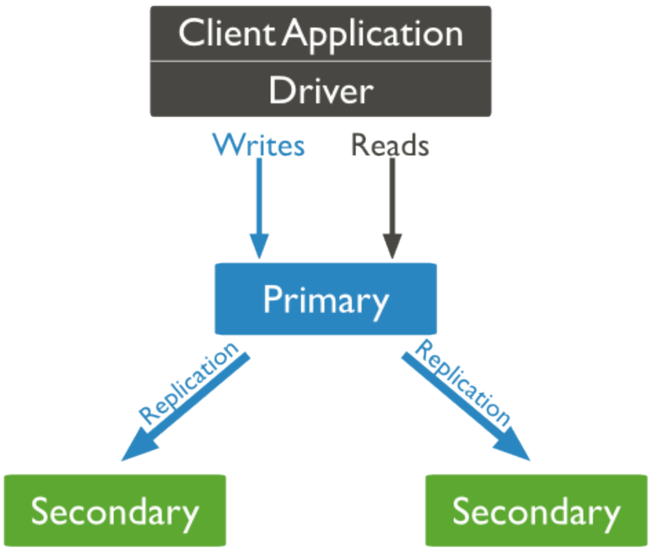


Figure 3.4: Default routing of reads and writes to the primary.

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.

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.

Load Balancing

MongoDB scales horizontally using sharding. The user chooses a shard key, which determines how the data in a collection will be distributed. The data is split into ranges,based on the shard key, and distributed across multiple shards. Alternatively, the shard key can be hashed to map to a shard - enabling an even data distribution.

MongoDB can run over multiple servers, balancing the load duplicating data to keep the system up and running in case of hardware failure. MongoDB is easy to deploy, and new machines can be added to a running database.

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.

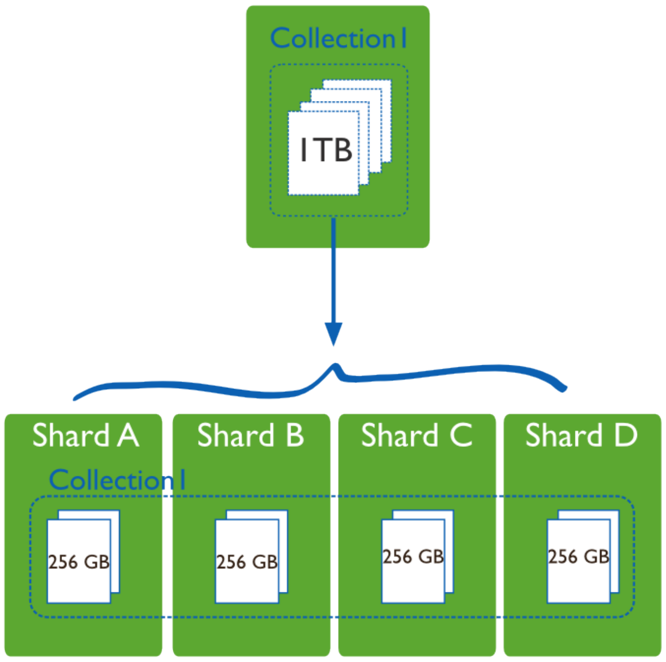


Figure 3.5: Large collection with data distributed across 4 shards.

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

- Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows. As a result, a cluster can increase capacity and throughput horizontally. For example, to insert data, the application only needs to access the shard responsible for that record.

- Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows.

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.

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.

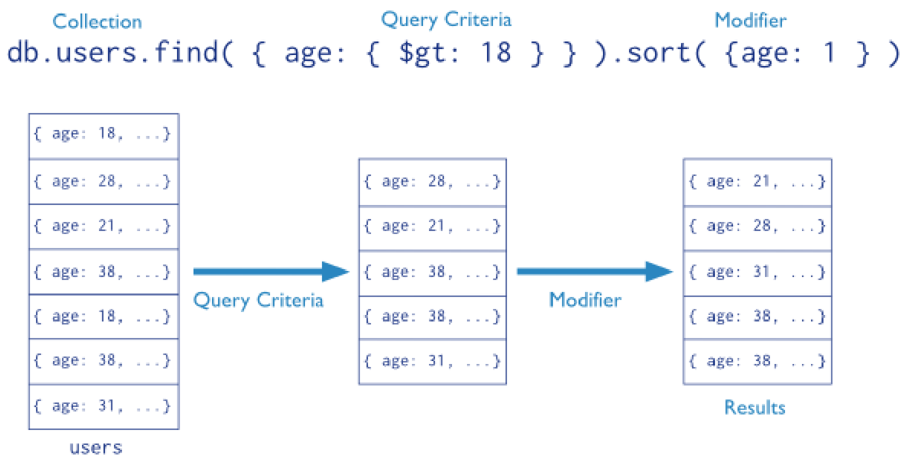


Figure 3.6: The stages of a MongoDB query with a query criteria and a sort modifier.

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.

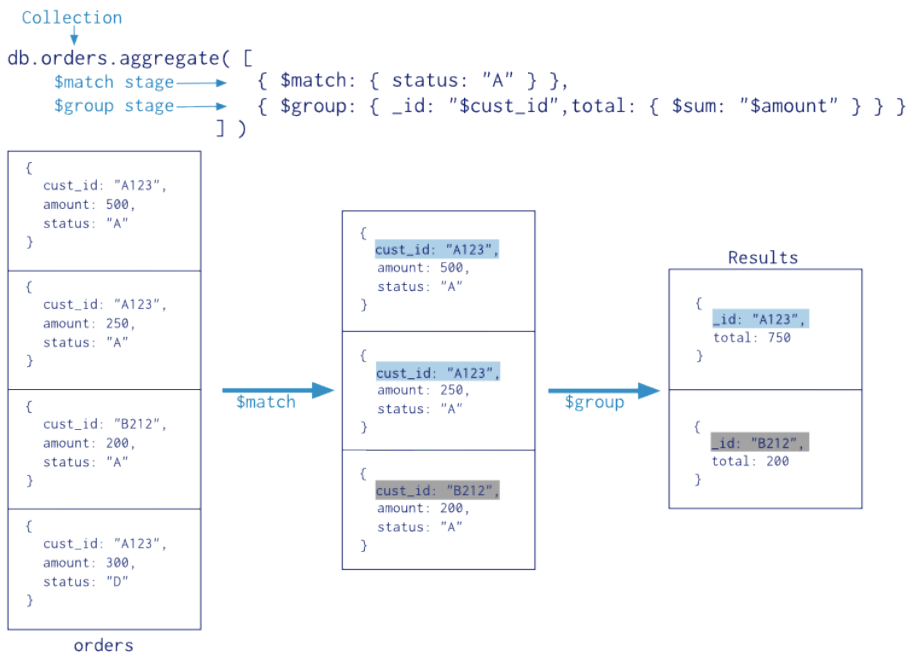

Collection
db.orders.aggregate([
$match stage
{ $match: { status: "A" } },
$group stage
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
$match
$group
Results
orders

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.

3.3 Phonegap

Phonegap [55] is an open-source mobile development framework. It allows the use of standard web technologies such as HTML5, CSS3, and JavaScript for creating native mobile applications, avoiding the interaction with 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 and want 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 (special browser window) that can access device-level APIs, or if you want 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, by default a local file named index.html, 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.

The Cordova-enabled WebView may provide the application with its entire user interface. On some platforms, it can also be a component within a larger, hybrid application that mixes the Web-

View 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 executable 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 [27] 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 [48].

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. Most web sites were based on complete HTML pages where each user action required that the complete 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) [68, 34] 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 [28] 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 [15] 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 Xuggler

Xuggler [73] is a Java library that allows developers to easily decompress, 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.

3.6 AngularJs

AngularJS [10] is an open-source web application framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model-view-controller (MVC) and model-view-viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

The AngularJS framework works by first reading the HTML page, which has embedded into it additional custom tag attributes. Angular interprets those attributes as directives to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code, or retrieved from static or dynamic JSON resources.

Core Features

The core features of the AngularJS framework are presented in Figure 3.8 and most of them are described below in greater detail.

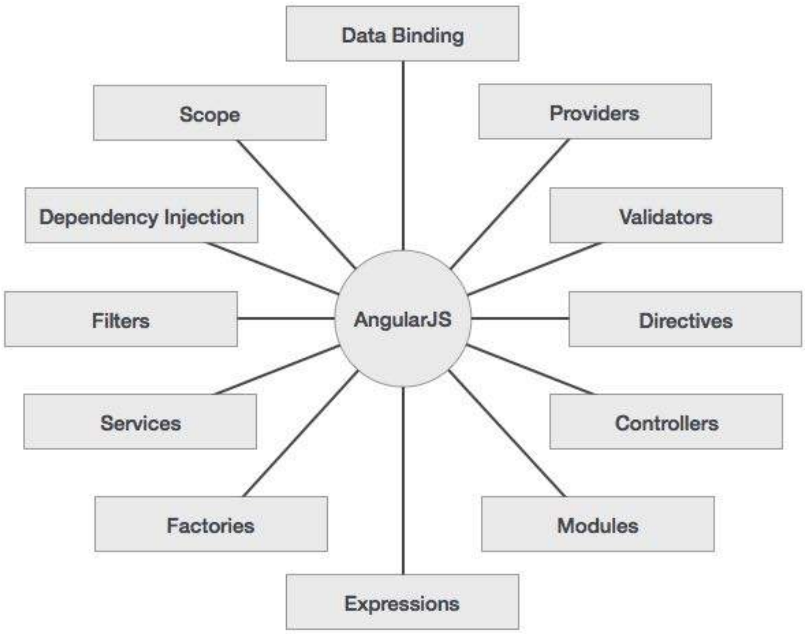
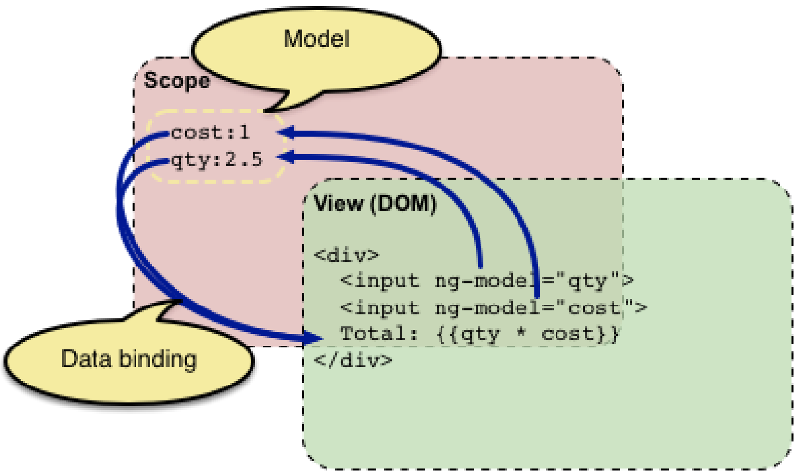


Figure 3.8: Important parts of AngularJS.

Data-binding

Data-binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa. An example of two way data-binding is presented in Figure 3.10.



```
<div>
  <input ng-model="qty">
  <input ng-model="cost">
  Total: {{qty * cost}}
</div>
```

Figure 3.9: Two way data binding in AngularJS.

Scope

Scope is an object that refers to the application model. It is an execution context for expressions. Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watch expressions and propagate events.

Controller

In Angular, a Controller is a JavaScript constructor function that is used to augment the Angular Scope. When a Controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified Controller's constructor function. A new child scope will be available as an injectable parameter to the Controller's constructor function as $scope.

Services

Angular services are substitutable objects that are wired together using Dependency Injection (DI). You can use services to organize and share code across your app. The Angular services are:

- Lazily instantiated - Angular only instantiates a service when an application component depends on it.
- Singletons - Each component dependent on a service gets a reference to the single instance generated by the service factory.

Filters

A filter formats the value of an expression for display to the user. They can be used in view templates, controllers or services and it is easy to define your own filter.

Directives

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's HTML compiler ($compile) to attach a specified behavior to that DOM element or even transform the DOM element and its children.

Angular comes with a set of these directives built-in, like ngBind, ngModel, and ngClass. Much like you create controllers and services, you can create your own directives for Angular to use. When Angular bootstraps your application, the HTML compiler traverses the DOM matching directives against the DOM elements.

Templates

In Angular, templates are written with HTML that contains Angular-specific elements and attributes. Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

Model View Controller

MVC is a design pattern for dividing an application into different parts (called Model, View and Controller), each with distinct responsibilities. AngularJS does not implement MVC in the traditional sense, but rather something closer to MVVM (Model-View-ViewModel). A definition of MVC pattern in AngularJS is presented in Figure 3.10.

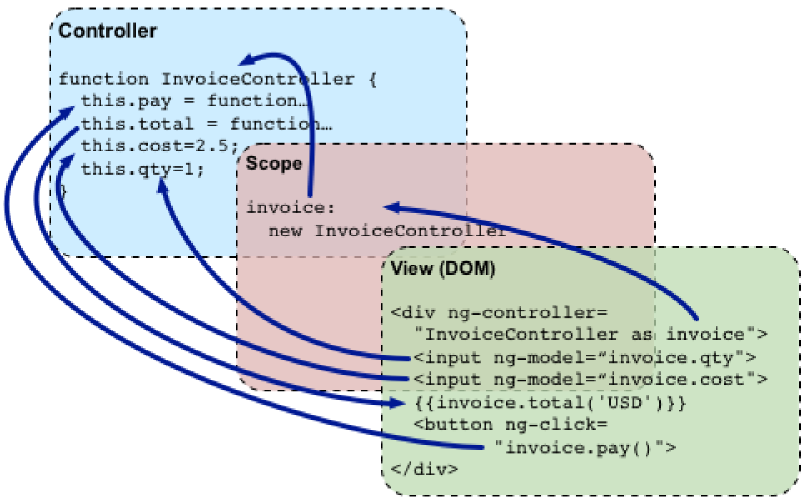


Figure 3.10: Definition of MVC pattern in AngularJS.

MVC is popular because it isolates the application logic from the user interface layer and supports separation of concerns. The controller receives all requests for the application and then works with the model to prepare any data needed by the view. The view then uses the data prepared by the controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.

Forms

Form and controls provide validation services, so that the user can be notified of invalid input before submitting a form. This provides a better user experience than server-side validation alone because the user gets instant feedback on how to correct the error. Keep in mind that while client-side validation plays an important role in providing good user experience, it can easily be circumvented and thus can not be trusted. Server-side validation is still necessary for a secure application.

Deep Linking

Deep linking allows you to encode the state of application in the URL so that it can be bookmarked. The application can then be restored from the URL to the same state.

Dependency Injection

Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies. The Angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

Chapter 4

Functional Specification

This chapter describes the functional specification of the platform and tools that have been developed for the management of the life-cycle of the structures and their instances (e.g objects). It specifies the stakeholders of our 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 4.1 specifies the system's stakeholders and their role in our system, while Section 4.2 discusses the technical requirements of the tool. Finally, Section 4.3 demonstrates the functionality that has to be provided by our system in the form of use cases.

4.1 Stakeholders

A stakeholder is someone or something that has a vested interest in the behavior of the use case. Our platform is targeted for organizations and large structures owners who want to facilitate their visitors and give them the ability to become acquainted with their buildings. It is worth noting that we designed our platform in order to be able to accommodate as more as possible different types of buildings. On the other side, our platform is targeted to everyone who wants to navigate and familiarize with a new building which he visits, either he visits a university building, or he visits a museum or a library. During the application's design process we bore in mind that our applications are targeted to two different teams of people, the buildings owners and the buildings visitors. To do this, we tried to implement a user friendly and simple user interface for both of these teams of people. Building's owner, museum's director, hotel's owner or mall's director has different needs in order to describe their buildings. For those scenarios, the system's stakeholders are:

- The **Museum Director** who wants to describe the museum's exhibits in order to give information to the visitors about them. This user wants to give further information for an exhibit such as the description or historical background, and give them characteristics such as the category of this exhibit etc.

- The **University Staff Member** who wants to create a floor plan about the course room, office of professors, conference room or other kind of room in order for a simple visitor to be able to retrieve information about a room such as position, owner etc.
- The **Hotel Owner** who wants to give the ability to the guests to familiarize with hotel's spaces and take pleasure of every different kind of facilities which are offered by this hotel.
- The **Airport Staff Member** who wants to create an extensive and representative floor plan of airport spaces in order to navigate the travellers and visitors to multiple different spaces of airport such as cafeterias, water closets and of course the Gates.
- The **Hospital Staff Member** who wants to describe the floor map of hospital in order to give the visitors the ability to find fast and easily the desired room or department.
- The **Malls Staff** who will use our system to build a floor plan which contains the different kind of shops in order to give the ability to the visitors to search and find the desired shop.
- The **Shipping Companies** who want to give an interactive floor plan to the travellers in order to familiarize with their ships and find the different spaces of ships, such as a cabin, water closet, bars or the lounge.
- The **Library Staff** who want to give a fast and easy way to the user to search and find the position of their desired book or paper and navigate there through a sequence of simple instructions.
- The **Visitors of Buildings** who want to retrieve information about the visited building, navigate inside the building and of course familiarize with the new for them building. By using our application they have the ability to take advantage of all offered facilities.

4.2 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 4 categories. Section 4.2.1 describes the requirements of the Model. Section 4.2.2 describes the requirements of the editor tool that assists the creation, design and management of buildings and the definition and editing of structures, and Section 4.2.3 describes the requirements of the mobile application. Finally, Section 4.2.4 describes the requirements of the system's backend.

4.2.1 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 structures. However, we have identified four entities that form the base of our model. These are:

- Building: The building is a very important component of our model. It should be created using an editor tool and retrieve it by a mobile application. It consists of several parts, called floors, including the floor composition (e.g objects), the presentation information and the descriptive meta-data.
- Floor: The floor is an integral part of the building. In essence, floor is the plan that is being used by the user in order to draw and design the desired floor plan. The floor contains a number of objects that represent all the different objects of this floor, such as rooms, stairs or movable objects (exhibits, statues, etc.).
- Object: The object is the cornerstone component of our model. An object is a structure that is instantiated in a certain building and floor. This may be a statue with its own characteristics and different attributes, such as the category, the height, the sculptor etc.
- Structure: The structure is the primitive type of object. A structure may be indoor structure, multi floor structure or movable structure. User is able to create the desired structure and give it a list of different attributes corresponding to his needs. An example of structure might be the secretary office or a conference room.

Some examples describing the context that our model can be applied, include the following:

- **Museums**: In the context of museum, our model should provide support for creation and designing objects which are related to museums. Such objects need to present both textual information and multimedia (photos, videos, audios). Moreover, it is very important to give the ability to the designer to create the desired types of objects (structures) in order to accommodate the building's needs. In case of the museum, the designer can create a structure of a movable object which is characterized by attributes which are referred to the material of the exhibit, the creation date, the artist etc. Additionally, in this case the ability of uploading multimedia files is very useful in order to provide further and representative information to the visitors of the museum.
- **Hospitals**: In the context of Hospitals, our model should provide support for creating and designing objects which are related to hospitals. It is very useful for visitors to be aware

of the exact location of the specific ward they are looking for. Indoor locations such as classic rooms, emergency rooms and other type of medical rooms can be designed through our application.

- **Universities**: In the context of University, our model should provide support for creating and designing objects which are related to the university environment. Through our application a university staff member is able to create and design multiple types of rooms, such as secretary office, professor office, conference room, courses rooms etc. in order to give the visitors the ability to search and find the desired place.
- **Airports**: In the context of Airport, our model should provide support for creating and designing objects which are related to an Airport building. These objects may be cafeterias that are inside the building, the counters of different Airlines or the most important place of the airport which are the Gates. In this way, the travellers are able to find the desired place in order to save their time and avoid the hassle.
- **Shopping Malls**: In the context of Shopping Malls, our model should provide support for creating and designing objects which are related to this domain. To be more specific through our application the user will be able to design indoor spaces such as shops with different characteristics (category, etc.) and of course will be able to design the variety of different products.
- **Conference Center**: In the context of Conference Center, our model will provide support for creating and designing objects which are related to this domain. In other words the user should be in the position of creating and designing multiple types of rooms. For example it is useful for someone to know the capacity of a room or whether this room provides a projector or other kind of equipment.
- **Libraries**: In the context of Libraries, our model should provide support for creating and designing objects which are related to the library building. To be more specific through our application the user will be able to design interior spaces and objects such as shelves with specific category, books with particular characteristics or newspaper accordingly.

4.2.2 Editor Tool

The editor tool needs to be a desktop application due to the extended functionality that should offer to the building creator. More specifically it should be web based in order to be easily accessible, to have minor setup requirements and allow future collaboration in the building creation process. As a web application it should be compatible with the state-of-the-art standards and its user interface needs to be responsive and operate smoothly in any screen and browser. Using this tool the users will be able to create and manage their buildings, create and edit their objects and finally create their own new types of structures if needed. In more details, the editor tool should:

- Be compatible with state-of-the-art standards.
- Have responsive interface.
- Be able to be used not only by professionals but also for non experts.
- Have a multilingual interface in order to attract people of different nationalities and allow them to contribute in the tools building library.
- Support extensibility for making new structure through our interface.
- Allow the definition and management of new structures.
- Support the floor plan editing process by providing the means to easily create objects and structures for different types of building.
- Allow the creation and management of building and its containing objects.
- Provide search and filter functionality in order to allow easy finding of an already existing structure based on different characteristics.
- Provide a multimedia file library for uploading and managing media material in order to be used in a building creation.

4.2.3 Mobile Application

The end-user application needs to be a mobile application due to the fact that it should support the navigation inside a specific building. A problem with this requirement is the fact that different device vendors provide different frameworks for the development of applications compatible with their devices. Most of these are using completely different programming languages(e.g Java for Android, Objective C and Swift for iOS). 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 and thus, provide support for the majority of 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, not requiring the browser opening and the entering of the url. In addition, when the device is not connected to the internet, the use of 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 disadvantages[39], we use 'PhoneGap', which is the most popular among them and of course is Open Source and free.

The features of the end-user application should include:

- Compatibility with most state-of-the-art mobile devices/platforms.
- Beautiful and usable UI/UX, able to be used by everyone naive user.
- Support of real time browsing of available building and its objects.
- Ability to play, pause and load videos and sounds.
- Ability to open external links and resources within the application.
- Compatibility with our editor tool, in the sense that it should reproduce building and floor maps that have been generated by the web application.
- Ability to use the hardware tools (camera, compass etc.) of the mobile device.

4.2.4 Backend

The requirements that we have set for the backend of our system, serving both the editor tool and the mobile application, are summarized below:

- Compatible with state-of-the-art standards.
- Able to support the persistency of structures, buildings, floors and objects.
- Expose 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.
- Support data scalability.
- Support for the persistency of highly complex and structured data.
- Perform efficient serialization and de-serialization of the persisted information.

4.3 Use Cases

A use case describes the system's behavior under various conditions as the system responds to a request from one of the stakeholders, called primary actor. The primary actor initiates an interaction with the system to accomplish a goal. The system responds, protecting the interests of all the stakeholders.

This section describes the system's behavior under its interaction with the stakeholders. To this end, we exploit the method of use cases as they have been described by Alistair Cockburn [21]. For more understanding we present use cases clustered in form of use case diagrams. Figures 4.1, 4.2, 4.3 and 4.4 present the use cases of Editor Tool, also, figure 4.5 presents the use cases of Mobile Application.

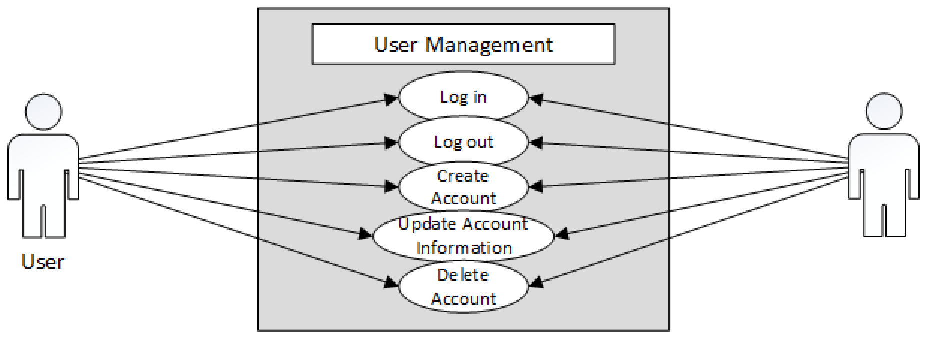


Figure 4.1: Editor Tool - User Management.

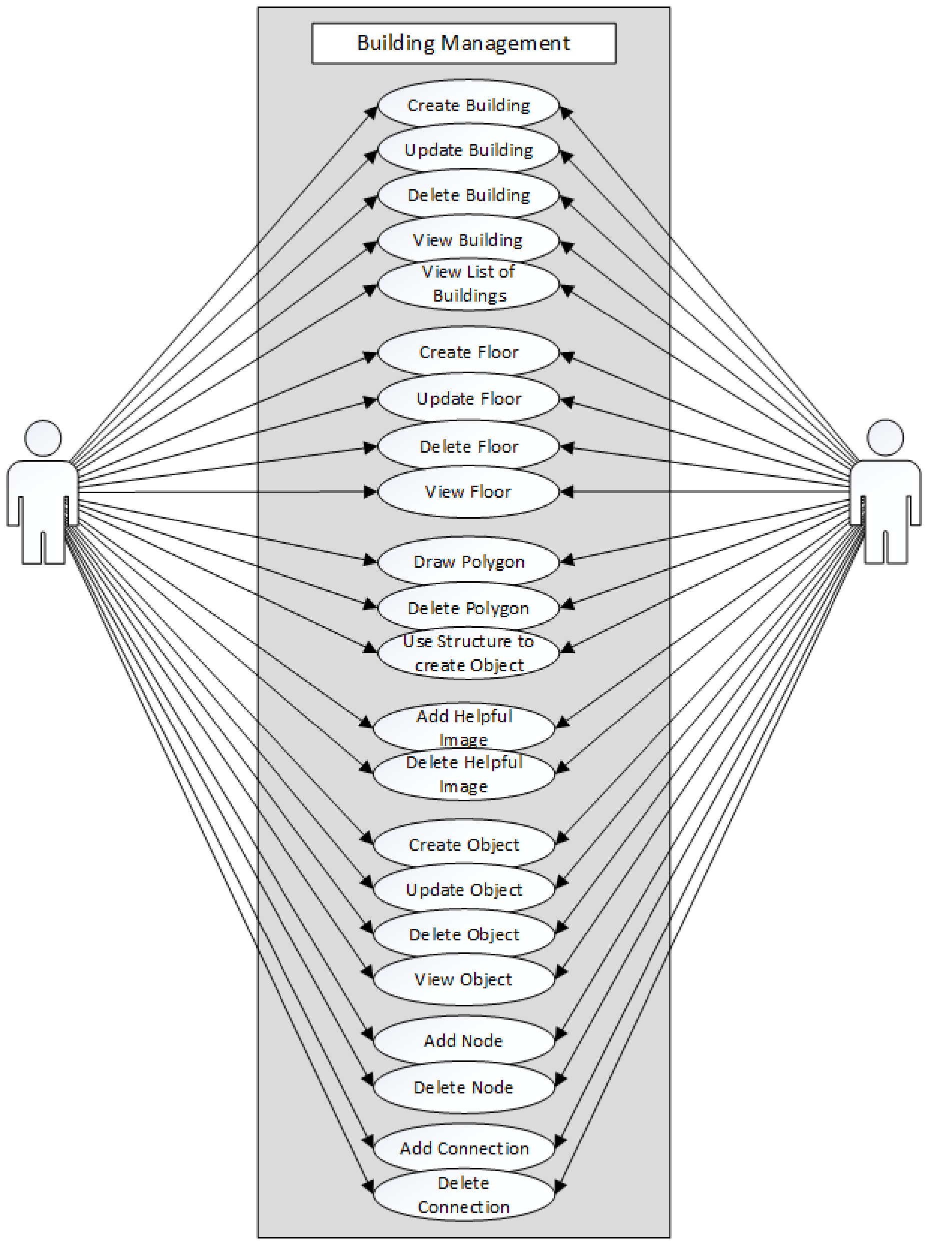


Figure 4.2: Editor Tool - Building Management.

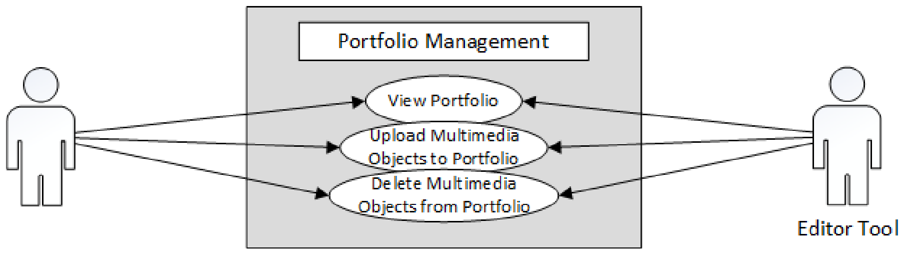

Portfolio Management
View Portfolio
Upload Multimedia Objects to Portfolio
Delete Multimedia Objects from Portfolio
Editor Tool

Figure 4.3: Editor Tool - Portfolio Management.

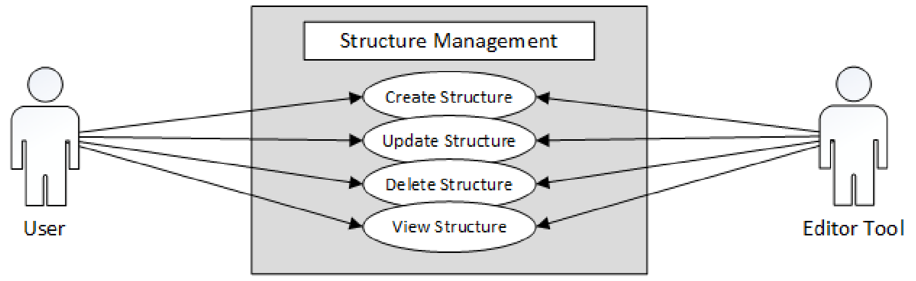


Figure 4.4: Editor Tool - Structure Management.

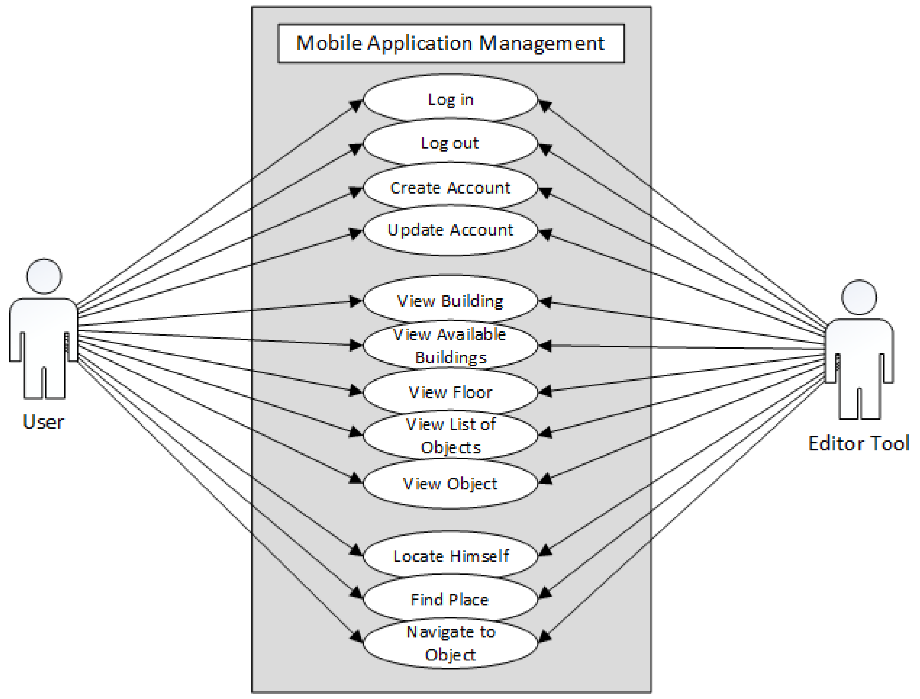


Figure 4.5: Mobile Application.

Table 4.1: Use case 1: "Log into the Editor Tool"

Use case 1: "Log into the Editor Tool"		
Context of Use	The user wants to be logged into the system in order to use the system's functionality.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user has already created successfully an account to the system.	
Success End Conditions	User logged into the system.	
Failed End Conditions	User could not log into the system.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user visits the application's url without being already logged in.	
Description	**Step**	**Action**
	1	The system displays the log in form.
	2	The 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.
Extensions	**Step**	**Branching Action**
	1	3a1. Submitted data are incorrect.
	2	3a2. The system displays the login form with the appropriate message.

Table 4.2: Use case 2: "Log into the Mobile Application"

Use case 2: "Log into the Mobile Application"		
Context of Use	The user wants to be logged into the Mobile Application in order to use the mobile functionality.	
Scope	System (Mobile Application)	
Level	Sub function	
Preconditions	The user has already an account to the system.	
Success End Conditions	User logged into the system.	
Failed End Conditions	User could not log into the system.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user opens the Mobile Application.	
Description	**Step**	**Action**
	1	The system displays the log in form.
	2	The 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.
Extensions	**Step**	**Branching Action**
	1	3a1. Submitted data are incorrect.
	2	3a2. The system displays the login form with the appropriate message.

Table 4.3: Use case 3: "Log out"

Use case 3: "Log out"		
Context of Use	The user wants to be logged out from the system.	
Scope	System (Editor Tool & Mobile Application)	
Level	Sub function	
Preconditions	The User is already logged in the system.	
Success End Conditions	The User is logged out successfully from the system.	
Failed End Conditions	The system could not log the user out of the system.	
Primary, Secondary Actors	User System (Editor Tool & Mobile Application)	
Trigger	The user selects to log out from the system.	
Description	**Step**	**Action**
	1	The system logs the User out of the system.
	2	The system displays the log in screen.
Extensions	**Step**	**Branching Action**

Table 4.4: Use case 4: "Create Account"

Use case 4: "Create Account"		
Context of Use	The user wants to create new account, so he can have full access to the system's functionality.	
Scope	System (Editor Tool & Mobile Application)	
Level	Sub function	
Preconditions	The user is not logged in the system.	
Success End Conditions	The user created a new account successfully.	
Failed End Conditions	The user could not create a new account.	
Primary, Secondary Actors	User System (Editor Tool & Mobile Application)	
Trigger	The user selects to create a new account in the system.	
Description	**Step**	**Action**
	1	The system displays the Sign Up form.
	2	The user fills in the required information about his new account.
	3	The system validates the user's input.
	4	The system creates a new user which is persisted in the database and displays the first screen to the user.
Extensions	**Step**	**Branching Action**
	1	3a1. Submitted data are incorrect.
	2	3a2. The system displays the sign up form.

Table 4.5: Use case 5: "Create Building"

Use case 5: "Create Building"		
Context of Use	The user wants to create a new building.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system.	
Success End Conditions	The user created a new building successfully.	
Failed End Conditions	The user could not create a new building.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to create a new building.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new building.
	2	The user fills the required information about the building and if required edits optional information about building.
	3	The system validates the user's input and creates a new building which is persisted in the database.
	4	The system presents the newly created building to the user.
Extensions	**Step**	**Branching Action**
	1	3a1. Submitted data are incorrect.
	2	3a2. The system displays the interface for the creation of the new building.

Table 4.6: Use case 6: "View existing Building"

Use case 6: "View existing Building"		
Context of Use	The user wants to view an existing building.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system and there are buildings persisted in database.	
Success End Conditions	The user has viewed a specific building.	
Failed End Conditions	The user could not view a building.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to view a building from existing.	
Description	**Step**	**Action**
	1	The user browses the list of buildings which are created by him.
	2	The user selects to view a building from the available buildings.
	3	The system presents the interface of building.
Extensions	**Step**	**Branching Action**

Table 4.7: Use case 7: "Update Building"

Use case 7: "Update Building"		
Context of Use	The user wants to update a building.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user has already created this building.	
Success End Conditions	The user updated a building successfully.	
Failed End Conditions	The user could not update a building.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to update a building.	
Description	**Step**	**Action**
	1	The user selects a building to be updated.
	2	The system presents the interface for the creation with filled in the information fields.
	3	The user edits the information about the building.
	4	The system validates the user's input and creates a new building which is persisted in the database.
	5	The system presents the newly updated building to the user.
Extensions	**Step**	**Branching Action**
	1	4a1. Submitted data are incorrect.
	2	4a2. The system presents the interface for the creation of the building with filled the information.

Table 4.8: Use case 8: "Delete Building"

Use case 8: "Delete Building"		
Context of Use	The user wants to delete a building.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user has already created this building.	
Success End Conditions	The user deleted a building successfully.	
Failed End Conditions	The user could not delete a building.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete a building.	
Description	**Step**	**Action**
	1	The user selects a building to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the building.
	4	The system deletes the building and all related information about this building (containing objects, multimedia, nodes etc.).
	5	The system presents the first screen to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the building.

Table 4.9: Use case 9: "Create Floor"

Use case 9: "Create Floor"		
Context of Use	The user wants to add floor in a building.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already.	
Success End Conditions	The user created a new floor successfully.	
Failed End Conditions	The user could not create a new floor.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to create a new floor.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new floor.
	2	The user fills the required information about the floor and if required edits optional information about floor.
	3	The system validates the user's input and creates a new floor which is persisted in the database.
	4	The system presents the newly created floor to the user.
Extensions	**Step**	**Branching Action**
	1	3a1. Submitted data ara incorrect.
	2	3a2. The system displays the interface for the creation of the new floor.

Table 4.10: Use case 10: "View existing Floor"

Use case 10: "View existing Floor"		
Context of Use	The user wants to view an existing floor.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system and there are floors of selected building. persisted in database.	
Success End Conditions	The user has viewed a specific floor.	
Failed End Conditions	The user could not view a floor.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to view a floor from existing.	
Description	**Step**	**Action**
	1	The user browses the list of floors of building which are created by him.
	2	The user selects to view a floor from the available floors.
	3	The system presents the interface of floor.
Extensions	**Step**	**Branching Action**

Table 4.11: Use case 11: "Update Floor"

Use case 11: "Update Floor"		
Context of Use	The user wants to update a floor.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user has already created this floor.	
Success End Conditions	The user updated a floor successfully.	
Failed End Conditions	The user could not update a floor.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to update a floor.	
Description	**Step**	**Action**
	1	The user selects a floor to be updated.
	2	The system presents the interface for the creation with filled in the information fields.
	3	The user edits the information about the floor.
	4	The system validates the user's input and updated this floor which is persisted in the database.
	5	The system presents the newly updated floor to the user.
Extensions	**Step**	**Branching Action**
	1	4a1. Submitted data are incorrect.
	2	4a2. The system presents the interface for the creation of the floor with filled the information.

Table 4.12: Use case 12: "Delete Floor"

Use case 12: "Delete Floor"		
Context of Use	The user wants to delete a floor.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user has already created this floor.	
Success End Conditions	The user deleted a floor successfully.	
Failed End Conditions	The user could not delete a floor.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete a floor.	
Description	**Step**	**Action**
	1	The user selects a floor to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the floor.
	4	The system deletes the floor and all related information about this floor (containing objects, multimedia, nodes etc.).
	5	The system presents the main building interface to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the main building interface to the user.

Table 4.13: Use case 13: "Draw Polygon"

Use case 13: "Draw Polygon"		
Context of Use	The user wants to draw a polygon on a floor plan.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user drew a new polygon successfully.	
Failed End Conditions	The user could not draw a new polygon.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to draw a new polygon.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new polygon.
	2	The user draw the appropriate polygon about the composition of the floor plan.
	3	The system persists the polygon in the database.
	4	The system presents the newly created floor plan to the user.
Extensions	**Step**	**Branching Action**

Table 4.14: Use case 14: "Delete Polygon"

Use case 14: "Delete Polygon"		
Context of Use	The user wants to delete a polygon from a floor plan	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user deleted a polygon successfully.	
Failed End Conditions	The user could not delete a polygon.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to draw a new polygon.	
Description	**Step**	**Action**
	1	The user selects a polygon to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the polygon.
	4	The system deletes the polygon.
	5	The system presents the newly floor plan interface to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the floor plan interface to the user.

Table 4.15: Use case 15: "Add Node"

Use case 15: "Add Node"		
Context of Use	The user wants to add a node on a floor plan.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user added a new node successfully.	
Failed End Conditions	The user could not add a new node.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to add a new node.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new node.
	2	The user add the appropriate node about the composition of the graph of floor plan.
	3	The system persists the node in the database.
	4	The system presents the newly created graph on floor plan to the user.
Extensions	**Step**	**Branching Action**

Table 4.16: Use case 16: "Update Node"

Use case 16: "Update Node"		
Context of Use	The user wants to update a node.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user has already created this node.	
Success End Conditions	The user updated a node successfully.	
Failed End Conditions	The user could not update a node.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to update a floor.	
Description	**Step**	**Action**
	1	The user selects a node to be updated.
	2	The system presents the interface for the creation with filled in the information fields.
	3	The user edits the information about the node.
	4	The system validates the user's input and updated this node which is persisted in the database.
	5	The system presents the newly updated node to the user.
Extensions	**Step**	**Branching Action**
	1	4a1. Submitted data are incorrect.
	2	4a2. The system presents the interface for the creation of the node with filled the information.

Table 4.17: Use case 17: "Delete Node"

Use case 16: "Delete Node"		
Context of Use	The user wants to delete a node.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user deleted a node successfully.	
Failed End Conditions	The user could not delete a node.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete a node.	
Description	**Step**	**Action**
	1	The user selects a node to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the node.
	4	The system deletes the node.
	5	The system presents the newly floor plan interface to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the floor plan interface to the user.

Table 4.18: Use case 18: "Add Connection"

Use case 18: "Add Connection"		
Context of Use	The user wants to connect two nodes on a floor plan.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user add a new connection successfully.	
Failed End Conditions	The user could not add a new connection.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to add a new connection.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new connection.
	2	The user add the appropriate connection about the composition of the graph of floor plan.
	3	The system persists the connection in the database.
	4	The system presents the newly created graph on floor plan to the user.
Extensions	**Step**	**Branching Action**

Table 4.19: Use case 19: "Delete Connection"

Use case 19: "Delete Connection"		
Context of Use	The user wants to delete a connection.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user deleted a connection successfully.	
Failed End Conditions	The user could not delete a connection.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete a connection.	
Description	**Step**	**Action**
	1	The user selects a connection to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to connection the node.
	4	The system deletes the connection.
	5	The system presents the newly floor plan interface to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the floor plan interface to the user.

Table 4.20: Use case 20: "Add Helpful Image"

Use case 20: "Add Helpful Image"		
Context of Use	The user wants to add a helpful image on a floor plan.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user added a new helpful image successfully.	
Failed End Conditions	The user could not add a new helpful image.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to add a new helpful image.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new helpful image.
	2	The user add the appropriate helpful image about the composition of floor plan.
	3	The system persists the helpful image in the database.
Extensions	**Step**	**Branching Action**

Table 4.21: Use case 21: "Delete Helpful Image"

Use case 21: "Delete Helpful Image"		
Context of Use	The user wants to delete a helpful image.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user deleted a helpful image successfully.	
Failed End Conditions	The user could not delete a helpful image.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete a node.	
Description	**Step**	**Action**
	1	The user selects a helpful image to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the helpful image.
	4	The system deletes the helpful image.
	5	The system presents the newly floor plan interface to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the floor plan interface to the user.

Table 4.22: Use case 22: "Create Object"

Use case 22: "Create Object"		
Context of Use	The user wants to create a new object.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user created a new object successfully.	
Failed End Conditions	The user could not create a new object.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to create a new object.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new object.
	2	The user fills the required information about the object and if required edits optional information about object.
	3	The system validates the user's input and creates a new object which is persisted in the database.
	4	The system presents the newly created object to the user.
Extensions	**Step**	**Branching Action**
	1	3a1. Submitted data are incorrect.
	2	3a2. The system displays the interface for the creation of the new object.

Table 4.23: Use case 23: "View existing Object"

Use case 23: "View existing Object"		
Context of Use	The user wants to view an existing object.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system and here are objects persisted in database.	
Success End Conditions	The user has viewed a specific object.	
Failed End Conditions	The user could not view an object.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to view an object from existing.	
Description	**Step**	**Action**
	1	The user browses the list of objects which are created by him.
	2	The user selects to view an object from the available objects.
	3	The system presents the interface of object.
Extensions	**Step**	**Branching Action**

Table 4.24: Use case 24: "Update Object"

Use case 24: "Update Object"		
Context of Use	The user wants to update an object.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user has already created this object.	
Success End Conditions	The user updated an object successfully.	
Failed End Conditions	The user could not update an object.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to update an object.	
Description	**Step**	**Action**
	1	The user selects an object to be updated.
	2	The system presents the interface for the creation with filled in the information fields.
	3	The user edits the information about the object.
	4	The system validates the user's input and updated this object which is persisted in the database.
	5	The system presents the newly updated object to the user.
Extensions	**Step**	**Branching Action**
	1	4a1. Submitted data are incorrect.
	2	4a2. The system presents the interface for the creation of the object with filled the information.

Table 4.25: Use case 25: "Delete Object"

Use case 25: "Delete Object"		
Context of Use	The user wants to delete an object.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user has already created this object.	
Success End Conditions	The user deleted an object successfully.	
Failed End Conditions	The user could not delete an object.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete an object.	
Description	**Step**	**Action**
	1	The user selects an object to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the object.
	4	The system deletes the object and all related information about this object (containing objects, multimedia, nodes etc.).
	5	The system presents the main building interface to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the main building interface to the user.

Table 4.26: Use case 26: "View Portfolio"

Use case 26: "View Portfolio"		
Context of Use	The user wants to view portfolio.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system.	
Success End Conditions	The user has viewed his portfolio successfully.	
Failed End Conditions	The user could not view portfolio.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to view portfolio.	
Description	**Step**	**Action**
	1	The system presents the interface of portfolio (images).
	2	The user can select to view videos or audios in portfolio.
Extensions	**Step**	**Branching Action**

Table 4.27: Use case 27: "Upload multimedia files to Portfolio"

Use case 27: "Upload multimedia files to Portfolio"		
Context of Use	The user wants to upload multimedia files to portfolio.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system.	
Success End Conditions	The user uploaded multimedia file in portfolio successfully.	
Failed End Conditions	The user could not upload multimedia to portfolio successfully.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to upload multimedia files to portfolio.	
Description	**Step**	**Action**
	1	The system presents the interface of selection multimedia files from desktop.
	2	The user selects a multimedia file.
	3	The system presents progress bar about upload.
	4	The system presents the correct status.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the selection.
	2	2a2. The system presents the interface of portfolio.
	3	4a1. The multimedia file could not uploaded successfully.
	4	4a2. The system presents the interface of portfolio.

Table 4.28: Use case 28: "Delete Multimedia File"

Use case 28: "Delete Multimedia File"		
Context of Use	The user wants to delete a multimedia file.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user has already uploaded multimedia files.	
Success End Conditions	The user deleted a multimedia file successfully.	
Failed End Conditions	The user could not delete a multimedia file.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete a multimedia file.	
Description	**Step**	**Action**
	1	The user selects to delete a specific multimedia file.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the multimedia file.
	4	The system deletes the multimedia file successfully.
	5	The system presents the portfolio interface to the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents portfolio interface.

Table 4.29: Use case 29: "Create Structure"

Use case 29: "Create Structure"		
Context of Use	The user wants to create a new structure.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user is already logged in the system.	
Success End Conditions	The user created a new structure successfully.	
Failed End Conditions	The user could not create a new structure.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to create a new structure.	
Description	**Step**	**Action**
	1	The system presents the interface for the creation of the new structure.
	2	The user fills the required information about the structure and if required edits optional information about structure.
	3	The system validates the user's input and creates a new structure which is persisted in the database.
	4	The system presents the newly created structure to the user.
Extensions	**Step**	**Branching Action**
	1	3a1. Submitted data are incorrect.
	2	3a2. The system displays the interface for the creation of the new structure.

Table 4.30: Use case 30: "View existing Structure"

Use case 30: "View existing Structure"		
Context of Use	The user wants to view an existing structure.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user is already logged in the system and there are structures persisted in database.	
Success End Conditions	The user has viewed a specific structure.	
Failed End Conditions	The user could not view a structure.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to view a structure from existing.	
Description	**Step**	**Action**
	1	The user browses the list of structures which are persisted in database.
	2	The user selects to view a structure from the available structures.
	3	The system presents the interface of structure.
Extensions	**Step**	**Branching Action**

Table 4.31: Use case 31: " Update Structure"

Use case 31: " Update Structure"		
Context of Use	The user wants to update a structure.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user has already created this structure.	
Success End Conditions	The user updated a structure successfully.	
Failed End Conditions	The user could not update a structure.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to update a structure.	
Description	**Step**	**Action**
	1	The user selects an structure to be updated.
	2	The system presents the interface for the creation with filled in the information fields.
	3	The user edits the information about the structure.
	4	The system validates the user's input and updated this structure which is persisted in the database.
	5	The system presents the newly updated structure to the user.
Extensions	**Step**	**Branching Action**
	1	4a1. Submitted data are incorrect.
	2	4a2. The system presents the interface for the creation of the structure with filled the information.

Table 4.32: Use case 32: "Delete Structure"

Use case 32: "Delete Structure"		
Context of Use	The user wants to delete a structure.	
Scope	System (Editor Tool)	
Level	Summary	
Preconditions	The user has already created this structure.	
Success End Conditions	The user deleted a structure successfully.	
Failed End Conditions	The user could not delete a structure.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to delete a structure.	
Description	**Step**	**Action**
	1	The user selects a structure to be deleted.
	2	The system asks the user to confirm the action.
	3	The user selects to delete the structure.
	4	The system deletes the object and all related information about this structure (sub-structures, structure's type objects, etc.).
Extensions	**Step**	**Branching Action**
	1	2a1. The user selects to cancel the deletion.
	2	2a2. The system presents the structure.

Table 4.33: Use case 33: "Use structure to Create Object"

Use case 33: "Use structure to Create Object"		
Context of Use	The user wants to use structure to create object.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user is already logged in the system, and has created a building already and added successfully a floor.	
Success End Conditions	The user selected a structure to create an object successfully.	
Failed End Conditions	The user could not select structure to create object.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to create structure to create object.	
Description	**Step**	**Action**
	1	The system presents the interface of available structure.
	2	The user selects a structure
	3	The user selects to use this structure to create object.
Extensions	**Step**	**Branching Action**

Table 4.34: Use case 34: "Update Account Information"

Use case 34: "Update Account Information"		
Context of Use	The user wants to update account information.	
Scope	System (Editor Tool)	
Level	Sub function	
Preconditions	The user is already logged in the system.	
Success End Conditions	The user has successfully updated account information.	
Failed End Conditions	The user could not successfully update account information.	
Primary, Secondary Actors	User System (Editor Tool)	
Trigger	The user selects to update account information.	
Description	**Step**	**Action**
	1	The user selects to view profile.
	2	The system presents account information.
	3	The user edited account information.
	4	The system validates the user's input and present the newly updated account information.
Extensions	**Step**	**Branching Action**

Table 4.35: Use case 35: "View available Buildings"

Use case 35: "View available Buildings"		
Context of Use	The user wants view the available buildings.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system.	
Success End Conditions	The user viewed available buildings successfully.	
Failed End Conditions	The user could not view available buildings successfully.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to view available buildings.	
Description	**Step**	**Action**
	1	The system presents the available buildings.
Extensions	**Step**	**Branching Action**

Table 4.36: Use case 36: "View Building"

Use case 36: "View Building"		
Context of Use	The user wants to view a specific building.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system and to view available buildings.	
Success End Conditions	The system presents the selected building.	
Failed End Conditions	The system could not present the selected building.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to view a specific building.	
Description	**Step**	**Action**
	1	The system presents the available buildings.
	2	The user selects to view a specific building.
	3	The system present the selected building.
Extensions	**Step**	**Branching Action**

Table 4.37: Use case 37: "View Floors of Building"

Use case 37: "View Floors of Building"		
Context of Use	The user wants to view floors of building.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system and to view a specific building.	
Success End Conditions	The system presents the floors of specific building.	
Failed End Conditions	The system could not present the floors of specific building.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to view the floors of specific building.	
Description	**Step**	**Action**
	1	The system presents the floors of a specific building.
	2	The user selects a floor.
	3	The system presents the information of a selected floor.
Extensions	**Step**	**Branching Action**

Table 4.38: Use case 38: "View Objects of Floor"

Use case 38: "View Objects of Floor"		
Context of Use	The user wants view objects of floor.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system and to view a specific floor.	
Success End Conditions	The system presents the objects of specific floor.	
Failed End Conditions	The system could not present the objects of specific floor.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to view the objects of specific floor.	
Description	**Step**	**Action**
	1	The system presents the objects of a specific floor.
	2	The user selects an object.
	3	The system presents the information of a selected object.
Extensions	**Step**	**Branching Action**

Table 4.39: Use case 39: "View Objects of Building"

Use case 39: "View Objects of Building"		
Context of Use	The user wants view objects of floor.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system and to view a specific building.	
Success End Conditions	The system presents the objects of specific building.	
Failed End Conditions	The system could not present the objects of specific building.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to view the objects of specific building.	
Description	**Step**	**Action**
	1	The system presents the objects of a specific building.
	2	The user selects an object.
	3	The system presents the information of a selected object.
Extensions	**Step**	**Branching Action**

Table 4.40: Use case 40: "User locates himself"

Use case 40: "User locates himself"		
Context of Use	The user wants to locate himself inside the building.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system and is able to scan a QR code or type a set of keywords.	
Success End Conditions	The user locates himself successfully.	
Failed End Conditions	The user could not locate himself.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to locate himself and scan a QR code or type a set of keywords.	
Description	**Step**	**Action**
	1	The system presents the appropriate UI to the user and gives him the ability to use the camera of his mobile to scan a QR code.
	2	The user scan a QR code.
	3	The systems presents a floor map which contains the highlighted position of the user.
Extensions	**Step**	**Branching Action**
	1	2a1. The user scans an invalid QR code.
	2	2a2. The system inform the user and use the camera to capture another QR code.
	3	2a1. The user types a set of keywords.

Table 4.41: Use case 41: "User scans QR Code"

Use case 41: "User scans QR Code"		
Context of Use	The user wants to scan a QR code inside the building.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system and is able to scan a QR code.	
Success End Conditions	The user retrieve the object that is connected with QR code.	
Failed End Conditions	The user could not retrieve the object that is connected with QR code.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to scan a QR code.	
Description	**Step**	**Action**
	1	The system presents the appropriate UI to the user and gives him the ability to use the camera of his mobile to scan a QR code.
	2	The user scan a QR code.
	3	The systems presents the appropriate object that is connected with scanned QR code.
Extensions	**Step**	**Branching Action**
	1	2a1. The user scans an invalid QR code.
	2	2a2. The system inform the user and use the camera to capture another QR code.

Table 4.42: Use case 42: "Find a Place"

Use case 42: "Find a Place"		
Context of Use	The user wants to find a place inside the building.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system.	
Success End Conditions	The system presents the searched object.	
Failed End Conditions	The system could not present the searched object.	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to find a place.	
Description	**Step**	**Action**
	1	The user type the desired keywords.
	2	The system presents the retrieved results which are matched with the keywords.
	3	The user selects the desired object from the results.
	4	The system presents the selected object.
Extensions	**Step**	**Branching Action**
	1	2a1. The keywords do not match at an existing object.
	2	2a2. The system informs the user for non-existent results.

Table 4.43: Use case 43: "Navigate to Object"

Use case 43: "Navigate to Object"		
Context of Use	The user wants to navigate to an object.	
Scope	System (Mobile Application)	
Level	Summary	
Preconditions	The user is already logged in the system and is able to select to navigate to an existing object.	
Success End Conditions	The system presents the appropriate sequence of commands to the user.	
Failed End Conditions	The system could not present the appropriate sequence of commands to the user	
Primary, Secondary Actors	User System (Mobile Application)	
Trigger	The user selects to navigate to a specific object.	
Description	**Step**	**Action**
	1	The system highlights the position of the user and the positions of the user's destination.
	2	The system presents the suitable instructions to the user in order to navigate him to his destination.
	3	The user selects the current command and follow the instruction.
Extensions	**Step**	**Branching Action**
	1	2a1. There is not a path in order to the user reach to his destination.
	2	2a2. The system informs the user for non-existent path.

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 floor plans and their containing objects. 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 5

Model

This chapter presents the model that has been developed to support, among others, the representation of buildings, floors, objects, structures, nodes etc. Section 5.1 describes the model's basic notions and structure, while Section 5.2 provides some model implementation examples in order to acquire deeper understanding of the model's capabilities.

5.1 Specification

The model developed and supported by our system has been designed with extensibility in mind, so as to be able to describe as many as possible different types of buildings. To do this, we identified that a building should be described as a sequence of building objectives (e.g floor and objects) and each of these objectives should have the potential to be further differentiated with specific attributes or characteristics. Figure 5.1 presents the identified model entities in the form of a class diagram.

User

The User class represents the users of our system and includes both building creators and simple users, differentiating based on their role. The attributes that each user has are those presented below:

- *lastSeen*: the last date that the user logged in to our system.
- *id*: the user's unique identifier.
- *firstName*: the user's first name.
- *lastName*: the user's last name.

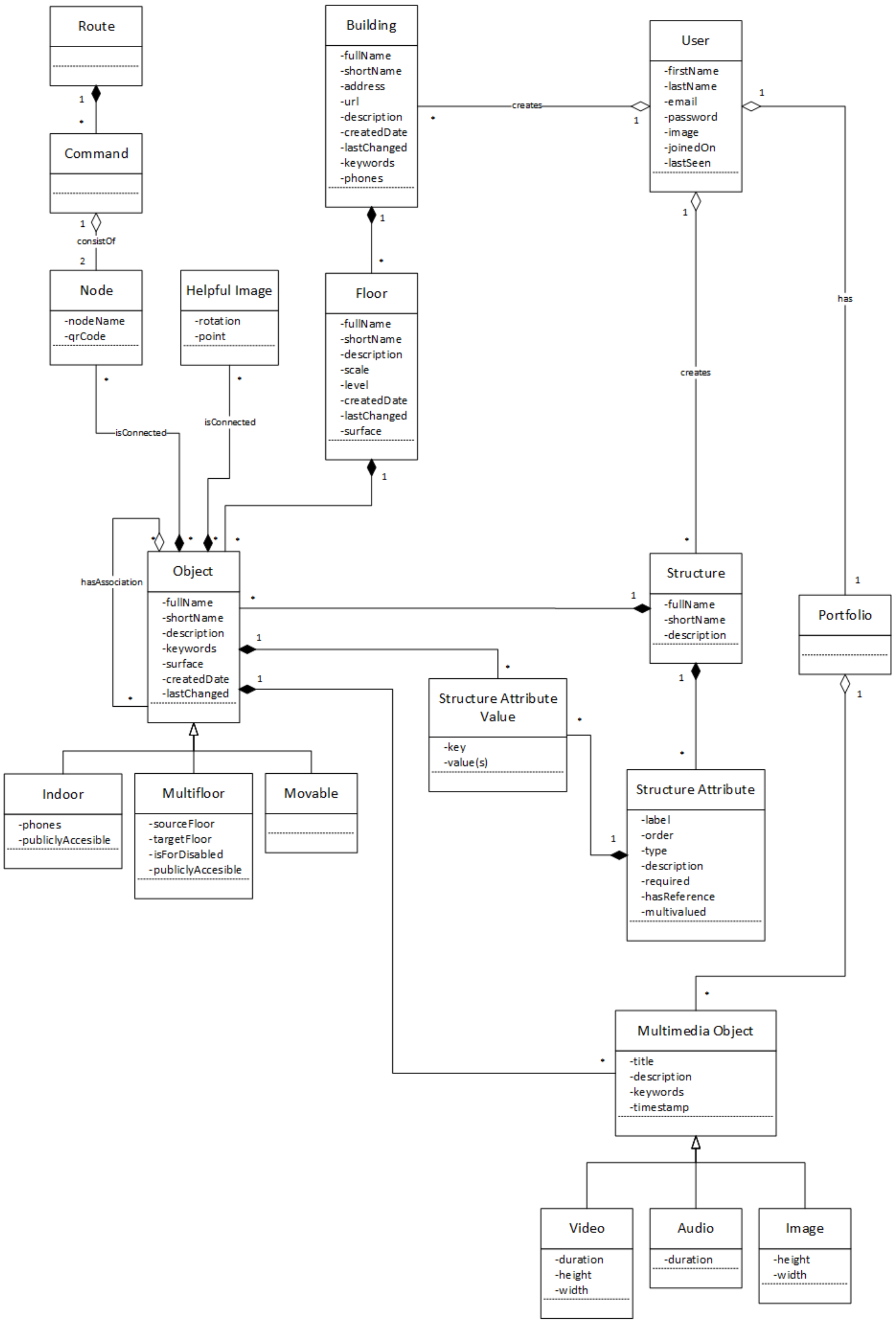


Figure 5.1: Model.

- *email*: the user's email address
- *password*: the password used for logging in the system.
- *image*: a profile picture of the user.
- *joinedOn*: the date on which the user signed up to our application.
- *lastSeen*: the last date that the user logged in to our system.

Building

The Building class represents the buildings which can be created and managed by our system. It can be created by users in order to set up a building, while it can also be used for the navigation of simple user inside it. It is consisted of a sequence of floors which contain objects. Since buildings can be bounded to a specific location, they can also be used for building searching and filtering. The attributes comprising a building are the following:

- *id*: the building's unique identifier.
- *fullName*: the building's full name.
- *shortName*: the building's short name.
- *address*: the building's address
- *url*: the building's url.
- *description*: the building's description.
- *createdDate*: the date on which the building is been created.
- *lastChanged*: the last date that the user changed information about the building.

Floor

The Floor class represents the floors which compose a building. They can be created by building creators in order to set up a building. Each floor has its own borders which are defined by the building creator and is consisted of a sequence of containing objects. A floor may be described using the following attributes:

- *id*: the floor's unique identifier.
- *fullName*: the floor's full name.
- *shortName*: the floor's short name.

- *level*: the floor's level in building.
- *scale*: the floor's map scale depends on the user selection compared with editor surface.
- *description*: the floor's description.
- *createdDate*: the date on which the floor is been created.
- *lastChanged*: the last date that the user changed information about the floor.

Object

The Object class represents a floor objective. In essence, an object is an instance of predefined structure. Each of object has a structure and take the appropriate attributes from it. In other words, all of building's containing objects refer to Object class. The main attributes of an object are the following:

- *id*: the object's unique identifier.
- *fullName*: the object's full name.
- *shortName*: the object's short name.
- *description*: the object's description.
- *surface*: the object's overall surface.
- *createdDate*: the date on which the object is been created.
- *lastChanged*: the last date that the user changed information about the object.

Indoor Object

The Indoor Object class represents the "Object" which is referred to an indoor location object. This object may be a course room, a conference room, an exhibition room etc. Additionally this type of object is characterized by its accessibility(e.g if is a privately accessible or publicly accessible).

Multifloor Object

The Multifloor Object class represents the "Object" which is referred to a multifloor location object. This object may be an elevator, stairs etc. Additionally this type of object is characterized by its accessibility(e.g if is a privately accessible or publicly accessible), by source and target floor, which are the connected floors, and whether this transition is for disabled people or not.

Movable Object

The Movable Object class represents the "Object" which is referred to an movable object. This object may be an exhibit, such as statue, painting, musical instrument, aquarium etc. Additionally, this type of object is characterized by its relation to other objects. In other words, a movable object may be near to another movable object or inside an object. To be more specific, relations such as is below, above, near, opposite, north, etc are some of possible choices in order to be used for the better navigation to the simple user.

Structure

The Structure class represents the structures which can be created, defined and managed by our system. It is comprised of several descriptive and presentation meta-data. Additionally, it consists of a sequence of attributes that each of them represents the desired attribute of user in order to create a new primitive structure depending on his needs. The Structure consists of the following attributes:

- *id*: the structure's identifier.
- *fullName*: the structure's full name.
- *shortName*: the structure's short name.
- *description*: the structure's description.
- *createdDate*: the date on which the structure is been created.

Structure Attribute

The Structure Attribute class represents a structure objective. Each structure attribute describes a specific characteristic which user wants to include in his created structure. A structure attribute consists of the following attributes:

- *id*: the structure attribute's unique identifier.
- *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 the 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.
- *required*: defines if the attribute must be populated in the instantiation process.
- *hasReference*: defines if the attribute has reference to another object.
- *createdDate*: the date on which the object is been created.

Structure Attribute Value

The Structure Attribute Value class represents the instantiation of a Structure Attribute. In essence, this object contains key - value(s) pairs in order to set values at the desired attributes.

Route

The Route class represents the total set of instructions that are given to the simple user in order to navigate him inside the building. A route can contain multiple destinations but contain only one origin. In other words, a route consists of a sequence of instructions that represent certain actions in order for the user to arrive at his destination. The attributes comprising a route are the following:

- *id*: the route's identifier.
- *origin*: the route's origin position.
- *intermidiatePosition*: the route's intermediate positions.
- *destination*: the route's destination position.

Command

The Command class represents a sentence which is given to the simple user in order to navigate him from one position to another. Each command has one origin node and one destination node which is connected to each other. Additionally these commands are dynamically generated from our system depending on the user selection. The main attributes of a command are the following:

- *id*: the command's identifier.
- *origin*: the command's origin and the position of the user's device.
- *destination*: the command's destination and the next position of the simple user.
- *associatedObject*: the command's associated object is the object which the user moves inside.

- *id*verb: the command's verb which informs the user about the action that is needed.
- *content*: the command's content which construct a completed sentence.
- *curveNumber*: the command's curve number is the number of curve that the user may need to turn in order to arrive at his destination.
- *distance*: the command's distance is the distance between the origin and the destination of command.
- *degrees*: the command's degrees is the degrees of possible curve which is included in command.
- *direction*: the command's direction is the direction of the curve in order to inform the user about the direction of potential curve.

Node

The Node class represents a specific position on floor map in order to set it there a QR code. Each node is connected with a list of nodes composing a graph of paths. Furthermore, each node is connected with a list of objects so as the position of these objects can be retrieved by simple user. The Node consists of the following attributes:

- *id*: the node's unique identifier.
- *nodeName*: the node's name.
- *point*: the node's position on a floor map.

Useful Image

The Useful Image class represents a specific position on floor map in order to set it there a Helpful Image. Each helpful image has specific position and direction in order to give to the user more information about the content that is around. Furthermore, each node is connected with a list of objects. The Useful Image consists of the following attributes:

- *id*: the node's unique identifier.
- *rotation*: the useful image's rotation.
- *point*: the useful image's position on a floor map.

Multimedia Object

The Multimedia Object class represents multimedia objects of type: video, image, text and audio. Each type of Multimedia Object is depicted as a different class, holding its own descriptive attributes.

Audio

The Audio class represents the multimedia objects of type sound. Optionally, it contains a GPS point specifying the location that has been recorded.

Image

The Image class represents the multimedia objects that are of type image. Optionally, it contains a GPS point with information about the location where it was captured, as well as other descriptive metadata.

Video

The Video class represents the multimedia objects that are of type video. Optionally, it contains a list of GPS points with information about the location where it was recorded, as well as any other information provided by the camera. The list of GPS points can be used to recreate the path that the user took for the duration of the capturing.

Portfolio

The Portfolio class represents the multimedia library which has a user. These multimedia files will be used on creation buildings, floors or objects.

5.2 Model Examples

As already mentioned, our model has been created with flexibility and extensibility in mind in order to support a wide variety of buildings. To this end, our application provides a series of basic building blocks, supported by our model, so as assist a user in the creation of either simple or a highly complex and structured building. This section presents some examples of buildings, structures and objects that can be directly supported by our model and application.

Museum

A museum is a more representative example of building that our application can be applied. In case of museum, the museum director should be able to create and design the appropriate environment in order to give a useful content to the visitor of museum. The first step for the director of the museum is to create the desired structures according to building's needs. In more details, the user is able to create the indoor objects which are contained inside the museum. These indoor objects may contain an exhibit hall, a souvenir shop, a number of water closet or a cafeteria. For example an exhibit hall type may be characterized by the subject of containing objects or the number of containing objects in order to give further information to the visitor. Simultaneously, a museum has a wide variety of exhibits, such as statues, paintings etc. each of those has its own attributes. For example a statue characterized by its owner or its material, while a painting may be oil painting or pastel painting. On the other side a visitor of museum should be able to search and retrieve information about the exhibits and the halls of museum and of course retrieve its positions. As in all cases the retrieved data based on the basic information and the list of attributes of each object.

University

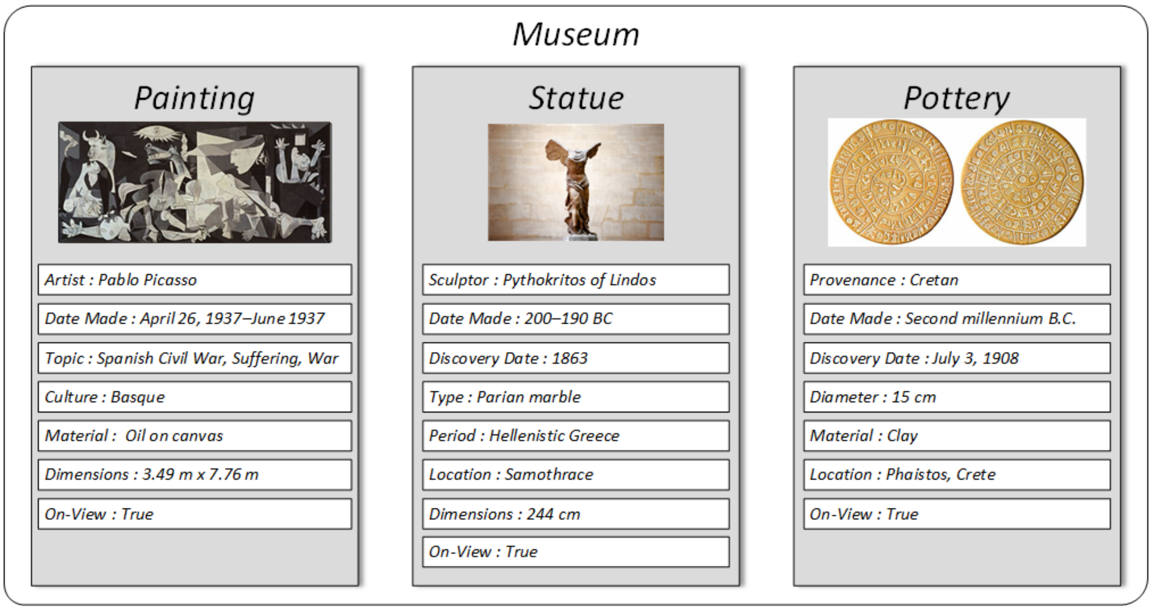
Another building that can be supported from our application is a university building. In this case, the university staff member should be able to create and design the appropriate structures that are contained inside the university building. Structures such as courses room, secretary offices, professor offices are some of the desired objects in case of university. More specifically, it is so important for a professor to know how many students fit in a course room, if a course room provides a projector or other useful equipment or if it is appropriate for a presentation. On the other side in many cases, especially in huge buildings such as universities, the students need to know where the secretary's office, a lab or a professor office are and navigate to those with simple instructions.

Library

One type of building that we target through our application is the library or similar kind of buildings. In this case, a library member is able to create and design the desired structures that are contained inside the Library. More specifically, it is very useful for a visitor of library to search and retrieve the library's content (books, journals, cds, dvds, etc.). All of these different structures have its own different attributes. In more details, a book is characterised by his author, publisher, pages, language etc, or a DVD has duration, language, subject etc. Furthermore, a student needs to know where he can look for educational material (books, dvds, journals, etc.) with specific characteristic in order to find the desired information.

In the following figures we present some of objects, that are supported by our system, and theirs attributes in order to provide analytical description about the capabilities of our application. More specifically, Figure 5.2 presents a subset of objects that are contained in the museum building, Figure 5.3 presents the objects that may be contained in University building and Figure 5.4 presents some of objects that a user is able to design in case of library.

Figure 5.2 presents a set of objects that can be created through our application in the context of Museum. In more details, we present a painting with a list of attributes, such as Artist, Date Made, Material etc, a Statue accompanied by its sculptor, discovery date, discovery location etc, and a Pottery which have been discovered at Phaistos.



Artist : Pablo Picasso
Date Made : April 26, 1937–June 1937
Topic : Spanish Civil War, Suffering, War
Culture : Basque
Material : Oil on canvas
Dimensions : 3.49 m x 7.76 m
On-View : True

Sculptor : Pythokritos of Lindos
Date Made : 200–190 BC
Discovery Date : 1863
Type : Parian marble
Period : Hellenistic Greece
Location : Samothrace
Dimensions : 244 cm
On-View : True

Provenance : Cretan
Date Made : Second millennium B.C.
Discovery Date : July 3, 1908
Diameter : 15 cm
Material : Clay
Location : Phaistos, Crete
On-View : True

Figure 5.2: Model Example - Museum.

Figure 5.3 presents a set of objects that can be located in a University building. These may be a Course Room, a Laboratory, a Professor Office, etc. In the following figure 5.3 is presented this set of object. More specifically, we present a course room with its capacity, sector, provision of projector, etc, a laboratory and a professor office.

Figure 5.4 presents a very representative example of building that can be supported by our system. In this figure present a set of object that are useful in case of Library. More specifically, objects

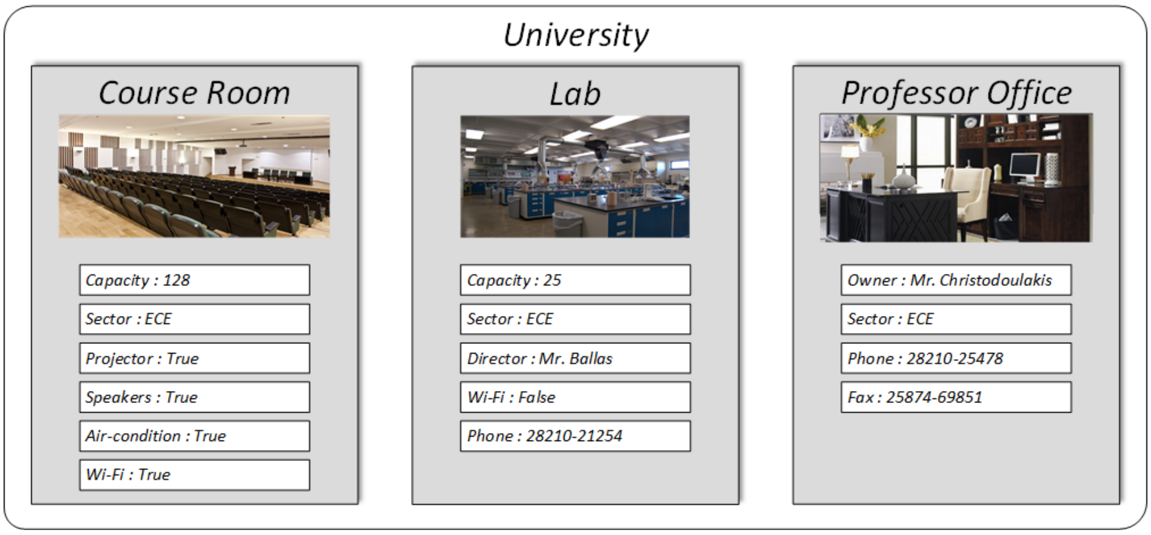


Figure 5.3: Model Example - University.

such as books, dvds and journal are very commons inside a library. All of these objects have its own attributes.

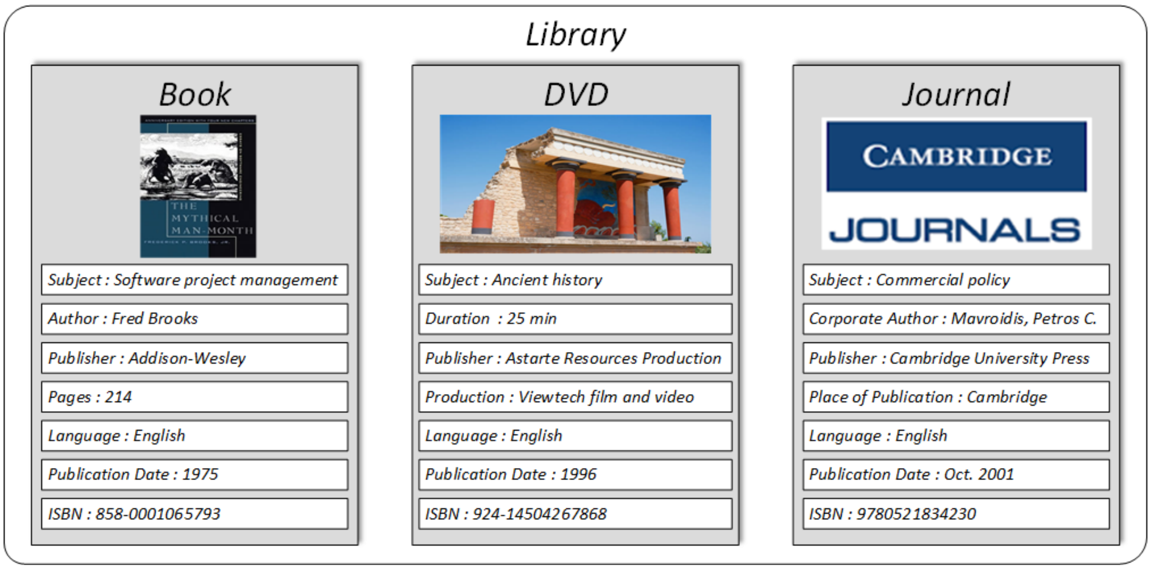

Library
Book
THE MYTHICAL MAN-MONTH
Subject : Software project management
Author : Fred Brooks
Publisher : Addison-Wesley
Pages : 214
Language : English
Publication Date : 1975
ISBN : 858-0001065793
DVD
Subject : Ancient history
Duration : 25 min
Publisher : Astarte Resources Production
Production : Viewtech film and video
Language : English
Publication Date : 1996
ISBN : 924-14504267868
Journal
CAMBRIDGE JOURNALS
Subject : Commercial policy
Corporate Author : Mavroidis, Petros C.
Publisher : Cambridge University Press
Place of Publication : Cambridge
Language : English
Publication Date : Oct. 2001
ISBN : 9780521834230

Figure 5.4: Model Example - Library.

5.3 Navigation Capabilities

In this section we present the navigation capabilities that be offered from our application. In more detail, we analyze the algorithm that we used for shortest path computation and the language that we defined in order to provide a well-defined set of instructions for navigation purposes.

5.3.1 Shortest Path Calculation

Shortest Path calculation is a very common topic when we need to implement navigation systems. The problem of finding the shortest path between two intersections on a road map, may be modelled by a special case of the shortest path problem in graphs. One of the most known algorithm to solve this kind of problem has been conceived by the Dutch computer scientist Edsger Dijkstra and bears the name of Dijkstra algorithm.

The latter can be applied to real navigation systems as follows: Suppose you would like to find the shortest path between two intersections on a city map, a starting point and a destination. Dijkstra's algorithm initially marks the distance, from the starting point, to every other intersection on the map with infinity. This is done not to imply that there is an infinite distance, but to note that those intersections have not yet been visited. Now, at each iteration, select the current intersection. For the first iteration, the current intersection will be the starting point, and the distance to it will be zero. For subsequent iterations, after the first, the current intersection will be the closest unvisited intersection to the starting point.

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabelling the unvisited intersection with this value, if it is less than its current value. In effect, the intersection is if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabelled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighbouring intersection, mark the current intersection as visited, and select the unvisited intersection with lowest distance, from the starting point, as the current intersection. Nodes marked as visited are labelled with the shortest path from the starting point to it and will not be revisited or returned to.

Continue this process of updating the neighbouring intersections with the shortest distances, then marking the current intersection as visited and moving onto the closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited, as is the case with any visited intersection, you have determined the shortest path to it, from the starting point, and can trace your way back, following the arrows in reverse. In the algorithm's implementations, this is usually done, after the algorithm has reached the destination node, by following the nodes' parents from the destination node up to the starting node. That is why we keep also track of each node's parent.

Listing 5.1: Pseudocode of Dijkstra's Algorithm.

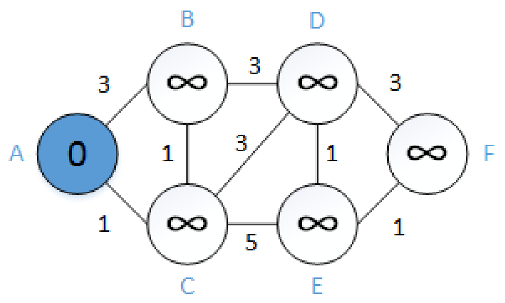
```
function Dijkstra(Graph, source):
  create vertex set Q
  for each vertex v in Graph:// Initialization
    dist[v] = INFINITY// Unknown distance from source to v
    prev[v] = UNDEFINED// Previous node in optimal path from source
    add v to Q// All nodes initially in Q (unvisited nodes)
  dist[source] = 0// Distance from source to source
  while Q is not empty:
    u = vertex in Q with min dist[u]// Source node will be selected
        first
    remove u from Q
    for each neighbor v of u:// where v is still in Q.
      alt = dist[u] + length(u, v)
      if alt < dist[v]:// A shorter path to v has been found
        dist[v] = alt
        prev[v] = u
  return dist[], prev[]
```

This algorithm makes no attempt to direct "exploration" towards the destination as one might expect. Rather, the sole consideration in determining the next "current" intersection is its distance from the starting point. This algorithm therefore expands outward from the starting point, interactively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path. However, it may also reveal one of the algorithm's weaknesses: its relative slowness in some topologies. Figure 5.5 shows an example of Dijkstra's algorithm with some real values

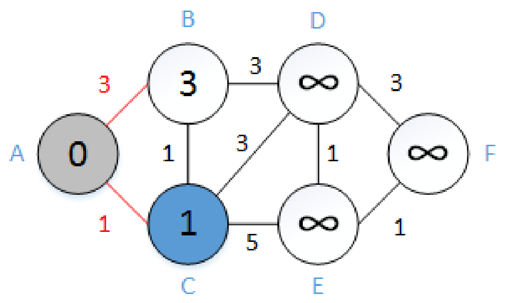
5.3.2 Navigation Instructions Generation

The generation of navigation instructions is a very important part of our application and it is a fact that makes a very difficult and complex process. The user of our application is able to reach his desired destination following a sequence of instructions. In order to provide the navigation instructions, we should take into consideration a set of things. First of all, we should calculate the shortest path, depending on the user needs. For example, in case of disabled people our algorithm calculates the shortest path which avoid stairs for transition between the floors. For each case we compute the shortest path using the Dijkstra Algorithm in order to avoid the nodes of graph that we can't cross (stairs nodes in case of disabled people).

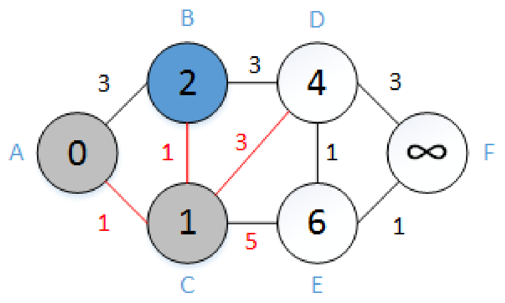
Secondly, taking the shortest path from our algorithm, the next step is to generate the appropriate instructions that should be followed by the user. In order to give a set of well-defined and simple



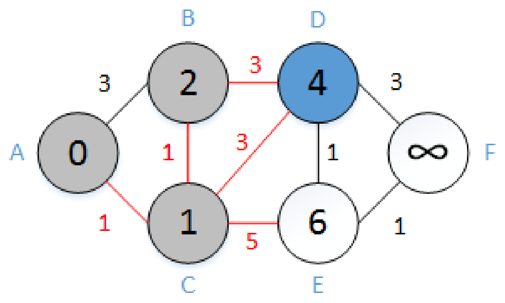
(a) Step 1



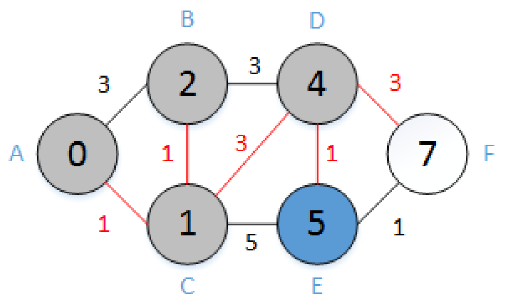
(b) Step 2



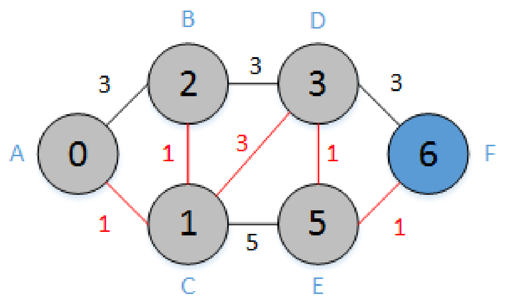
(c) Step 3



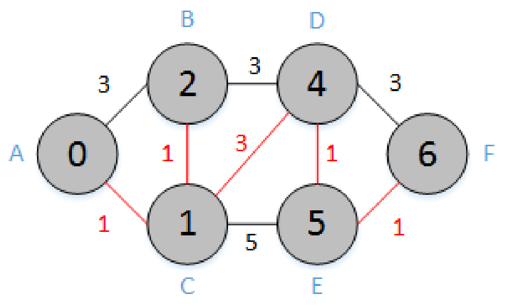
(d) Step 4



(e) Step 5



(f) Step 6



(g) Step 7

Figure 5.5: Dijkstra's algorithm steps.

instructions we created a language with its own grammar and syntax. This grammar consists of a set of verbs, subjects, objects and local determinations.

To begin with defining the meaning of instruction. An instruction is a sentence that urge the user do an action and consists of a set of words in the appropriate format and sequence that well defines the desired action. Each instruction follows a set of rules that should be applied in order to guide the user at his desired destination. More specifically, an instruction should be started with a verb that indicates the desired action to the user. The following are the basic set of verbs that we used in each occasion.

- *Walk*: Using this verb, we can advise the user to walk a number of meters or feet in order to reach the next node of our graph. This word should be followed by a local determination as "ahead", "right", "left" in order to give the appropriate direction to the user. Furthermore, this verb can be followed by a number of feet that should be traversed by the user.
- *Use*: This verb is used in case the user should change floor in order to reach his destination. Using this verb, we can advise the user to use a multifloor object in order to be transferred in other floor. Followed by an object, suggests the user to use an elevator, stairs etc. to ascend or descend floors.
- *Ascend - Descend*: Using these verbs, we can advise the user to ascend or descend floors in case the user should change floor in order to reach his destination. These verbs can be followed by the level of floor that the user should be transferred to meet the next node of our graph.
- *Turn*: This verb is used in case the user should change direction in order to reach his destination. This word should be followed by a local determination that indicated the appropriate direction to the user. More specifically these local determinations are the words "left" and "right" connected with a quantitative determination that suggest how the user must turn in order to follow the correct direction. It is worth mentioning, that we took into consideration the size of angle in each case and if this size was a significant factor for userâĂŹs navigation. If the angle that the user should turn was less than 10 degrees, we ignored it and we advised user to walk ahead to the next node.
- *Arrive*: Using this verb, we can notify the user that he reached his destination. This word is used only in the last instruction and can be followed by the object that user wanted to find.
- *Proceed*: This verb is used in case that the user should walk inside an indoor object for a number of feet. This word should be followed by a number of feet that the user should cover and an object that the user should going through in order to reach the next node of his route.

- *See*: Using this verb, we can point out the objects that the user can encounter through his transfer from a node to another. This word should be followed by the set of objects that the user encounter in each side and of course the side that the user can be see these objects.

Additionally we had enriched these verbs with a set of local determinations which can be enabled in some cases. These determinations are used in order to indicate the desired direction to the user and may be the words "right", "left", "front", "back", etc. Using these determinations we are able to give a complete instruction to the user in order to complete an action.

As we already mentioned, our grammar contains a set of objects that are used in order to make a correct instruction form. These words, may be contains the whole set of objects that the user created through our application, as office, corridor, room, elevator etc.

In more detail we compose our grammar using the following rules and expressions.

expression ⇒ exp | exp and exp

exp ⇒ `use_exp` | `turn_exp` | `arrive_exp` | `proceed_exp` | `walk_exp` | `meet_exp`

`use_exp` ⇒ `use_kwd` `multifloor_object` "to" `transition_kwd` "at" num `number_suffix` floors

`turn_exp` ⇒ `turn_verb` `direction_kwd`

`turn_verb` ⇒ `turn_kwd` | `turn_kwd` slightly

`direction_kwd` ⇒ right | left

`arrive_exp` ⇒ `arrive_kwd` "at" object | "the" destination object is front of you

`proceed_exp` ⇒ `proceed_kwd` num feet inside `indoor_object`

`walk_exp` ⇒ `walk_kwd` num feet

`meet_exp` ⇒ will meet `object_list` on your left and `object_list` on your right

`object_list` ⇒ object | object `object_list`

object ⇒ `indoor_object` | `multifloor_object` | `movable_object`

`transition_kwd` ⇒ ascend | descend

num ⇒ ['0'-'9']+

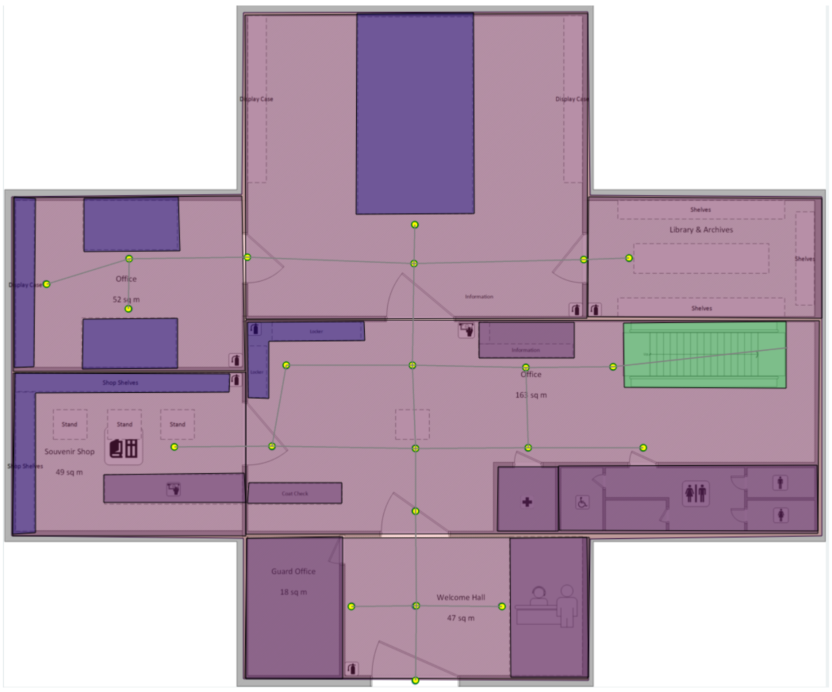
`number_suffix` ⇒ st | nd | rd | th

Some examples describing the instructions that our system can produce, include the following:

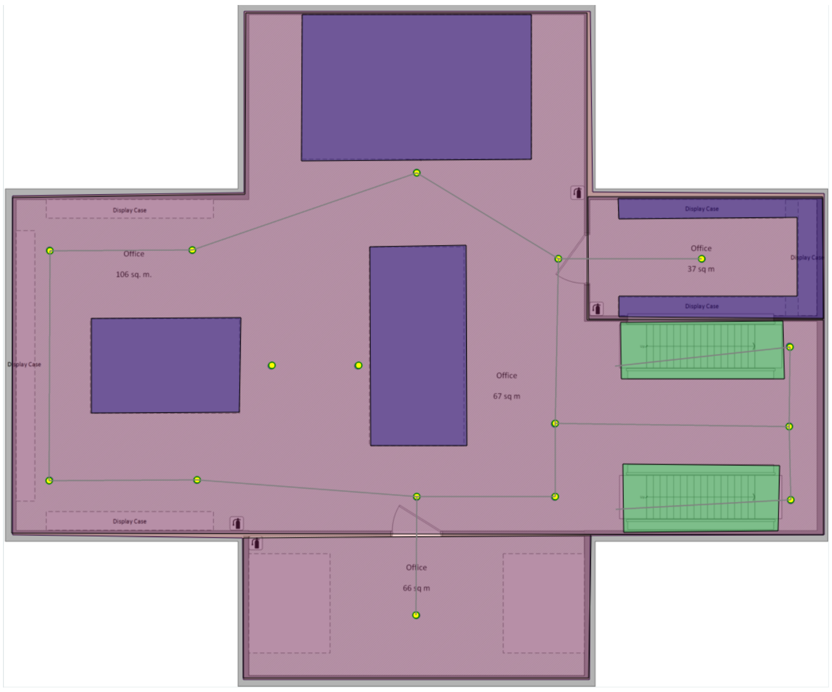
- Walk 18 feet and then turn left.

- Walk 12 feet, turn slightly left.
- Use elevator to ascend at the 2nd floor.
- Walk 17 feet, Use Elevator B to go to 4th Floor.
- Walk 84 feet, arrive at Christodoulakis Office.
- Use stairs to ascend to the 2nd floor.
- Walk 25 feet, Use Stairs to go to 3th Floor.
- Walk 31 feet, the destination Conference Center is on your left.
- Arrive at your destination.
- During this path, you can meet the Library Room and 145P48 Course Room on your left and Garofalakis Office on your right.
- The destination Coffee Shop is front of you.

To determine the appropriate instructions profound knowledge of geometric relations was needed, since the calculation of angles and the direction was a very complex process. More specifically, at each step we took into account the position of three nodes, as well the direction of the user in order to generate the correct instruction. It is worth mentioning, that we took into consideration the size of angle in each case and if this size was a significant factor for userâĂŹs navigation. If the angle between the three nodes was less than 10 degrees, we ignored it and we advised user to walk ahead to the next node. In each case we should enable the appropriate set of words in order to give a correct instruction to the user. In the following figures, Figure 7.8, Figure 7.9, Figure 7.10, and Figure 7.11 are presented the floor plans of Natural History Museum with all connected nodes and the sequence of instructions which are given to the user in order to reach his destination. This building consists of four floors which are connected with stairs. Each floor contains numerous of objects, either indoor or movable objects.

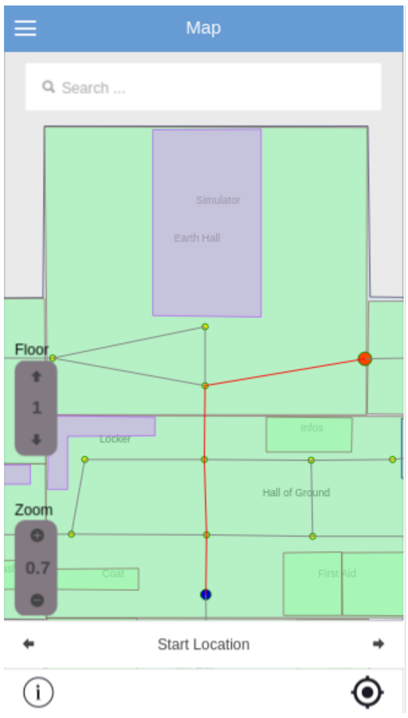


(a) First Floor.



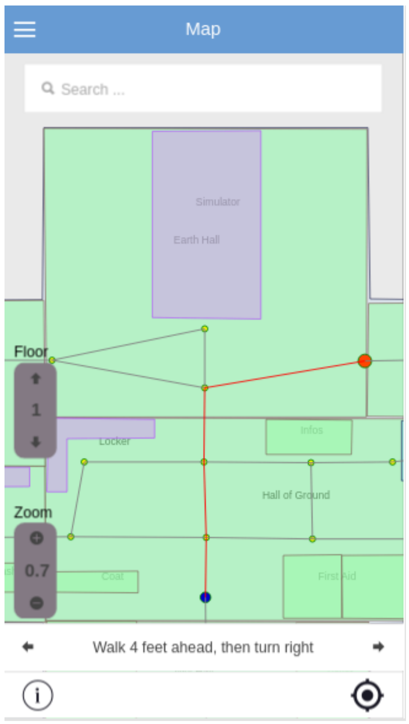
(b) Second Floor.

Figure 5.6: Natural History Museum - First and Second Floor.



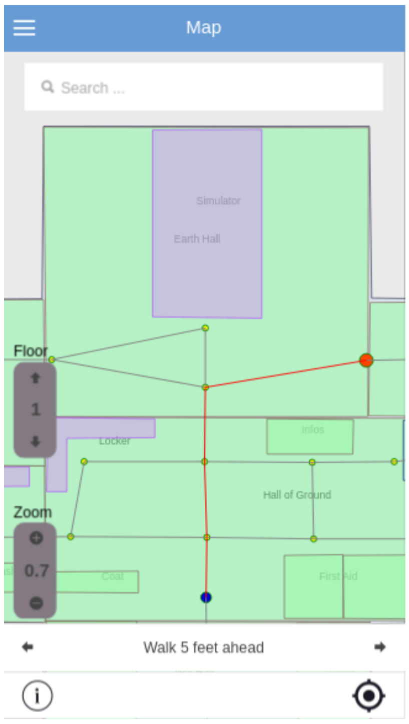
Start Location

(a) Start Location.



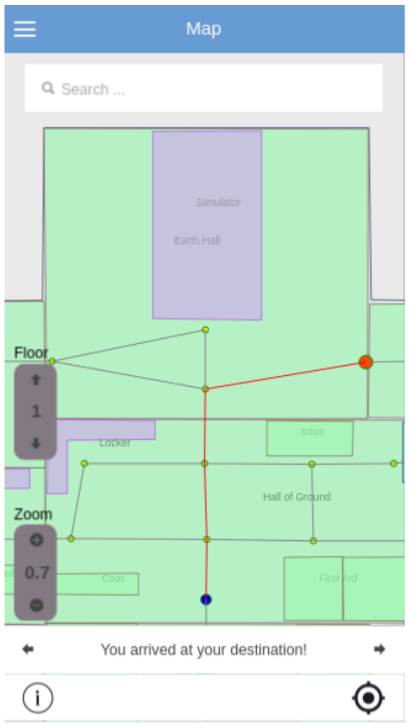
Walk 4 feet ahead, then turn right

(b) Walk 4 feet ahead, then turn right.



Walk 5 feet ahead

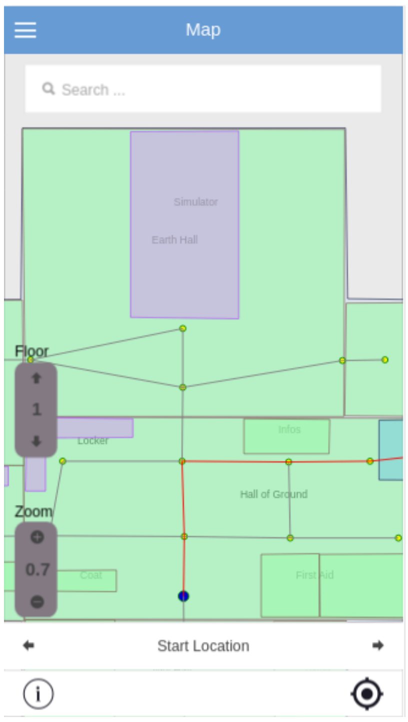
(c) Walk 5 feet ahead.



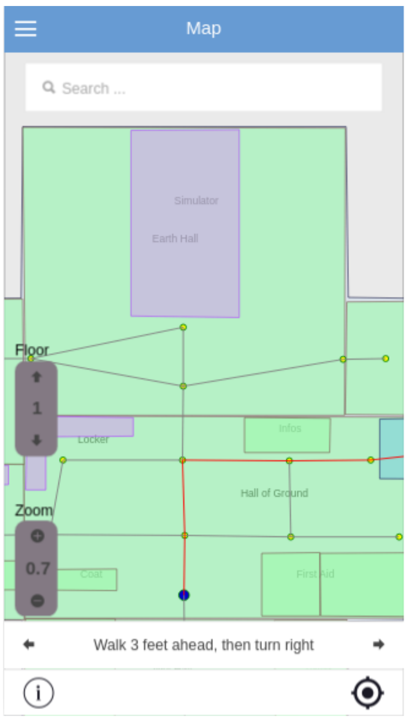
You arrived at your destination!

(d) You arrived at your destination.

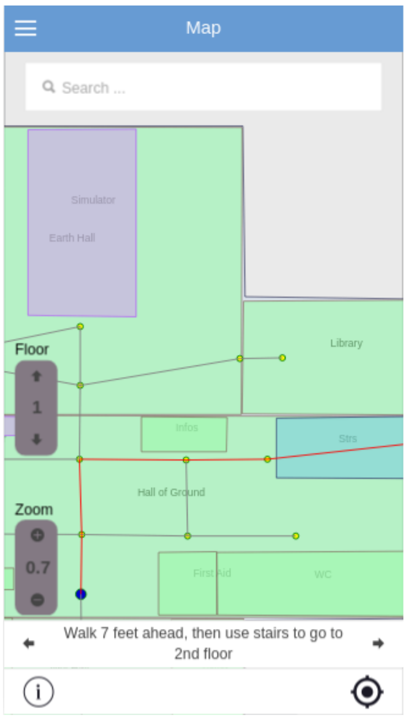
Figure 5.7: Natural History Museum - Path from Entry to Library.



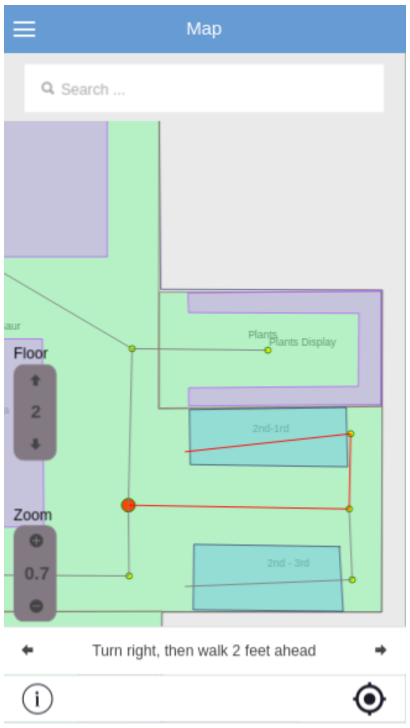
(a) Start Location.



(b) Walk 3 feet ahead, then turn right.

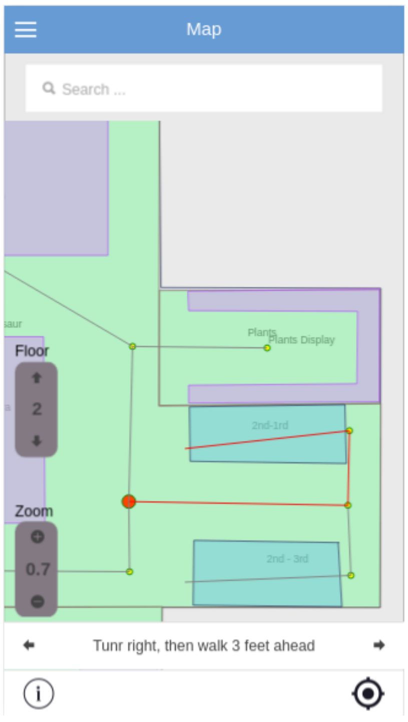


(c) Walk 7 feet ahead, then use stairs to go to 2nd floor.

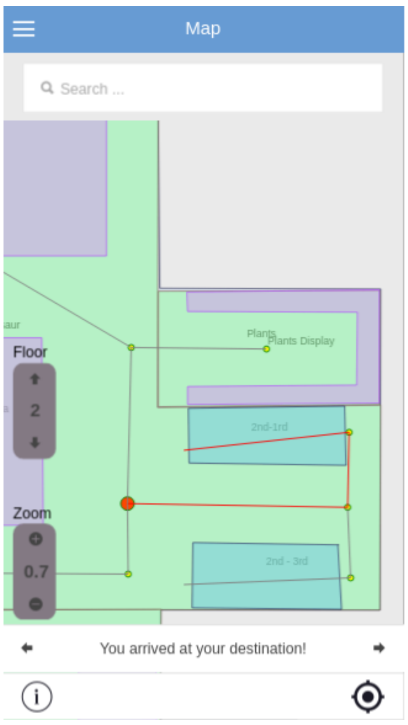


(d) Turn right, then walk 2 feet ahead.

Figure 5.8: Natural History Museum - Path from Entry to Tyrannosaurus (cont 'd).



(a) Turn right, then walk 3 feet ahead.



(b) You arrived at your destination.

Figure 5.9: Natural History Museum - Path from Entry to Tyrannosaurus.

Summary

In this chapter we presented the model that we have developed for the support of the building creation and design. Furthermore, we described three example implementations for the Model in order to support creation and annotation of floor plans. Finally, we described the navigation capabilities through description of shortest path calculation and the description of navigation instruction.

Chapter 6

Architecture

This chapter described the overall system architecture, identifies its basic components and provides an in depth analysis of the internal functionality. Moreover it allocates the architectural decisions that were made for the most important application aspects.

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 the web server hosting the application. Figure 6.1 displays the overall system architecture.

For the development of the application we adopted several design patterns [32]. 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 [9, 31]. 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 [58, 59] and the Observer patterns 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 6.1 presents the server side architecture, while Section 6.2 presents the client side architecture.

6.1 Server Side

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 components, scalability, robustness and security. As shown in Figure 6.1, the server side is comprised of a number of distinct modules which are described

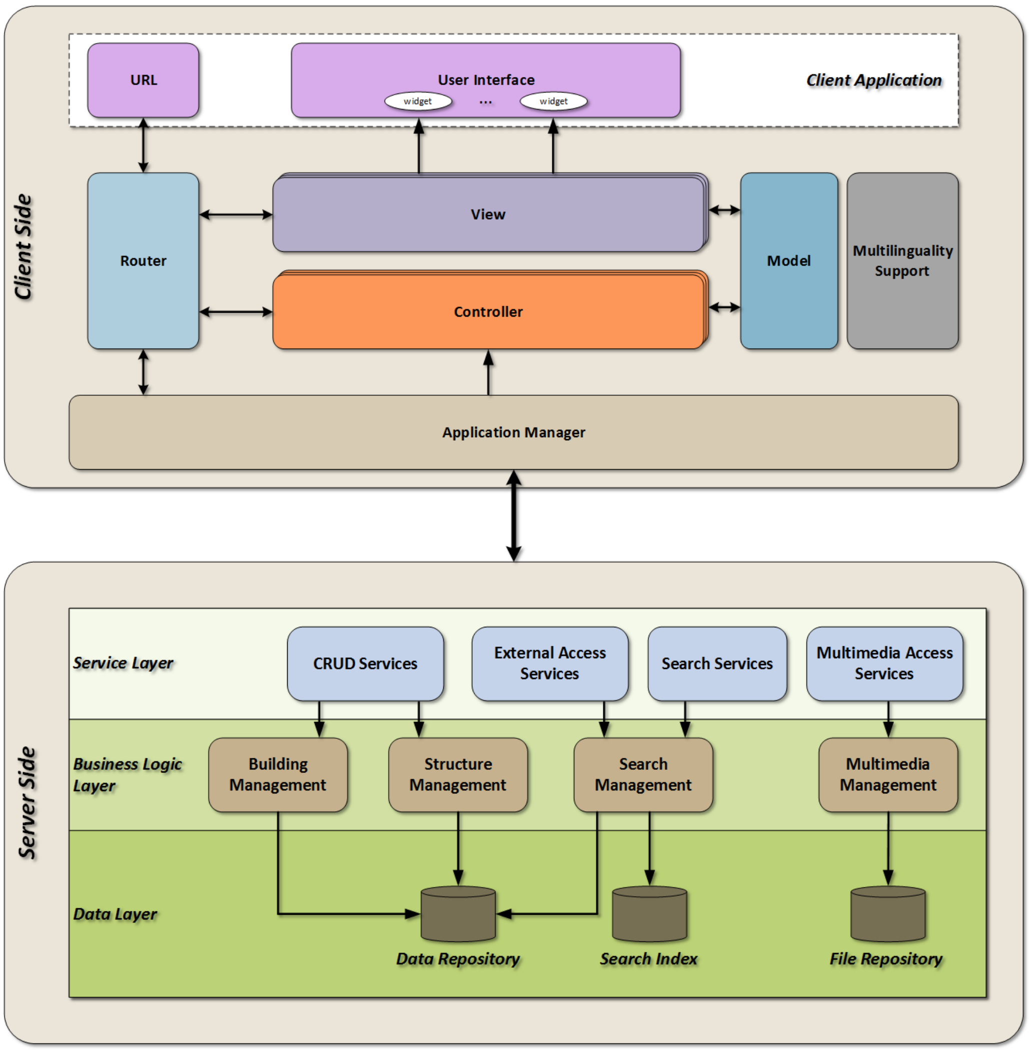


Figure 6.1: System reference architecture.

below.

6.1.1 Service Layer

The Service Layer controls the communication between the client-side logic and the server-side logic, by exposing a set of services (operations) to the client-side components[8]. These services comprise the middle-ware concealing the application business logic from the client and have been build as RESTful [52, 30]. The basic system services are:

- *CRUD Services*: Facilitate the creation, retrieval, update and deletion of a building, a floor, an object associated with a structure, a user etc.
- *External Access Services*: Provide the means for the client side and external systems to user the data of the system.
- *Multimedia Access Services*: Provide the means for the client side and external systems to user the data of the system.
- *Search Services*: Provide the client side with the ability to search the data.
- *Multimedia Access Services*: Enable the access to the uploaded multimedia files and their respective thumbnails.

6.1.2 Business Logic Layer

The Business Logic Layer, also know as Domain Layer, contains the business logic of the applications and separates it from the Data Layer and the Service Layer. In more details:

- *The Building Management Module*, which manages the building, as well as for the floor and object (de)composition in our system.
- *The Structure Management Module*, is responsible for the structure management.
- *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.

6.1.3 Data Layer

The Data Layer accommodates external systems which are used to index and persist both data and multimedia files. Such systems are: (*a*) *the Data Repository*, storing all the data of the system, (*b*) *the Search Index*, allowing faster full text queries, and (*c*) *the File Repository*, persisting the multimedia files and thumbnails.The implementation details of the components that comprise the Data Layer can be found in Section 7.2.

6.2 Client Side

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 a high level of decoupling between the components forming the client logic we adopted the Model View Controller (MVC) design pattern [58, 59], 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, View and 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 behaviour, and (*c*) separation of the business logic from the user interface logic from User Interface logic to make the code easier to understand and maintain.

On the other side, the Observer pattern is also a key part in the familiar model-view-controller (MVC) architectural pattern and provides the means for achieving the decoupling of the user interface components. It is a software design pattern in which an object, called the subject maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is used in our system for enabling the message transmission between the entities that reside on the Client Side. In the next sub-chapter, we describe the components of the client-side architecture (Figure 6.1) in depth, focusing on their objective and internal functionality.

6.2.1 Application Manager

The Application Manager acts as a centralized point of control for the system's client side logic, handling the communication between the Controllers and the server side by making call to the RESTful services exposed in the Service Layer, and notifying the Presenters for their responses. It implements the Mediator pattern and handles the communication between the Controller and the Server Side. Additionally it enables seamless data caching on the client side with the use of a dedicated cache.

6.2.2 Router

The Router is used for deep-linking URLs to controllers and views (HTML partials). It manages the URL of the client browsers. Specifically provides a different path to each distinct interface, without raising a browser event that will force a reload on the whole page. Additionally when the URL changes, the Router analyses the new path (location) and handles the transition to the new View. This is implemented using mappings between the different URLs supported in the system, as well as the Controllers and the Views.

6.2.3 Multilinguality

The Multilinguality Support module manages the translation of the user interface elements through the use of certain configuration files. The process is easily adaptable and the system can be extended to support any language with the minimum effort. In other words, if is needed to support another language we can add a new configuration file with all translated terms of our application and then our application serve the desired language to the user.

6.2.4 Model

The Model refers to the business objects which are being used by our system. When the system needs to present information about a business object, the client side requests the respective information from the server side using the services. In turn, the server side services transfer the Model to the client side. Accordingly, when an update on the Model needs to be persistent, the client side send s the updated Model to the server side, initiating the corresponding procedures.

6.2.5 View

The Views are responsible for the presentation of the information in the user interface. Each vies controls a number of widget on the application's graphical user interface. It contains multiple handles that are responsible for listening user's actions and the HTML templates that define the presentation of the widgets.

6.2.6 Controller

The Controllers are the most important part of our application because contains the majority of the client side logic of our application. They are the modules that respond to the user input and interact with the Views in order to perform any change on the user interface. Additionally, they maintain the Model and change it appropriately. Every View has a dedicated Controller which manage, handle and propagate any changes that are to be performed or have already been performed to the user interface. It is also important that there are a number of Controllers that are form a composition of Controller in order to perform more specific, wide and complex functionality. The Controller also have access to the browsers in order to use all the desired feature of them, such as Local Storage, File Upload, GPS Location etc.

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 7

Implementation

The functionality of the system and the architecture has been successfully implemented as described above. This chapter provides an extended an in depth description about the implementation details of some of the components, as well as source code examples. This way the reader can realize how specific parts of the system have been implemented. We have split the chapter into two Sections. Section 7.1 describes the Client Side, while Section 7.2 describes the Server Side. For the source control management of our application we have used multiple Git repositories on BitBucket[1]. Furthermore, we have used Jenkins[2], which is a continuous integration server, allowing us to run automatic tests and deploy the applications easier and in a more controlled way.

7.1 Client Side

The client side refers to both the editor tool (desktop application) and the mobile application. These two systems are based on the latest web-application standards, rely on the JavaScript programming language and make extensive use of the AngularJS framework in order to extend the DOM functionality. Additionally, they have been created in order to match different user requirements, and their user interfaces are implemented differently in order to match the goals of their stakeholders.

Regarding to our mobile applications it is worth to note that it has been compiled and packages as a native mobile application using PhoneGap, making it compatible with all the major mobile operation systems, such as Android, iOS and Windows. This allows our applications to run without the need of internet access, or loading its source each time that the users accesses it. Furthermore, the use of Apache Cordova allowed us to provide more functionality by using various native platform features that are otherwise unavailable to web applications.

[1]https://bitbucket.org/

[2]https://jenkins-ci.org/

The client side is built using state-of-the-art technologies, such as JavaScript, HTML5 and CSS3. For the code organization of the client side we have used various open source libraries and frameworks such as AngularJS, Bootstrap etc. Section 7.1.1 presents the use of MVC pattern in our system. Section 7.1.2 provides some details about the way that the user interface have been implemented for our desktop and mobile applications. Section 7.1.3 describes the parts that bind the HTML templates with controllers and urls, while sections 7.1.4 and 7.1.5 describe the AngularJS controller and service implementation on the client side. Finally, the other two sections 7.1.6 and 7.1.7 describe the implementation of multilinguality and the use of SVG elements in our applications.

7.1.1 MVC Pattern

The client side of our applications has been based heavily on the Model View Controller design pattern. Numerous JavaScript frameworks implements the MVC pattern have emerged during the last years. Most of the 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 AngularJS.

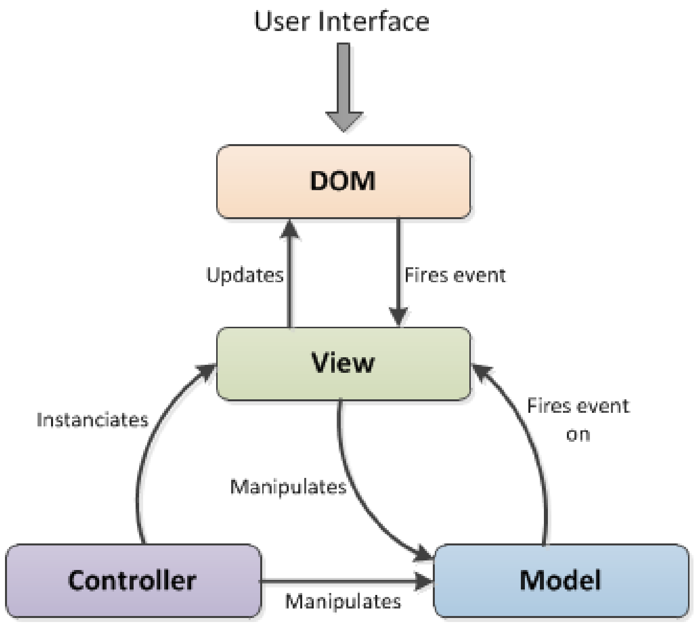


Figure 7.1: Model View Controller Pattern in our application.

AngularJS [10] has been developed as an MVC framework for JavaScript applications. It's main purpose is 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, including 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.

Figure 7.1 presents the MVC components and the interaction between them in our applications. The Controller instantiates the View and manipulates the Model. In turn, the View listens for updates of the Model, and when these happen it is re-rendered with the changes. This functionality is called two way binding and is presented in Figure 7.2. The View is also responsible for the updating of the Document Object Model (DOM) [41] when needed, as well as the handling of interaction events which are triggered by the user. In some cases the View also manipulates the Model.

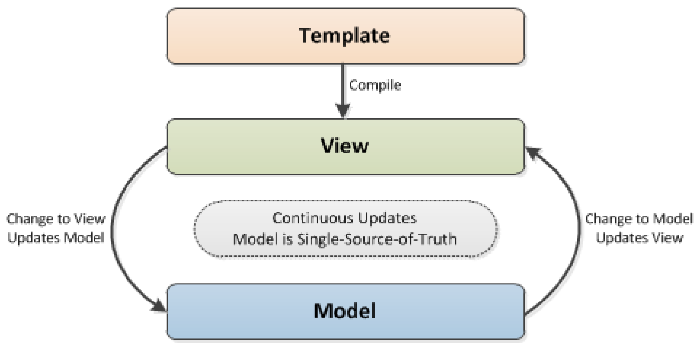

Template
Compile
View
Change to View Updates Model
Continuous Updates Model is Single-Source-of-Truth
Change to Model Updates View
Model

Figure 7.2: Two Way Binding.

7.1.2 User Interface

This section presents some user interface implementation details for our applications.

Editor Tool

When building an application optimized for desktop such as our application and editor tool, the developer take into consideration the size of the screen which affects instantly the usability of the applications. Figure 7.3 provides the main structure of our desktop application in order to ensure the best usability of our application. As we can see, we have a top bar with the basic information about the user and our application and also a sidebar which gives more choices to the user.

It is also important that we tried to build our desktop application bearing in mind the responsiveness which is a strong part of modern applications. As it is rational this responsiveness could not be achieved in every different page. It is worth to mention that we used Bootstrap Framework in

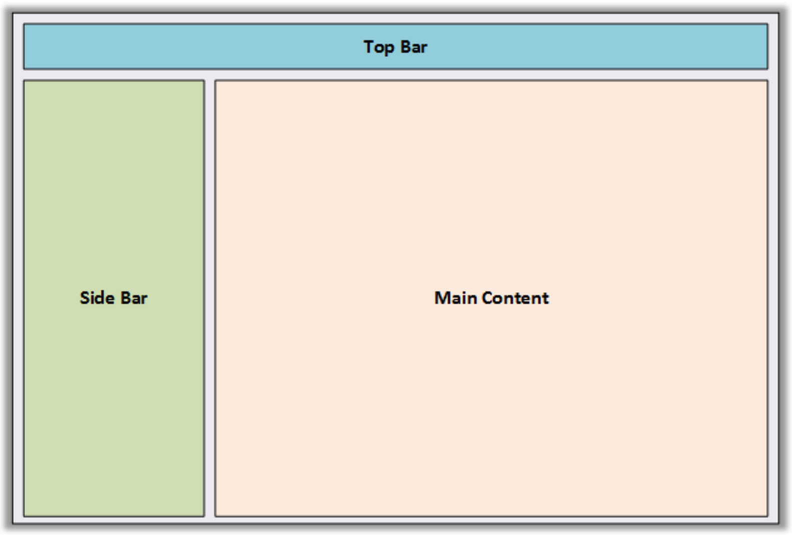


Figure 7.3: The main areas of our desktop user interfaces.

order to give the majority of responsiveness in our application but also we used specific annotations whenever is needed. Listing 7.1 presents some sample code of the css style using @media rule.

Listing 7.1: Example of css style.

```
@media (min-width: 992px) {
  .app.container {
    width: 970px;
  }
  .app.container .app-header,
  .app.container .app-aside {
    max-width: 970px;
  }
  .app.container .app-footer-fixed {
    max-width: 770px;
  }
  .app.container.app-aside-folded .app-footer-fixed {
    max-width: 910px;
  }
  .app.container.app-aside-dock .app-footer-fixed {
    max-width: 970px;
  }
}
```

7.1.3 Router

Modern web applications, built with the latest technologies, enable more powerful interaction with the users. Their features are very close to 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 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 being used to manipulate all of the interfaces presented to the user, they are becoming more complex. Two major problems have arise from this complexity. Firstly, the change of user interface using JavaScript needs special treatment in order to allow the browser's history support. This appears because the web browsers tend to push the URLs to history and reload a new one whenever a change is performed. In RIAs this is not always the case. The second problem is the ability of 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 applications and their URLs. Our Router is based on the Router object provided by the AngularJS framework. Listing 7.2 presents sample code of a router (state provider) in our application.

Listing 7.2: Example of router.

```
/** Home Page or Dashboard Page of my Application */
.state('app.dashboard', {
    url: '/dashboard',
    templateUrl: 'tpl/app_dashboard_v1.html',
    resolve: load(['js/controllers/chart.js'])
})

/** Page which contains building list */
.state('app.buildings', {
    url: '/buildings',
    controller: 'BuildingsAllCtrl',
    templateUrl: 'tpl/buildings.html',
    resolve: load(['js/controllers/buildings.js',
        'js/factories/BuildingFactory.js'])
})

/** Page which contains building detail */
.state('app.detailOfBuilding', {
    url: '/building/{buildingId}',
    templateUrl: 'tpl/building.detail.html',
    controller: 'BuildingDetailCtrl',
    resolve: {
        deps: ['$ocLazyLoad',
            function ($ocLazyLoad) {
                return $ocLazyLoad.load([
                    'js/controllers/building.js',
                    'js/factories/BuildingFactory.js',
                    'js/app/music/theme.css',
                    'com.2fdevs.videogular',
                    'com.2fdevs.videogular.plugins.controls',
                    'com.2fdevs.videogular.plugins.overlayplay',
                    'com.2fdevs.videogular.plugins.poster',
                    'com.2fdevs.videogular.plugins.buffering',
                    'js/app/music/theme.css']);
            }]
    }
})
```

7.1.4 Controller

In Angular, a Controller is defined by a JavaScript constructor function that is used to augment the Angular Scope. When a Controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified Controller's constructor function. In our implementation we use controller in order to respond to the user input and user action and interact with the View in order to perform any change on the user interface. Additionally, they are used to maintain the Model and change it appropriately. Listing 7.3 presents some sample code of a Controller in the JavaScript language.

Listing 7.3: Example of controller implementation

```javascript
app.controller('FloorCtrl', ['$rootScope', '$scope', '$state',
    '$stateParams', '$http', 'FileUploader', '$timeout', '$modal',
    '$log', '$filter', 'floorFactory', 'buildingFactory',
    '$localStorage', 'myConfig',
function ($rootScope, $scope, $state, $stateParams, $http,
    FileUploader, $timeout, $modal, $log, $filter, floorFactory,
    buildingFactory, $localStorage, myConfig) {

    $scope.apiUrl = myConfig.apiUrl;
    /**
     * Meter Analogy
     * */
    $scope.meterAnalogy = 1;
    /** Get Floor Id from stateParams */
    $scope.floorId = $stateParams.floorId;
    $scope.buildingId = $stateParams.buildingId;
    /** Initialize Url for Image Floor Plan */
    $scope.urlFloorPlan = '';
    $scope.floor = {
        scale: 1
    };
    /** Initialize all floor data */
    $scope.pos = {};
    $scope.polygon = {};
    $scope.basicsvg = {};
    /** Floor Borders */
    $scope.floorElements = [];
    $scope.nonSemanticObjectElements = [];
    /** Objects Elements */
    $scope.objectElements = [];
    $scope.detailedObjectElements = [];
    /**
     * Nodes Initialization
     * */
    $scope.nodes = [];
    $scope.links = [];
    /**
     * Temporary Points for Polygon
     * */
    $scope.temporaryPoints = [];
    /** Distance Calculator between two Points */
    function lineDistance(point1, point2) {
        var xs = 0;
        var ys = 0;
        xs = point2.x - point1.x;
        xs = xs * xs;
        ys = point2.y - point1.y;
        ys = ys * ys;
        var res = Math.sqrt(xs + ys);
        return (res / 100) * $scope.meterAnalogy;
    }
    /** Initialize all possible functions */
    $scope.onFirstBtnClickResult = "";
    $scope.secondBtnInput = "";
    $scope.onDblClickResult = "";
    $scope.onMouseDownResult = "";
    $scope.onMouseUpResult = "";
    $scope.onMouseEnterResult = "";
    $scope.onMouseLeaveResult = "";
    $scope.onMouseMoveResult = "";
    $scope.onMouseOverResult = "";
    /**
     Accepts a MouseEvent as input and returns the x and y
     coordinates relative to the target element.
     */
    var getCrossBrowserElementCoords = function (mouseEvent) {
        var result = {
            x: 0,
            y: 0
        };
        if (!mouseEvent) {
            mouseEvent = window.event;
        }
        if (mouseEvent.pageX || mouseEvent.pageY) {
            result.x = mouseEvent.pageX;
            result.y = mouseEvent.pageY;
        } else if (mouseEvent.clientX || mouseEvent.clientY) {
            result.x = mouseEvent.clientX + document.body.scrollLeft +
                document.documentElement.scrollLeft;
            result.y = mouseEvent.clientY + document.body.scrollTop +
                document.documentElement.scrollTop;
        }
        if (mouseEvent.target) {
            var offEl = mouseEvent.target;
            var offX = 0;
            var offY = 0;
            if (typeof(offEl.offsetParent) != "undefined") {
                while (offEl) {
                    offX += offEl.offsetLeft;
                    offY += offEl.offsetTop;

                    offEl = offEl.offsetParent;
                }
            } else {
                offX = offEl.x;
                offY = offEl.y;
            }
            result.x -= offX;
            result.y -= offY;
        }
        return result;
    };
    var getMouseEventResult = function (mouseEvent, mouseEventDesc) {
        var coords = getCrossBrowserElementCoords(mouseEvent);
        return mouseEventDesc + " at (" + coords.x + ", " + coords.y +
            ")";
    };
    /** Event Handlers */
    $scope.onFirstBtnClick = function () {
        $scope.onFirstBtnClickResult = "CLICKED";
    };
    $scope.onSecondBtnClick = function (value) {
        $scope.onSecondBtnClickResult = "you typed '" + value + "'";
    };
    $scope.onDblClick = function () {
        $scope.onDblClickResult = "DOUBLE-CLICKED";
    };
    $scope.onMouseDown = function ($event) {
        $scope.onMouseDownResult = getMouseEventResult($event, "Mouse
            down");
    };
    $scope.onMouseUp = function ($event) {
        $scope.onMouseUpResult = getMouseEventResult($event, "Mouse
            up");
    };
    $scope.onMouseEnter = function ($event) {
        $scope.onMouseEnterResult = getMouseEventResult($event, "Mouse
            enter");
    };
    $scope.onMouseLeave = function ($event) {
        $scope.onMouseLeaveResult = getMouseEventResult($event, "Mouse
            leave");
    };
    $scope.onMouseMove = function ($event) {
        $scope.onMouseMoveResult = getMouseEventResult($event, "Mouse
            move");
    };
    $scope.onMouseOver = function ($event) {
        $scope.onMouseOverResult = getMouseEventResult($event, "Mouse
            over");
    };
    /** End of editor functionality */
}]);
```

7.1.5 Services and Factories

The Services and Factories in our application are responsible for the communication between the client side and the server side. Besides of the services at the server side, we implemented a list of services and factories at the client side. In more details, through these services and factories we can call server side services through AJAX calls and get the responses in the client side. 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. Listing 7.4 presents some sample code of a Service and a Factory in the JavaScript language.

Listing 7.4: Example of a factory implementation.

```
app.factory('PortfolioFactory', function ($http, myConfig) {
    var portfolioFactory = {};
    portfolioFactory.getAllMedia = function (token) {
        var promise = $http({
            method: 'GET',
            header: {
                Authorization: 'Bearer ' + token
            },
            url: myConfig.apiUrl + '/portfolio'
        });
        return promise;
    };
    portfolioFactory.getAllMediaByType = function (type, token) {
        var promise = $http({
            method: 'GET',
            header: {
                Authorization: 'Bearer ' + token
            },
            url: myConfig.apiUrl + '/portfolio/' + type
        });
        return promise;
    };
    portfolioFactory.deleteMultimedia = function (multimediaId, token)
       {
        var promise = $http({
            method: 'DELETE',
            header: {
                Authorization: 'Bearer ' + token
            },
            url: myConfig.apiUrl + '/portfolio/' + multimediaId
        });
        return promise;
    };
    portfolioFactory.getAllMediaSplitByType = function (token) {
        var promise = $http({
            method: 'GET',
            header: {
                Authorization: 'Bearer ' + token
            },
            url: myConfig.apiUrl + '/portfolio/split'
        });
        return promise;
    };
    return portfolioFactory;
});
```

7.1.6 Multilinguality

Both our client side applications, the editor tool and the mobile application, support multilinguality. Currently all the user interfaces that have been implemented are translated in both English and Greek.

In order to tackle this aspect of the system we exploited the techniques and configurations provided by AngularJS. To this end, firstly we used ids (e.g., "content.left.description", "header.title", "content.right.logo") to identify the different elements in our HTML templates that should be translated. Then in the appropriate configuration files we set the textual values to these ids, representing the actual label translation. Listing 7.5 presents some sample code of an HTML template, while Listing 7.6 provides some sample code from the configuration file that we use for the English translation.

Listing 7.5: Example of an HTML template.

```
<ul class="nav">
    <li class="hidden-folded padder m-t m-b-sm text-muted text-xs">
        <span translate="aside.nav.HEADER">Navigation</span>
    </li>
    <li ui-sref-active="active">
        <a ui-sref="app.dashboard">
            <i class="glyphicon glyphicon-stats icon
                text-info-dker"></i>
            <span class="font-bold"
                translate="aside.nav.DASHBOARD">Dashboard</span>
        </a>
    </li>
    <li ui-sref-active="active">
        <a ui-sref="app.buildings">
            <i class="fa fa-building icon text-info-dker"></i>
            <span class="font-bold" translate="aside.nav.BUILDINGS">My
                Buildings</span>
        </a>
    </li>
    <li ui-sref-active="active">
        <a ui-sref="app.users">
            <i class="fa fa-users icon text-info-dker"></i>
            <span class="font-bold"
                translate="aside.nav.USERS">Users</span>
        </a>
    </li>
    <li ui-sref-active="active">
        <a ui-sref="app.portfolio.type({type: 'image', page: '1'})">
            <i class="fa fa-image icon text-warning-lter"></i>
            <span class="font-bold"
                translate="aside.nav.PORTFOLIO">Portfolio</span>
        </a>
    </li>
    <li>
        <a href class="auto">
          <span class="pull-right text-muted">
            <i class="fa fa-fw fa-angle-right text"></i>
            <i class="fa fa-fw fa-angle-down text-active"></i>
          </span>
            <i class="glyphicon glyphicon-list-alt icon
                text-primary-dker"></i>
            <span class="font-bold"
                translate="aside.nav.STRUCTURES">Structures</span>
        </a>
        <ul class="nav nav-sub dk">
            <li class="nav-sub-header">
                <a href>
                    <span
                        translate="aside.nav.STRUCTURES">Structures</span>
                </a>
            </li>
            <li ui-sref-active="active">
                <a ui-sref="app.indoor">
                    <span translate="mine.indoor"></span>
                </a>
            </li>
            <li ui-sref-active="active">
                <a ui-sref="app.movable">
                    <span translate="mine.movable"></span>
                </a>
            </li>
            <li ui-sref-active="active">
                <a ui-sref="app.multifloor">
                    <span translate="mine.multifloor"></span>
                </a>
            </li>
        </ul>
    </li>
    <li ui-sref-active="active">
        <a ui-sref="app.settings">
            <i class="fa fa-cogs icon text-info-dker"></i>
            <span class="font-bold"
                translate="aside.nav.SETTINGS">Settings</span>
        </a>
    </li>
    <li class="line dk hidden-folded"></li>
    <li class="hidden-folded padder m-t m-b-sm text-muted text-xs">
        <span translate="aside.nav.your_stuff.YOUR_STUFF">Your
            Stuff</span>
    </li>
    <li>
        <a ui-sref="app.page.profile">
            <i class="icon-user icon text-success-lter"></i>
            <span
                translate="aside.nav.your_stuff.PROFILE">Profile</span>
        </a>
    </li>
    <li>
        <a ui-sref="app.docs">
            <i class="icon-question icon"></i>
            <span
                translate="aside.nav.your_stuff.DOCUMENTS">Documents</span>
        </a>
    </li>
</ul>
```

Listing 7.6: Example of the configuration file format used for achieving multilinguality.

```
"building_new" : {
    "header" : {
      "basicInfo" : "Basic Information",
      "subtitle" : "Please fill the information to continue",
      "create_button" : "Create Building"
    },
    "footer" : {
      "create_button" : "Submit"
    },
    "content" : {
      "form" : {
        "description" : "Description of Building ",
        "keywords" : "Keywords",
        "fullName" : "Full Name of Building",
        "shortName" : "Short Name of Building",
        "code" : "Code",
        "address" : "Address of Building",
        "phones" : "Phones of Building",
        "website" : "Website of Building",
        "numLevels" : "Number of Floors",
        "keywords" : "Keywords of Building",
      },
      "placeholder" : {
        "fullName":"Enter the Full Name of Building",
        "shortName":"Enter the Short Name of Building",
        "description":"Enter the Description of Building",
        "address":"Enter the Address of Building",
        "url":"Enter the Url of Building",
      }
    }
  }
```

7.1.7 SVG

In order to design and implement a useful and full functional editor, we used SVG language. SVG stands for Scalable Vector Graphics and it is a language for describing 2D-graphics and graphical applications in XML and the XML is then rendered by an SVG viewer. SVG is mostly useful for vector type diagrams like Pie charts, Two-dimensional graphs in an X, Y coordinate system etc. Most of the web browsers can display SVG just like they can display PNG, GIF, and JPG. SVG is a great way to present vector based line drawings and is a great complement to bitmaps, the latter being better suited for continuous tone images. One of the most useful things about SVG is that it's resolution independent, meaning that you don't need to think about how many pixels you have on your device, the result will always scale and be optimized by the browser to look great. The SVG specification defines 14 functional areas or feature sets such as:

- **Paths**: Simple or compound shape outlines are drawn with curved or straight lines that can be filled in, outlined, or used as a clipping path. Paths have a compact coding. For example M (for 'move to') precedes initial numeric x and y coordinates and L (line to) precedes a point to which a line should be drawn. Further command letters (C, S, Q, T and A) precede data that is used to draw various BÃl’zier and elliptical curves. Z is used to close a path. In all cases, absolute coordinates follow capital letter commands and relative coordinates are used after the equivalent lower-case letters.

- **Basic shapes**: Straight-line paths and paths made up of a series of connected straight-line segments (polylines), as well as closed polygons, circles and ellipses can be drawn. Rectangles and round-cornered rectangles are also standard elements.
- **Text**: Unicode character text included in an SVG file is expressed as XML character data. Many visual effects are possible, and the SVG specification automatically handles bidirectional text (for composing a combination of English and Arabic text, for example), vertical text (as Chinese was historically written) and characters along a curved path (such as the text around the edge of the Great Seal of the United States).

7.1.8 QR Codes

A QR code is a type of matrix bar-code invented in 1994 by Toyota. QR codes were first intended to be used for the automotive industry. Due to their fast readability and large storage capacity, they quickly became famous for other purposes like advertising by storing an address or a Uniform Resource Locator (URL).

Figure 7.4: Example of QR Code.

In the context of this thesis, QR codes are used to identify the current position of a device inside the building. Indeed, the QR codes are placed at strategic places inside the building. When the user scans a QR code with his device, the system recognizes his current position. Figure 7.4 presents an example of QR Code.

Scanning QR codes on Android

There exist various open source software suites for reading QR codes. They all basically use the same decoding principle, namely to use the built-in camera of a mobile device to scan and decode a QR code. One of these software suites is called ZXing. The problem with ZXing is that it is not possible to use it within an existing application. Developers can only use it by first placing a button somewhere inside the UI of the application, and by implementing a click event that will open a camera window in order to use the functionalities provided by ZXing. This is clearly not in the sense we wanted our application to behave. The software suite that has been finally chosen for this thesis is called ZBar. The latter can be directly included into iPhone or Android applications, which makes the QR code scanning process completely transparent to the user.

Mobile Application

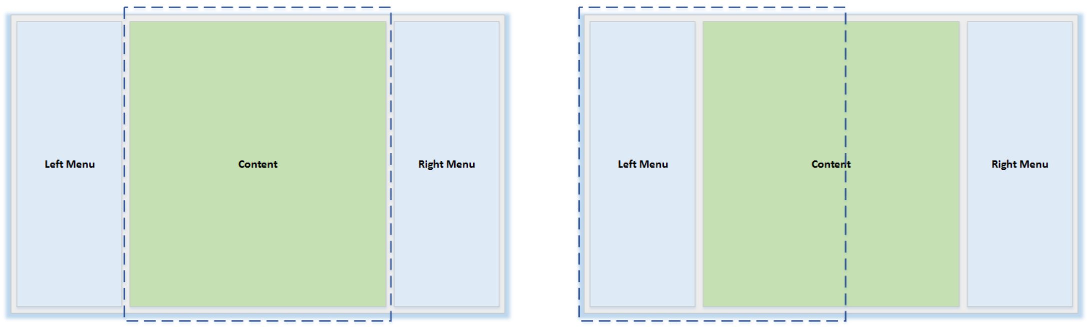
When building an application optimized for mobile devices, the developer must take into account the small size of the screen which affects the usability of the applications. The menus that are provided with the mobile applications are at most times hidden and become visible only when the user needs them. 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.

Figures 7.5a and 7.5b present the main elements of our mobile 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 right (depending on which menu is open), which causes one of the two menus to appear.

In order to make a native-like mobile application, we used Ionic Framework [2]. Ionic is a powerful HTML5 SDK that helps you build native-feeling mobile apps using web technologies like HTML, CSS, and Javascript.

Built on top of AngularJS and Apache Cordova, Ionic provides tools and services for developing hybrid mobile apps using Web technologies like CSS, HTML5, and Sass. Ionic is focused



(a) Menu is hidden. **(b)** Menu is visible.

Figure 7.5: The Viewport.

mainly on the look and feel, and UI interaction of mobile applications. Ionic simply match with PhoneGap and AngularJS in order to create a powerful SDK most suited to develop rich and robust applications. Listing 7.7 presents a sample code of an HTML template using the Ionic Framework in our mobile application. Ionic tags can be immediately recognized since their names are in the form of "ion-*", while the AngularJS tags use generally the form of "ng-*".

Listing 7.7: Example of html template.

```
<ion-view class="category-feeds-view">
    <ion-nav-title>
        <span>List Of Objects</span>
    </ion-nav-title>
    <ion-content>
        <div class="list category-feeds">
            <!-- Multimedia Files -->
            <div style="border: 1px solid;border-radius:
                5px;border-color: #ddd;margin-top:10px">
                <div class="item item-divider">
                    <i class="ion-images"></i>
                    Indoor Objects
                </div>
            </div>
            <a ng-repeat="static in staticElements" class="item
                item-icon-right"
               ui-sref="app.generalobject(
                         {
                             buildingId: buildingId,
                             floorId: static.floorId,
                             objectId: static.id
                         })">
                <div class="thumbnail-outer">
                    <pre-img ratio="_1_1" helper-class="">
                        <img class="thumbnail"
                             ng-src="{{static.imgUrl}}"
                             spinner-on-load>
                    </pre-img>
                </div>
                <div>
                    <span class="title">{{static.shortName}}</span>
                    <p class="description">{{static.description}}</p>
                </div>
                <i class="icon ion-chevron-right"></i>
            </a>
            <div style="border: 1px solid;border-radius:
                5px;border-color: #ddd;margin-top:10px">
                <div class="item item-divider">
                    <i class="ion-images"></i>
                    Movable Objects
                </div>
            </div>
            <a ng-repeat="mov in movableElements" class="item
                item-icon-right"
               ui-sref="app.generalobject(
                        {
                            buildingId: buildingId,
                            floorId: mov.floorId,
                            objectId: mov.id
                        })">
                <div class="thumbnail-outer">
                    <pre-img ratio="_1_1" helper-class="">
                        <img class="thumbnail"
                             ng-src="{{mov.imgUrl}}"
                             spinner-on-load>
                    </pre-img>
                </div>
                <div>
                    <span class="title">{{mov.shortName}}</span>
                    <p class="description">{{mov.description}}</p>
                </div>
                <i class="icon ion-chevron-right"></i>
            </a>
        </div>
```

```
    </ion-content>
</ion-view>
```

7.2 Server Side

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

7.2.1 Persistency

For the persistence of the data we have chosen a document database, MongoDB [46, 20]. 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. 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. The technology behind most NoSQL databases is based on distributed and parallel systems, as well as in-memory databases. There has been a lot of evaluation and criticism on NoSQL databases [43, 37], and a lot of debate about the selection of the best database technology between the two [67, 52]. 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.
- 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 are 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.

Some reasons, that led us to choose MongoDB are the following:

- Schema-less: Most of our data does not conform to a rigid relational schema. Hence, we can not bound it in the structure of a relational database and we need some more flexibility. Due to the fact that MongoDB allows us to store parts of our data in different forms with minor effort. Moreover, this makes our system capable of supporting interoperability with other external systems with minimum effort.
- 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.
- 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: Building, Structure, Object, Multimedia, User. 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.

7.2.2 RESTful Web Services

In order to expose our data to our client applications and other external systems we developed an API that enables data access through the use of RESTful services [4]. The reason of using REST was based on the following facts:

(*a*) less overhead compared with SOAP (*b*) minimize the coupling between client and server components in a distributed application (*c*) less duplication since HTTP already represents DELETE, GET, PUT and POST operations (*d*) our server is going to be used by many different clients that we do not have control over (*e*) more standardized, providing HTTP operations that are well understood and operate consistently (*f*) more human readable and testable there is no requirement to use complex data interchange formats like XML (*g*) a service that a client developer should be able to easily understand due to the consistent use of the HTTP protocol To this end, we built a set of

RESTful services exposing all the resources and functionality of our system and we created CRUD operations for every possible feature. These services are used by both our client side applications, the editor tool (desktop application) and the mobile application to fetch, create, update and delete data on our system.

7.2.3 Security

Security is a very important aspect in every system, especially in 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 transferringdata [61].

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.

7.2.4 JSON Web Token

JSON Web Token is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret or a public/private key pair using RSA. Below we explain some concepts of this definition further.

- **Compact**: Because of its size, it can be sent through an URL, POST parameter, or inside an HTTP header. Additionally, due to its size its transmission is fast.
- **Self-contained**: The payload contains all the required information about the user, to avoid querying the database more than once

In authentication, when the user successfully logs in using his credentials, a JSON Web Token will be returned and must be saved locally , instead of the traditional approach of creating a session in the server and returning a cookie. Whenever the user wants to access a protected route or resource, it should send the JWT, typically in the Authorization header using the Bearer schema. Figure 7.6 presents a diagram of this process. Therefore the content of the header should look like the following. **Authorization: Bearer <token>**

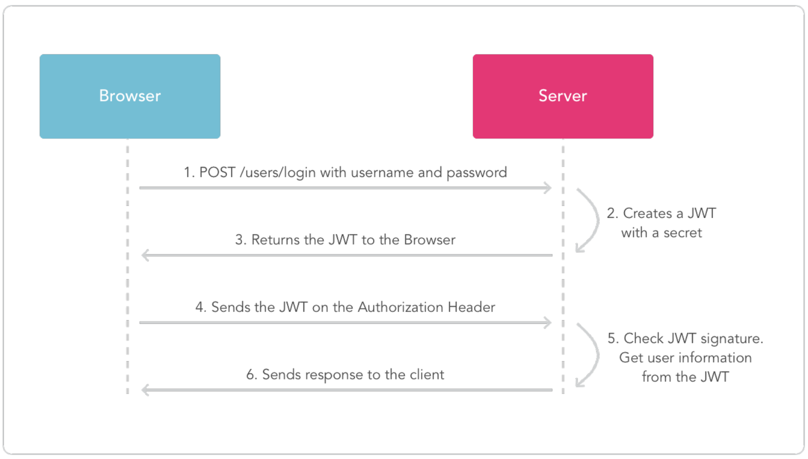

Browser
Server
1. POST /users/login with username and password
2. Creates a JWT with a secret
3. Returns the JWT to the Browser
4. Sends the JWT on the Authorization Header
5. Check JWT signature. Get user information from the JWT
6. Sends response to the client

Figure 7.6: Json Web Token process diagram.

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 8

Graphical User Interface

This chapter presents the methodology that we followed for designing the user interfaces of our applications, as well as the final product. Section 8.1 describes the phase of storyboard designing and user interface prototyping, Section 8.2 describes the evaluation methods that we had applied in order to design better and more useful applications and Section 8.3 presents the final outcome of the graphical user interface as they have been implemented.

8.1 Prototypes

At the initial stages of the design process, we sketched 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 [49]. 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.

The following sections present some early user interface prototypes of the system along with the action that the user was supposed to perform. More specifically, Section 8.1.1 presents the user interface prototypes of the editor tool (desktop application), while Section 8.1.2 provides the user interface prototypes of our player application (mobile application).

8.1.1 Editor Tool

The figures of this section present some of user interface prototypes of the editor tool which were drawn in the early stages of the design process. These prototypes were initially used for the creation of storyboards and of course for usability evaluation process. Each prototype has a title and the purpose that the user interface serves.

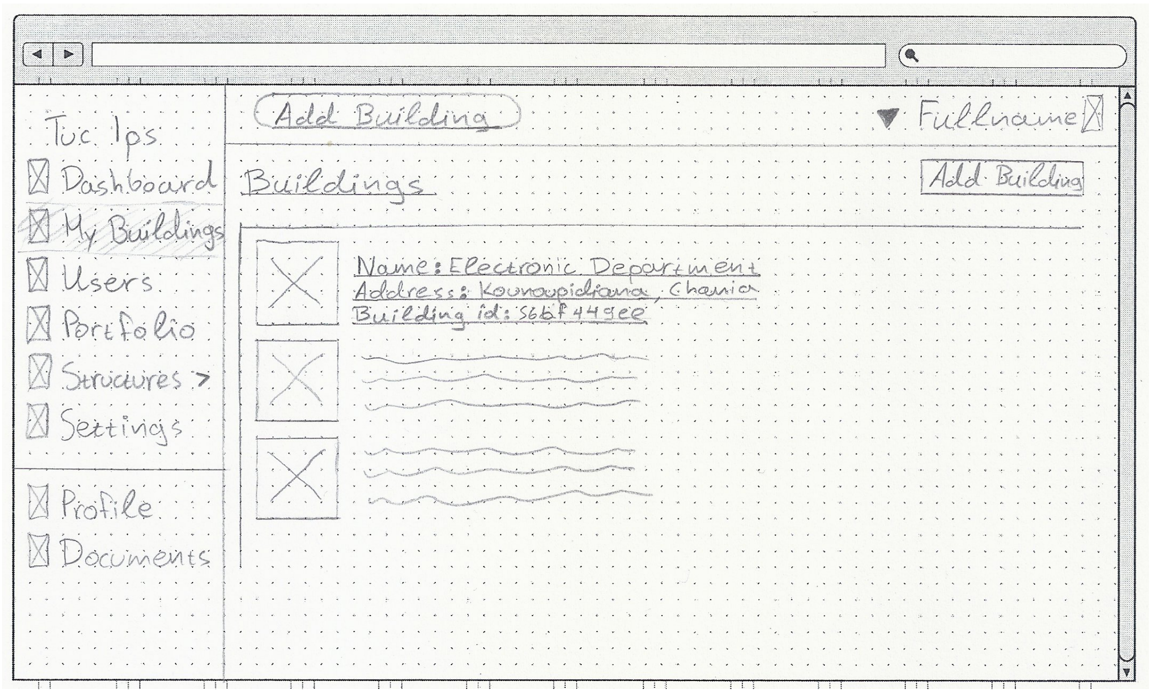


Figure 8.1: Editor Tool - View Buildings.

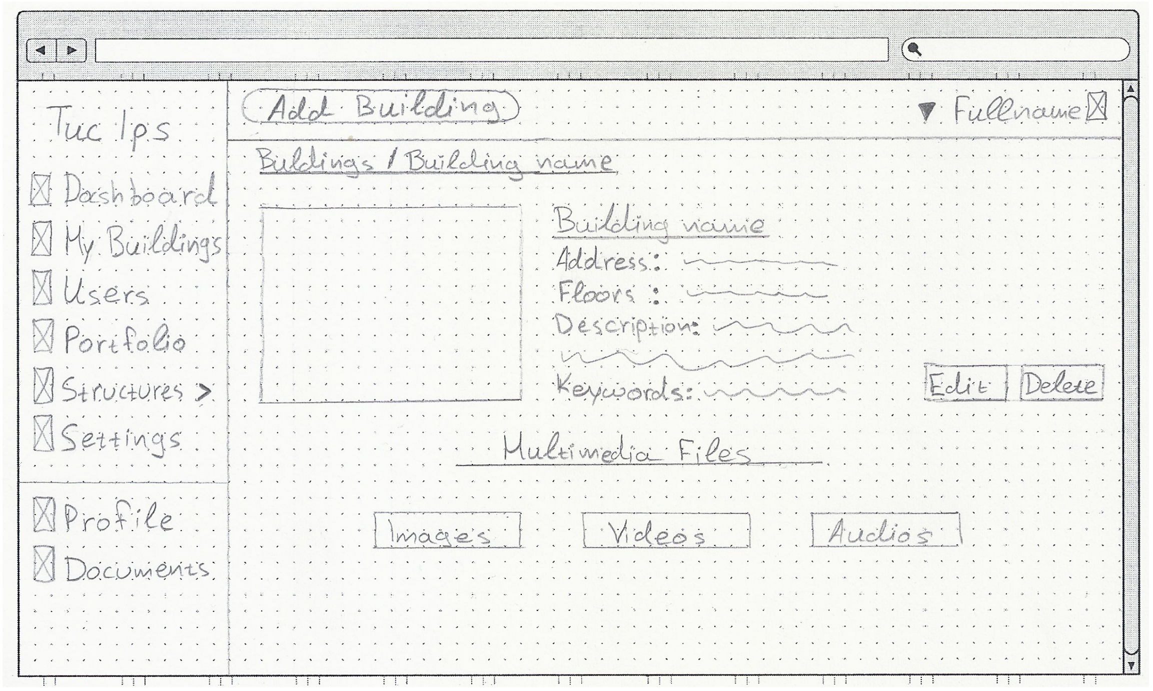


Figure 8.2: Editor Tool - View Building.

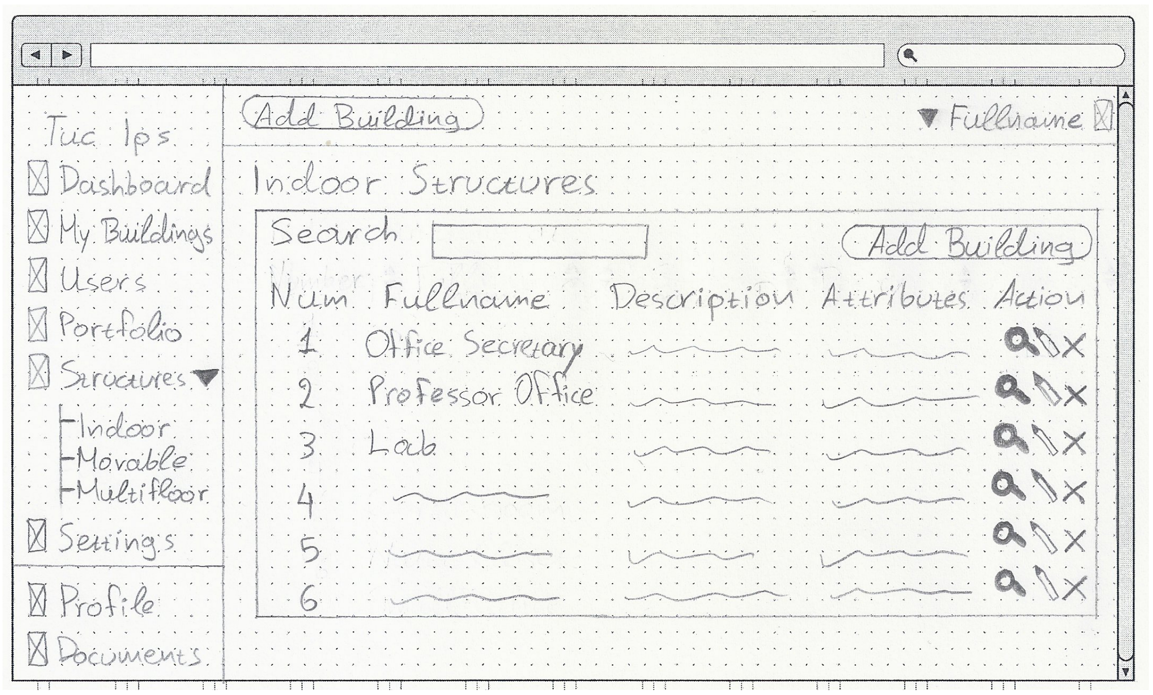


Figure 8.3: Editor Tool - View Structures.

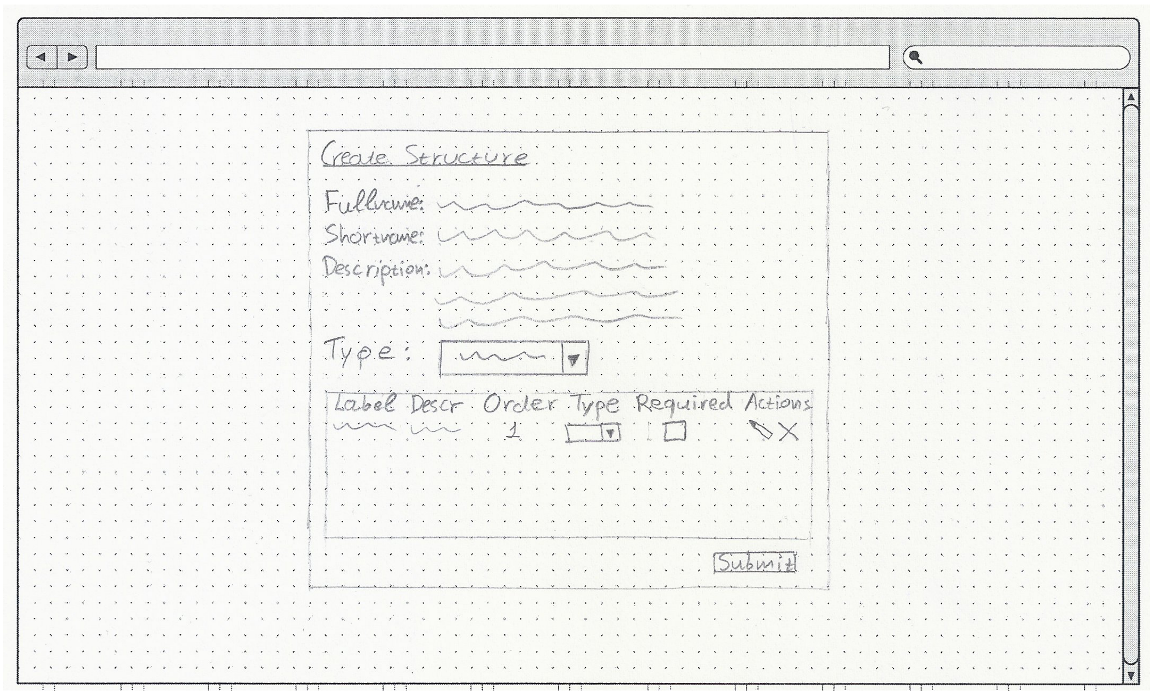


Figure 8.4: Editor Tool - Create Structure.

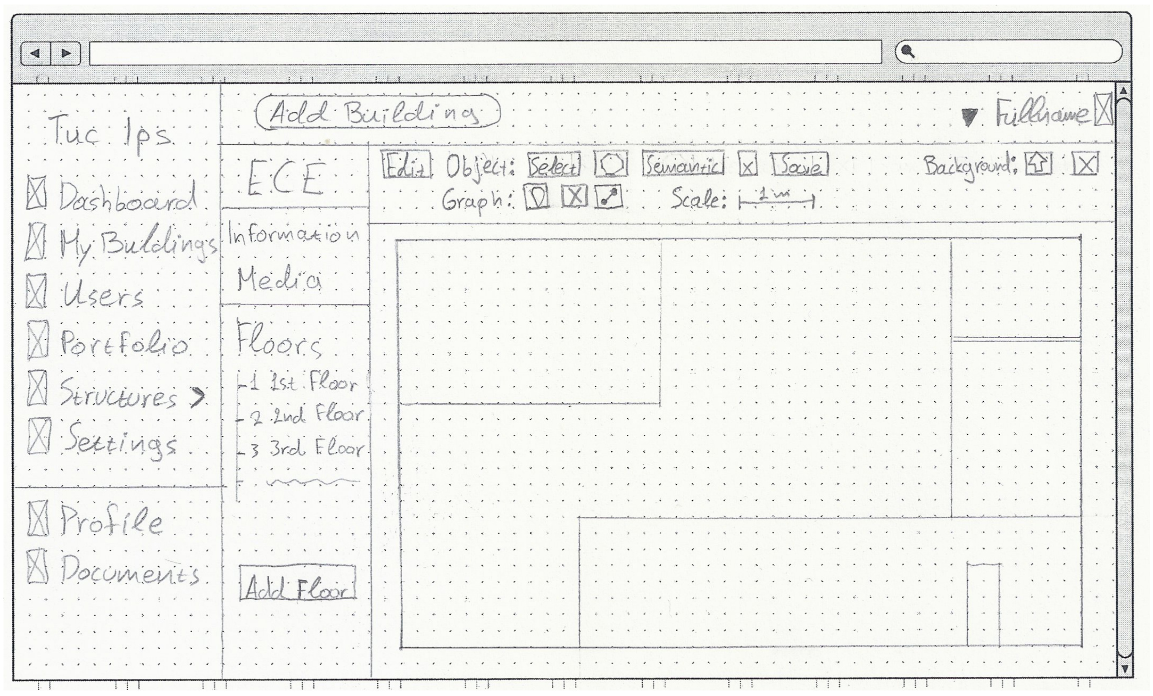


Figure 8.5: Editor Tool - Editor (View Plan).

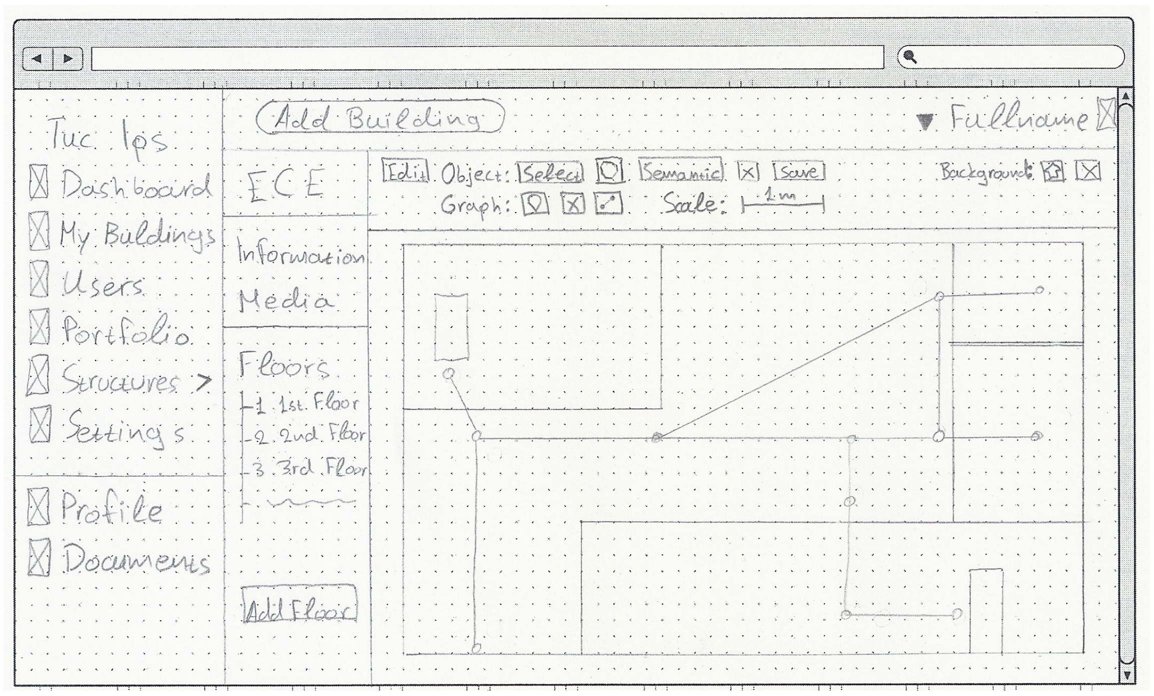


Figure 8.6: Editor Tool- Editor (View Nodes).

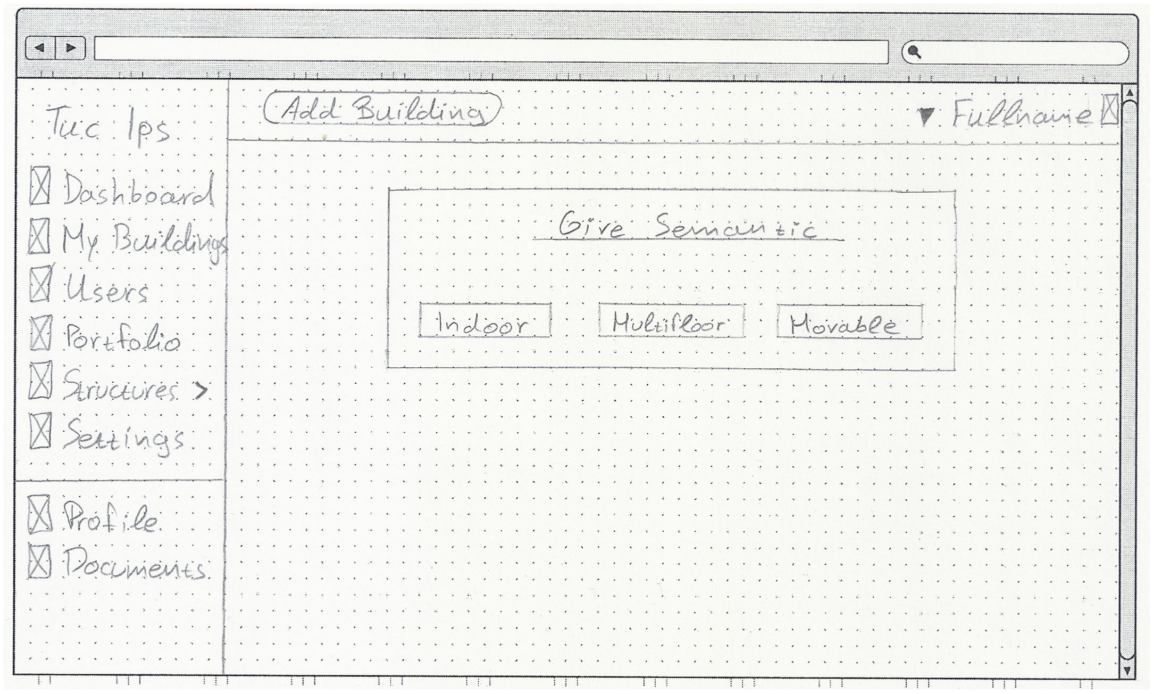


Figure 8.7: Editor Tool - Give Semantic Step 1.

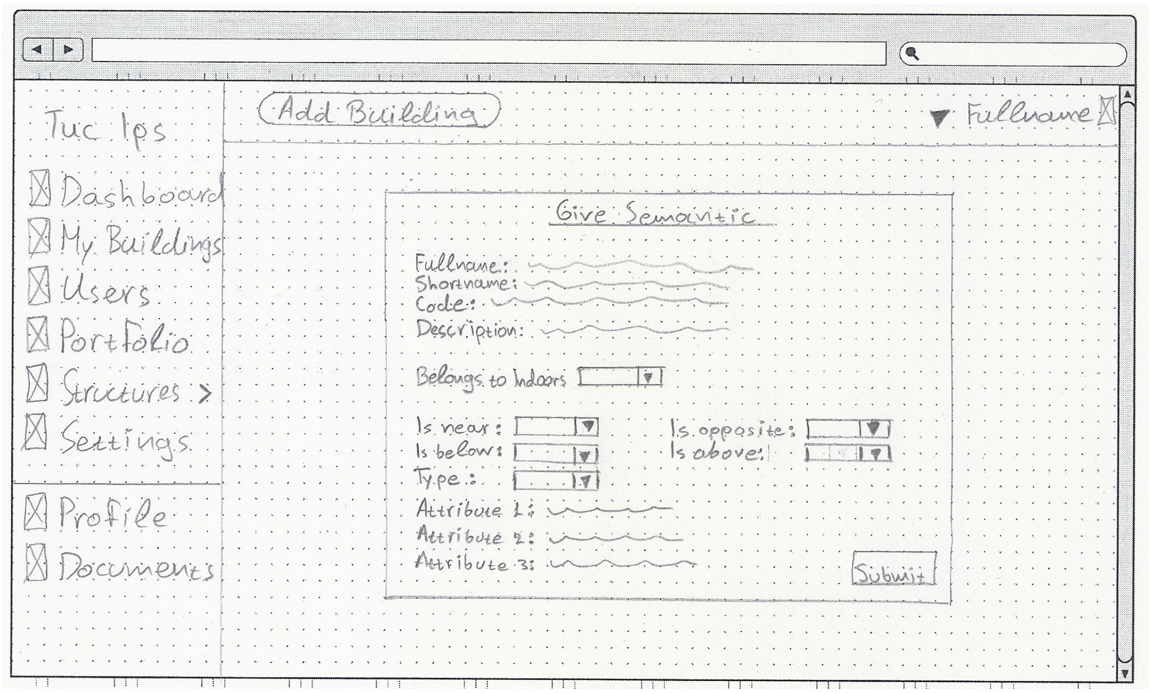


Figure 8.8: Editor Tool - Give Semantic Step 2.

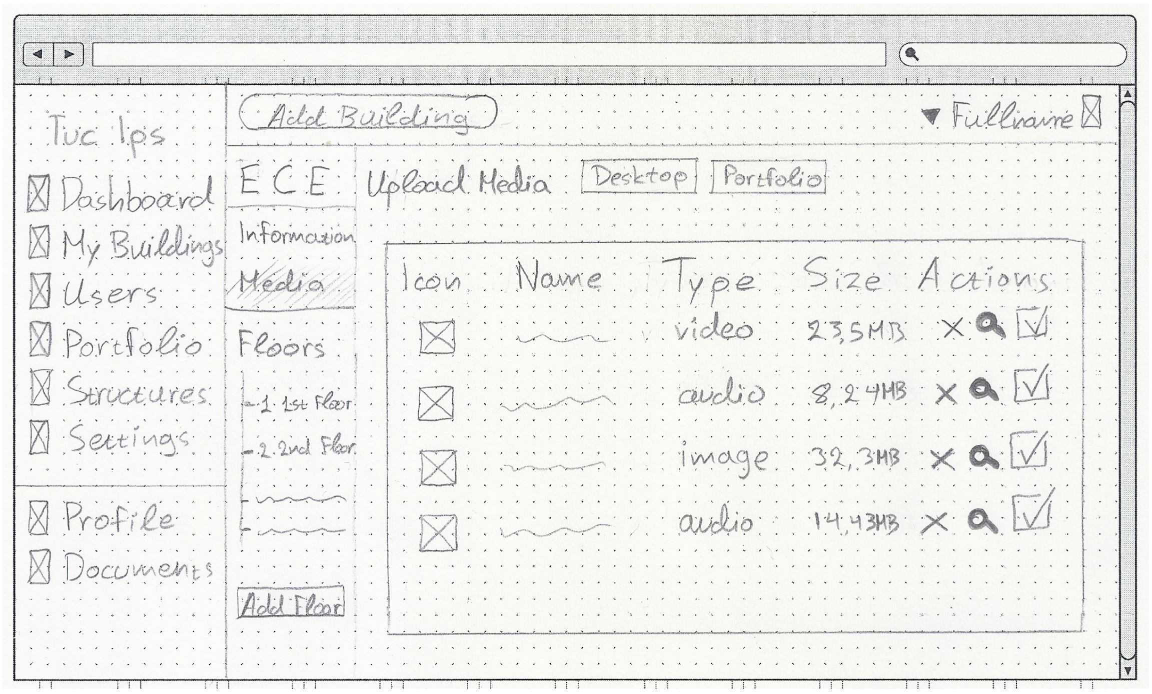


Figure 8.9: Editor Tool - Building Media.

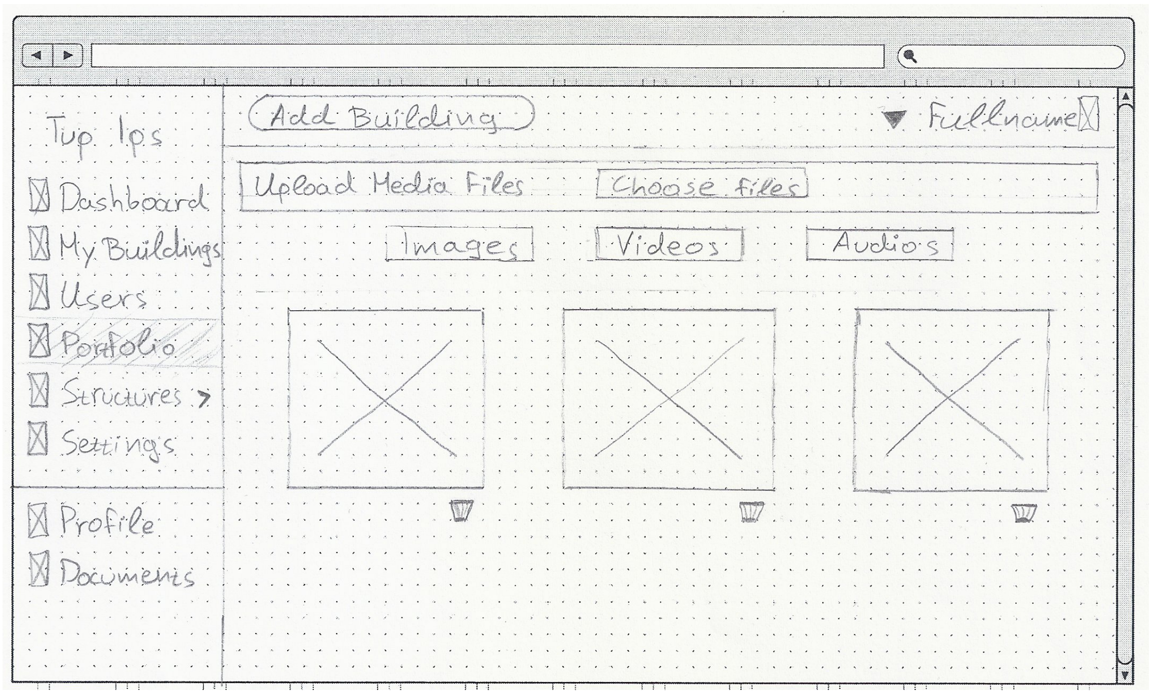
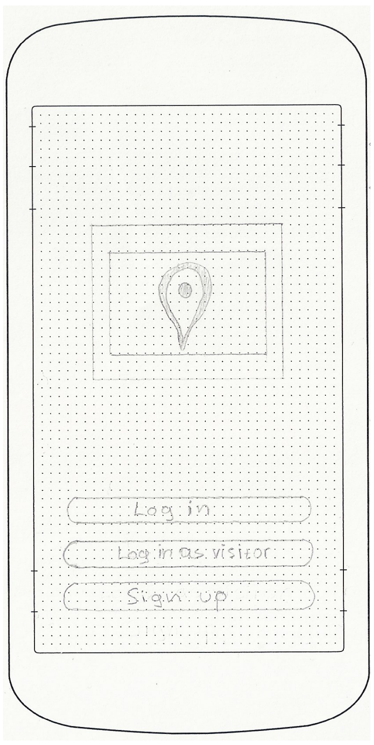


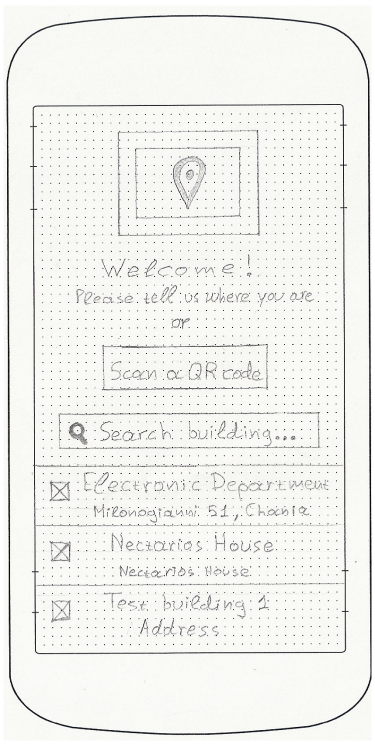
Figure 8.10: Editor Tool - Portfolio.

8.1.2 Mobile Application

The figures of this section present some of user interface prototypes of the mobile application which were drawn in the early stages of the design process. Each prototype has a title and the purpose that the user interface serves.

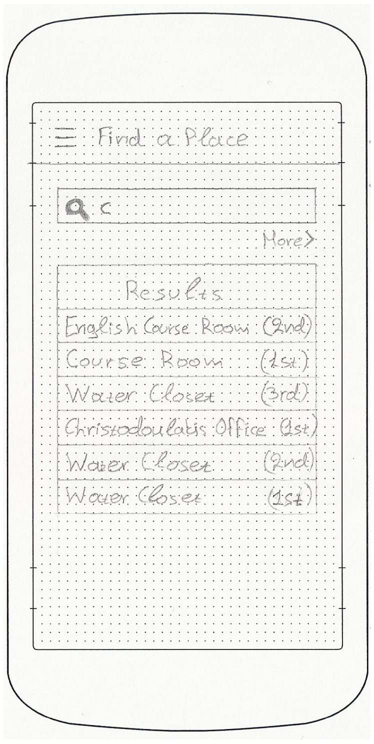


(a) Landing Page.

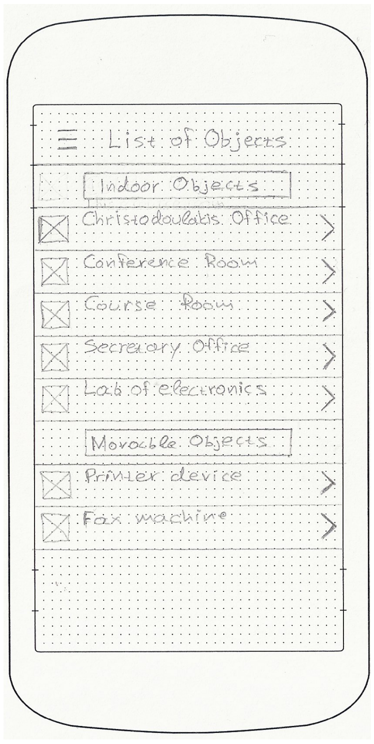


(b) Index Page.

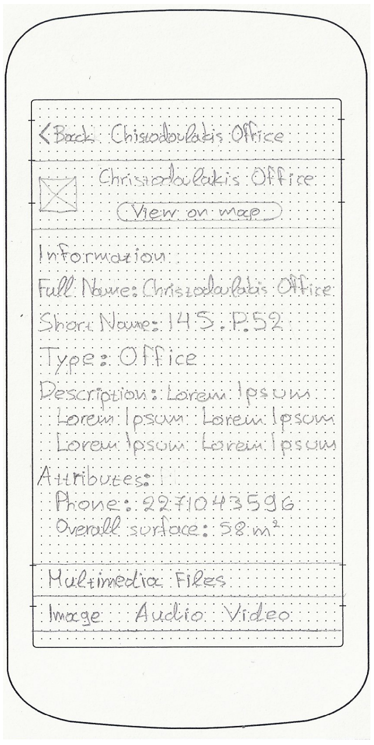
Figure 8.11: Mobile Application - User Interface Prototypes.



(a) Find a Place.

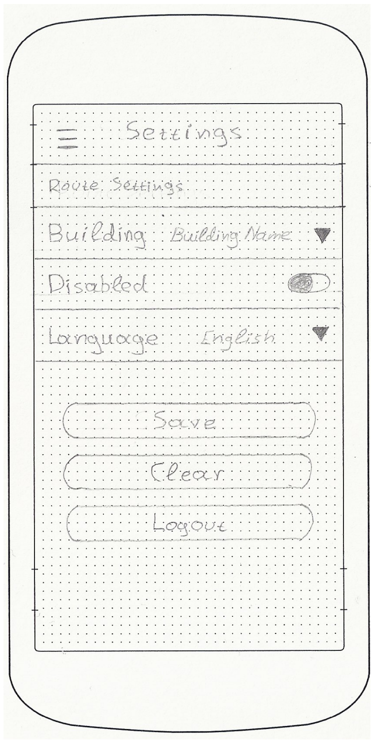


(b) List Of Objects.



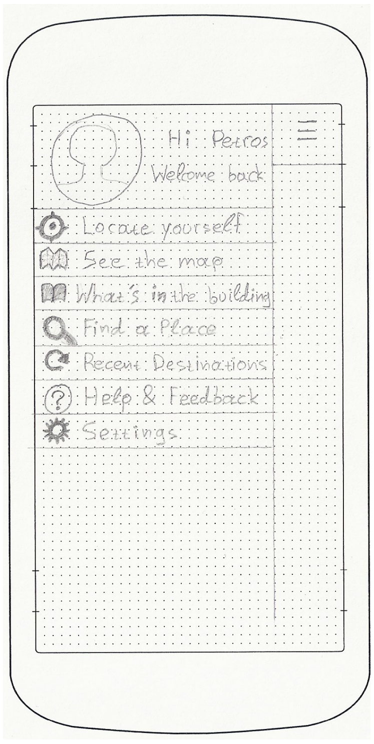
Information
Full Name: Christodoulakis Office
Short Name: 145.P.52
Type: Office
Description: Lorem Ipsum Lorem Ipsum Lorem Ipsum Lorem Ipsum Lorem Ipsum
Attributes:
Phone: 2271043596
Overall surface: 58 m²

(c) View Object.

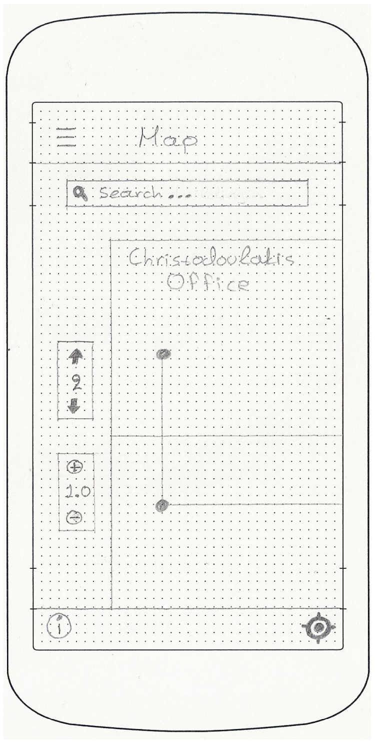


(d) Settings.

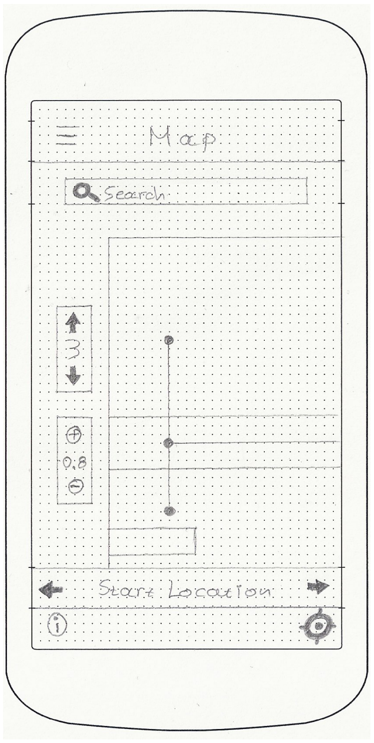
Figure 8.12: Player Application - User Interface Prototypes (cont'd).



(a) Side Menu.



(b) View Map.



(c) Map Instruction.

Figure 8.13: User Interface Prototypes.

8.2 Usability Evaluation

The following sections present the evaluation methods that have been applied in order to evaluate and improve usability of our applications. In more details, Section 8.2.1 describes the Pluralistic Walkthrough method, Section 8.2.2 describes the Heuristic Evaluation method and Section 8.2.3 describes the Think-Aloud method.

8.2.1 Pluralistic Walkthrough

The Pluralistic Walkthrough [3] is a usability inspection method used to identify usability issues in a piece of software or website in an effort to create a maximally usable human-computer interface. The method centers on using a group of users, developers and usability professionals to step through a task scenario, discussing usability issues associated with dialog elements involved in the scenario steps. The group of experts used is asked to assume the role of typical users in the testing. The method is prized for its ability to be utilized at the earliest design stages, enabling the resolution of usability issues quickly and early in the design process. The method also allows for the detection of a greater number of usability problems to be found at one time due to the interaction of multiple types of participants (users, developers and usability professionals). This type of usability inspection method has the additional objective of increasing developers sensitivity to users concerns about the product design.

Walkthrough Team

A walkthrough team must be assembled prior to the pluralistic walkthrough. Three types of participants are included in the walkthrough: representative users, product developers and human factors (usability) engineers/professionals. Users should be representative of the target audience, and are considered the primary participants in the usability evaluation. Product developers answer questions about design and suggest solutions to interface problems users have encountered. Human factors professionals usually serve as the facilitators and are also there to provide feedback on the design as well as recommend design improvements. The role of the facilitator is to guide users through tasks and facilitate collaboration between users and developers. It is best to avoid having a product developer assume the role of facilitator, as they can get defensive to criticism of their product. As it is rational we were not able to follow the above process but we tried to approach it through chose four representative users in order to take more accurate results. More specifically we chose two students from **School of Architecture**, one developer and the director of **Maritime Museum of Chios**, Miss Anna Sitara.

Procedure

We had printed our paper prototypes (as described in section 8.1) and put it together in packets in the same order that the screen would be displayed when the users were carrying out the specific tasks. This included hard copy panels of screens, dialog boxes, menus, etc. presented in order. Additionally, we gave to each participant a hard copy of the task scenario. There are several scenarios defined in this document complete with the data to be manipulated for the task. Each participant receives a package that enables him or her to write a response directly onto the page. The task descriptions for the participant are short direct statements.

Participants are given written instructions and rules at the beginning of the walkthrough session. The rules indicate to all participants (users, designers, usability engineers) to:

- Assume the role of the user.
- To write on the panels the actions they would take in pursuing the task at hand.
- To write any additional comments about the task.
- Not flip ahead to other panels until they are told to.
- To hold discussion on each panel until the facilitator decides to move on.

The selected tasks include the entire range of our applications (desktop and mobile). The tasks was performed by the users in Editor tool were the following:

1. Create account.
2. Browse existing buildings.
3. Preview a building.
4. Create a building.
5. Add Floor.
6. Draw a floor plan.
7. Add Object.
8. Add Node.
9. Preview all structures.
10. Add Structure.

The tasks were performed by the users in mobile application were the following:

1. Log in to the application.
2. Browse an existing building.
3. Find a place inside building.
4. Locate himself.
5. Take instruction for navigation to an object.

After the completion of this type of evaluation we gave a short questionnaire to the users in order to evaluate the interface and general usability and feel of our application. Although the difficulty of the tasks and the complexity of our application, the feedback was very hopeful, saying that it is handy especially if we corrected the identified problems and the carried on with the other evaluation types and that provides to user all the necessary functionality. The users pointed out that the lack of color on the interface did not help them, and that by adding it, the interface would become much better and even more convenient. We correct the identified problems and then initiated at the following evaluations.

8.2.2 Heuristic Evaluation

After we implemented user interfaces of our applications, based on the results from the previous phase of evaluation, we tried to make a heuristic evaluation. A heuristic evaluation[1] is a usability inspection method for computer software that helps to identify usability problems in the user interface design. It specifically involves evaluators examining the interface and judging its compliance with recognized usability principles.

These evaluation methods are now widely taught and practised in the new media sector, where UIs are often designed in a short space of time on a budget that may restrict the amount of money available to provide for other types of interface testing.

The main goal of heuristic evaluations is to identify any problems associated with the design of user interfaces. Usability consultant Jakob Nielsen developed this method on the basis of several years of experience in teaching and consulting about usability engineering.

Jakob Nielsen's heuristics are probably the most-used usability heuristics for user interface design. Nielsen developed the heuristics based on work together with Rolf Molich in 1990. The final set of heuristics that are still used today were released by Nielsen in 1994. The heuristics as published in Nielsen's book Usability Engineering[50] are as follows:

- *Visibility of system status*: The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

- *Match between system and the real world*: The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
- *User control and freedom*: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
- *Consistency and standards*: Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.
- *Error prevention*: Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.
- *Recognition rather than recall*: Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
- *Flexibility and efficiency of use*: Accelerators may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.
- *Aesthetic and minimalist design*: Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.
- *Help users recognize, diagnose, and recover from errors*: Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
- *Help and documentation*: Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

The simplicity of heuristic evaluation is beneficial at the early stages of design. Heuristic evaluation requires only one expert, reducing the complexity and expended time for evaluation and as it is rational this evaluation was made by us. It was held at the early stages of design and we examined the system twice using the rules of Nielsen. Firstly we tried to find out problems that naive users might encounter, as well as problems related to the conceptual model of our system. Then we examined the system having the user's primary goals, their ability targets and environmental parameters in mind.

8.2.3 Think Aloud

After the corrections of the heuristic evaluation we tried to do another kind of evaluation in order to take more accurate feedback about our applications. This time we tried to perform the same tasks on the real implemented system. The aim was to see how people react with our applications in real conditions and not over a paper without interactivity that has the electronic device. So we tried to evaluate our applications with the method of think aloud.

Think aloud protocols [5] consist of observing a user working with an interface while encouraging them to "think-aloud"; to say what they are thinking and wondering at each moment. Think-aloud protocols are of particular value because they focus on the problems a user has; when the user is working without difficulty, direct observation (and hence the think-aloud protocol itself) is of very limited use. This is because the user is unable to communicate as fast as they think and act, unless a specific problem arises which slows them down. It is at these times when this method really shines as it allows the observer to correlate the actions and statements of the participant. Test sessions are often audio and video recorded so that developers can go back and refer to what participants did and how they reacted. The purpose of this method is to make explicit what is implicitly present in subjects who are able to perform a specific task.

Procedure

In this stage we chose three different users, two of them were students of School of Architecture and one of them is the director of local super market of Chania. In this way we had a wide variety and representative user which are possible user in the future. According to the think aloud method we allowed the users unaided to follow all the steps needed to reach the completion of an action. We observe all of the user actions in order to see whether they make mistakes, when they are confused and how they are trying to correct their mistakes. Also a key requirement was that the user should tell us what he was thinking, and when he did not, we were trying to remind it to him. We recorded a video of what happens on the screen and the room, which enabled us to inspect and analyse the whole process more closely.

The results were impressive and the feedback was very hopeful. However, there were some problems which we tried to correct. After these correction, users were asked if the problems that they are encountered were solved and the results we got were very positive.

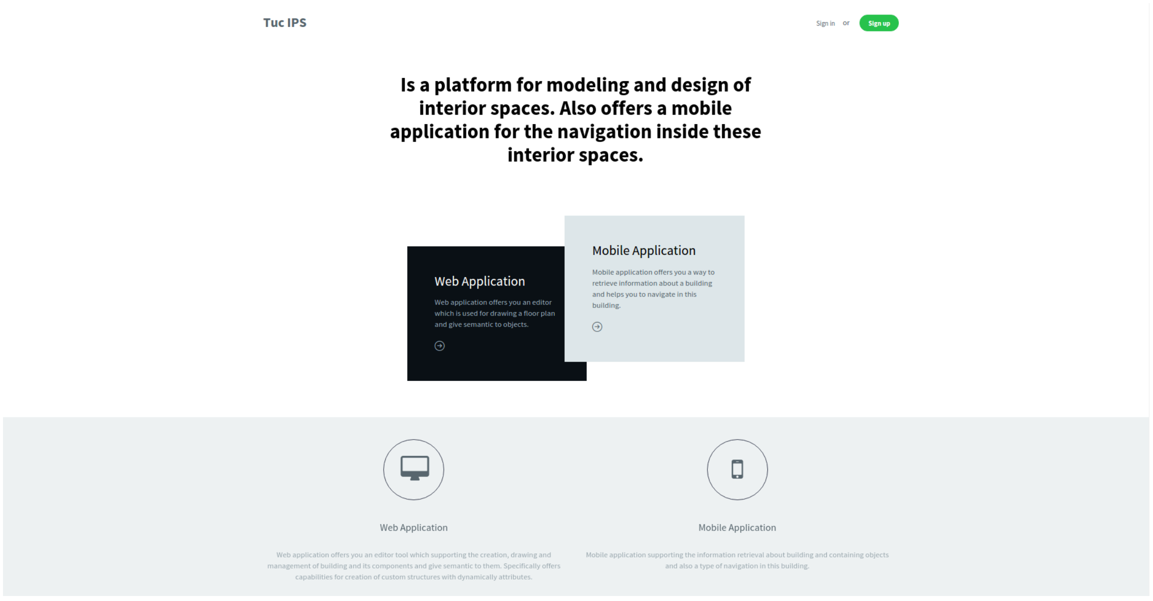
8.3 User Interfaces

The following section present some graphical user interfaces of our applications as they have been implemented in the final system. In more details, Section 8.3.1 presents user interfaces of our editor tool and Section 8.3.2 provides the user interfaces of our mobile application.

8.3.1 Editor Tool

Landing Page

When the users types our url and opens our application for the first time, the landing page (Figure 8.14) is presented which describes our objectives and capabilities which provided by our applications. The user also has the option to create a new account or login in order to get all of potential functionality.



Is a platform for modeling and design of interior spaces. Also offers a mobile application for the navigation inside these interior spaces.

Web Application

Web application offers you an editor which is used for drawing a floor plan and give semantic to objects.

Mobile Application

Mobile application offers you a way to retrieve information about a building and helps you to navigate in this building.

Web Application

Web application offers you an editor tool which supporting the creation, drawing and management of building and its components and give semantic to them. Specifically offers capabilities for creation of custom structures with dynamically attributes.

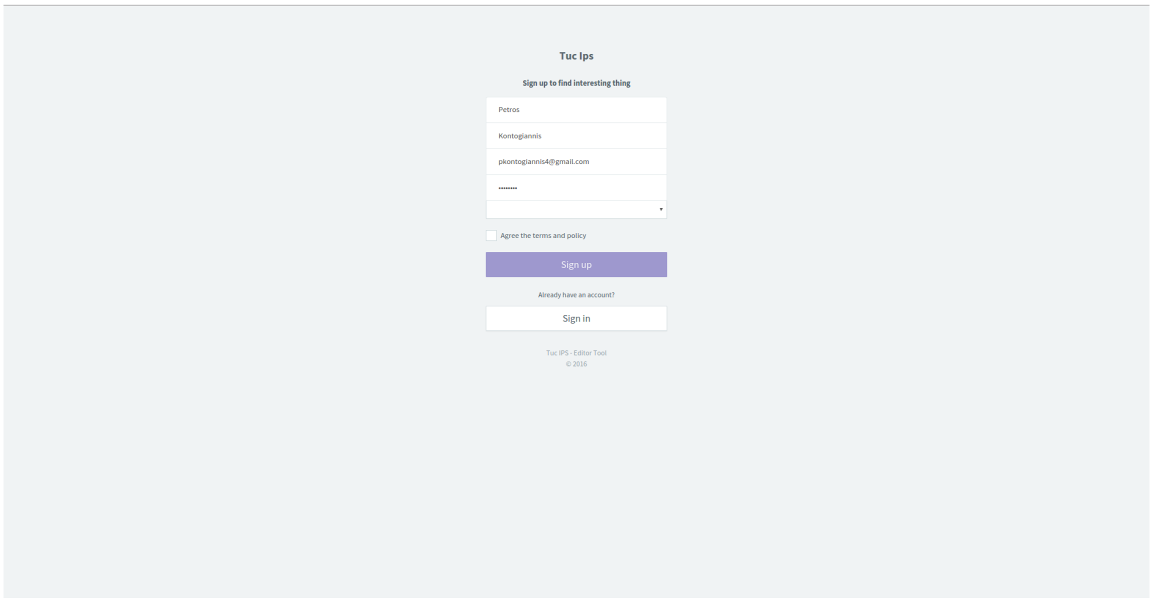
Mobile Application

Mobile application supporting the information retrieval about building and containing objects and also a type of navigation in this building.

Figure 8.14: Editor Tool - Landing.

Create Account

When the user selects the Signup button the screen (Figure 8.15) is displayed, and the system waits the user to type his personal information. The user has also the option to log in if he has already an account to our system.



Tuc Ips

Sign up to find interesting thing

Petros

Kontogiannis

pkontogiannis4@gmail.com

••••••••

Agree the terms and policy

Sign up

Already have an account?

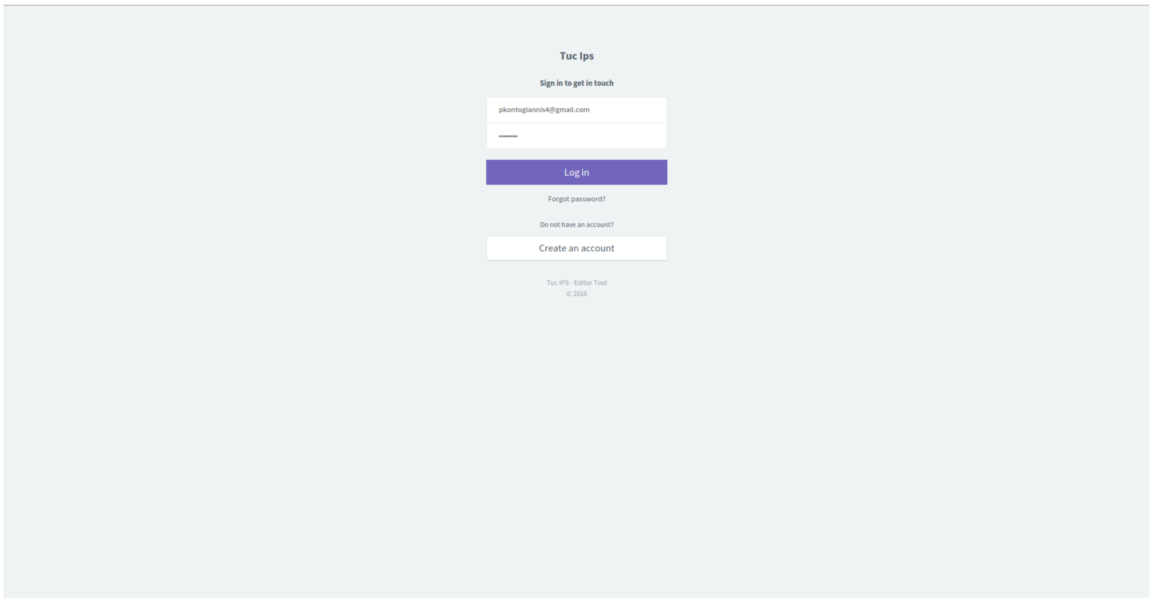
Sign in

Tuc IPS - Editor Tool
© 2016

Figure 8.15: Editor Tool - Create Account.

Log In

When the user selects the Sign In button the screen (Figure 8.16) is displayed, and the system waits the user to type his credentials. The user has also the option to create a new account, as well as to restore his password if he has forgotten it.



Tuc Ips

Sign in to get in touch

pkontogiannis4@gmail.com

••••••••

Log in

Forgot password?

Do not have an account?

Create an account

Tuc IPS - Editor Tool
© 2016

Figure 8.16: Editor Tool - Log in.

Home Page

After a successful login, the homepage(Figure 8.17) of our application is presented to the user's browser. The homepage is consisted of three main parts as we said in previous chapter. These parts is the top bar, the left side menu, and the main content of our application. The top bar and the left side menu are constantly presented to the user in order to take pleasure of the majority of application's capabilities in one step from every different page. The main content of our application is modified based on the page that the user wants to access. In the next paragraphs, we describe in more detail all the available options provided by these three main parts of the application.

Top Bar

- *Menu hide/show*: By selecting this option the user has ability to either hide or show the left side menu.
- *Profile button*: By selecting this option the left side menu presents some basic information regarding the user's profile.
- *Add Building*: This option is used for the creation of a new building from scratch.
- *Multilinguality*: Shows the current language of the graphical user interface and provides the option to change it using a dropdown menu.
- *Full screen*: This option is used for changing the presentation mode of the application to fullscreen.
- *Profile*: Presents the personal information and settings of the user that is currently logged in.

Left Side Menu

- *Dashboard*: Presents the homepage of the application.
- *My Buildings*: Presents the buildings which are created by the user.
- *Users*: Presents the users that have access in this application account, in order to contribute to the building creation and modification.
- *Portfolio*: Presents the personal multimedia object library of the user. Additionally, it offers the ability to upload new content and organize in the form of collections.
- *Structures*: Presents the structures which have been created through the application and provide several filtering options in order to enable better searching experience.

- *Profile*: Shows the user's profile page and allows the editing of information related to his account.
- *Help*: Provides information that explains how to use our application in order to perform certain tasks.

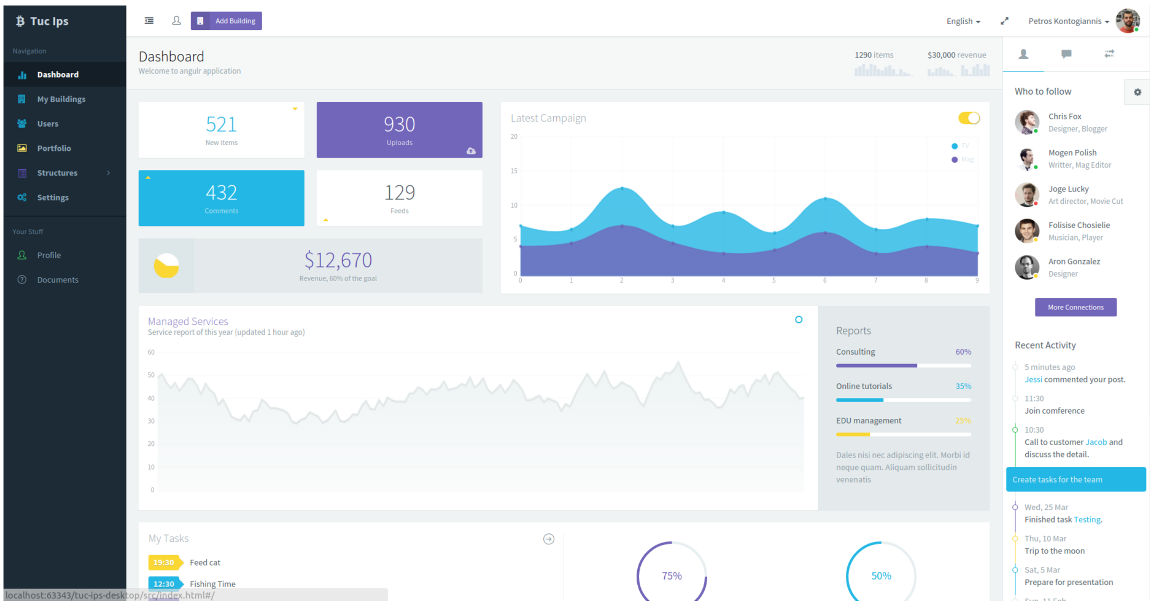


Figure 8.17: Editor Tool - Home.

Create Building

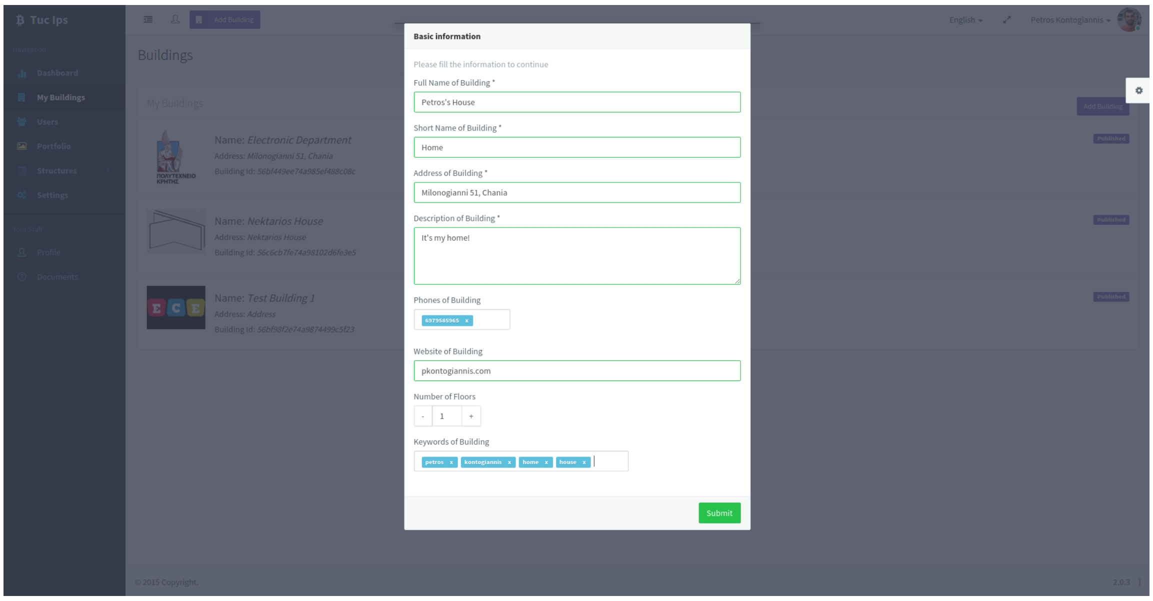
Figure 8.18 presents the user interface of our application when creating a new building from scratch. This is actually the first step in the building creation process and the user has to fill out some basic descriptive information about the building.

Create/Add Floor

Figure 8.19 presents the user interface that is displayed when creating/adding a new floor to an already existing building. The user fills some basic information about the floor on the pop-up window and after finishing his action the new floor is added to the building's floor list.

Draw Object

After creating a floor, the user should draw the plan of this floor using the editor tools. The Figure 8.20 presents screenshot of drawing process. The user select the polygon button and then draws


Basic information

Please fill the information to continue

Full Name of Building *
Petros's House

Short Name of Building *
Home

Address of Building *
Milonogianni 51, Chania

Description of Building *
It's my home!

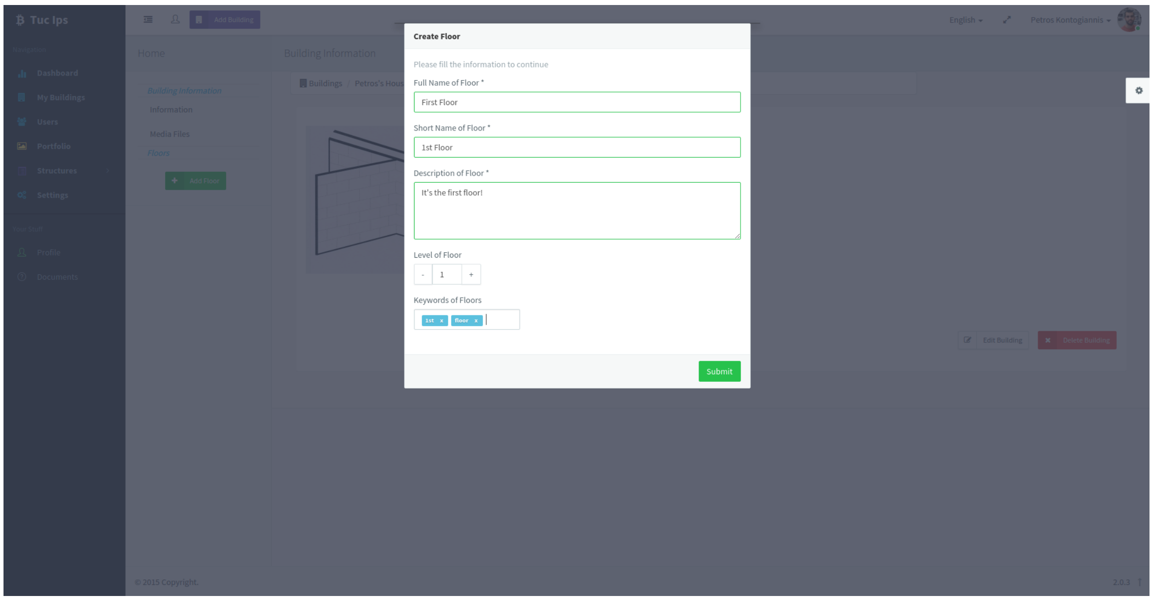
Phones of Building
6979585965

Website of Building
pkontogiannis.com

Number of Floors
1

Keywords of Building
petros, kontogiannis, home, house

Figure 8.18: Editor Tool - Create Building.


Create Floor

Please fill the information to continue

Full Name of Floor *
First Floor

Short Name of Floor *
1st Floor

Description of Floor *
It's the first floor!

Level of Floor
1

Keywords of Floors
1st, floor

Figure 8.19: Editor Tool - Add Floor.

his desired polygon. These polygons represents the containing objects of floor, such as rooms, stairs or movable objects.

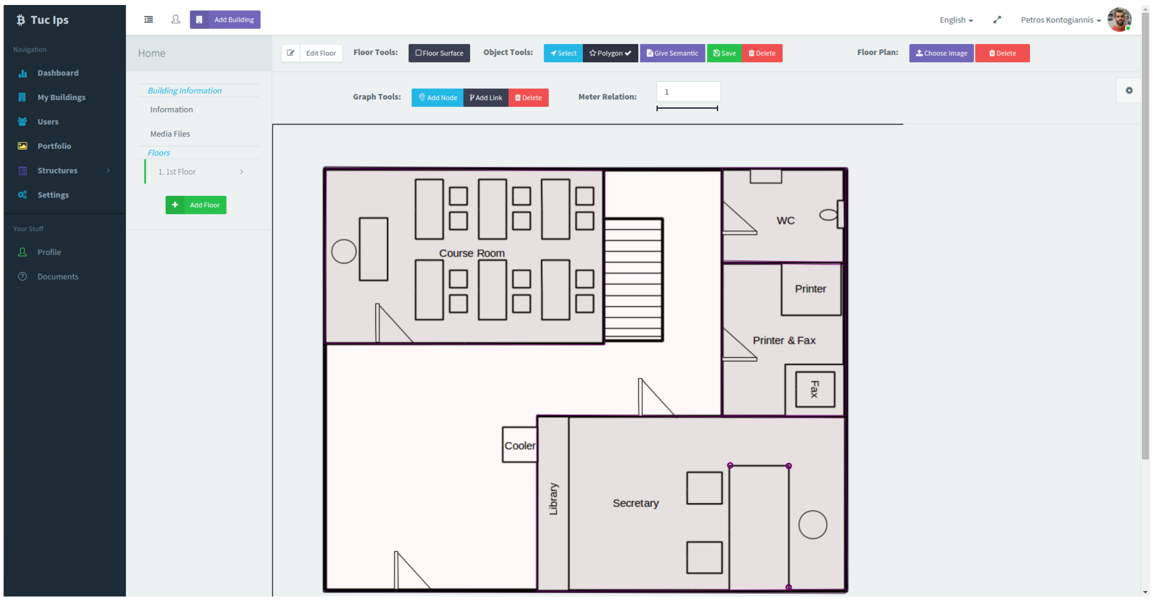


Figure 8.20: Editor Tool - Draw Object.

Give Semantic to an existing Object (2 screens)

After drawing a number of polygons, the user should describe them in order to give semantic to them. In this stage, the user selects the give semantic button and fills all the required and optional information in order to describe as well as possible the selected object. Information such as full name, short name, description etc. are presented below in Figure 8.21 and Figure 8.22.

Upload Media to an existing Object

Optionally the user is able to upload media files in order to give more representative information about the objects of building. In this screen (Figure 8.23) the user can upload three types of media files, such as images, audios or videos. After the uploading of these files we give a thumbnail of each file.

Add Node

For navigation purposes the user should create and set a complete graph which represent the possible paths inside the building. To do this, the user should to add multiples nodes, as shown in the

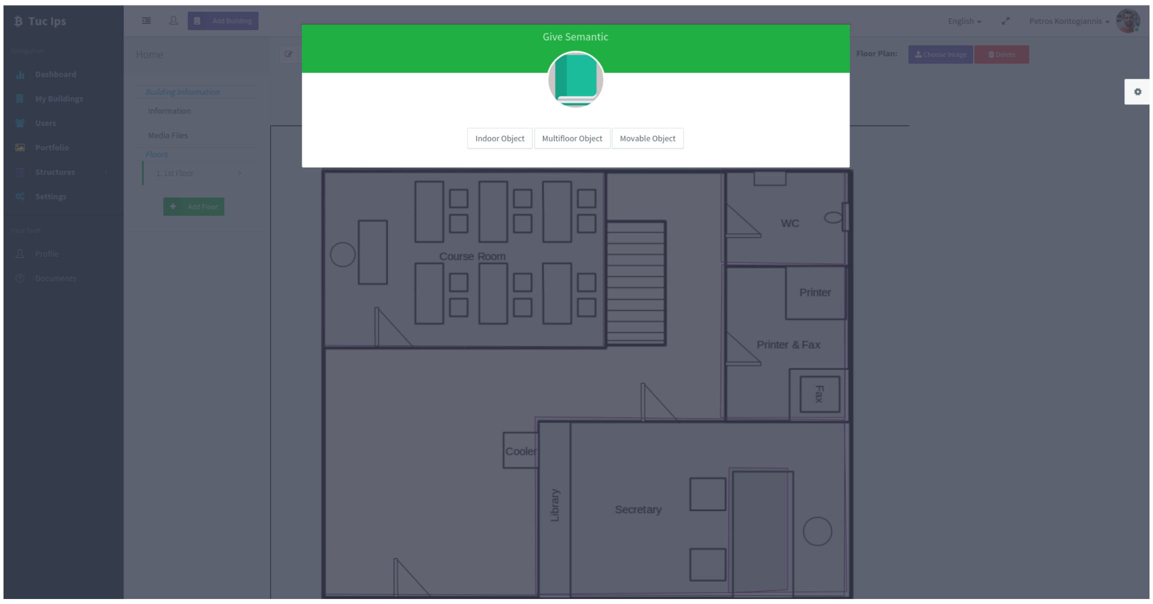
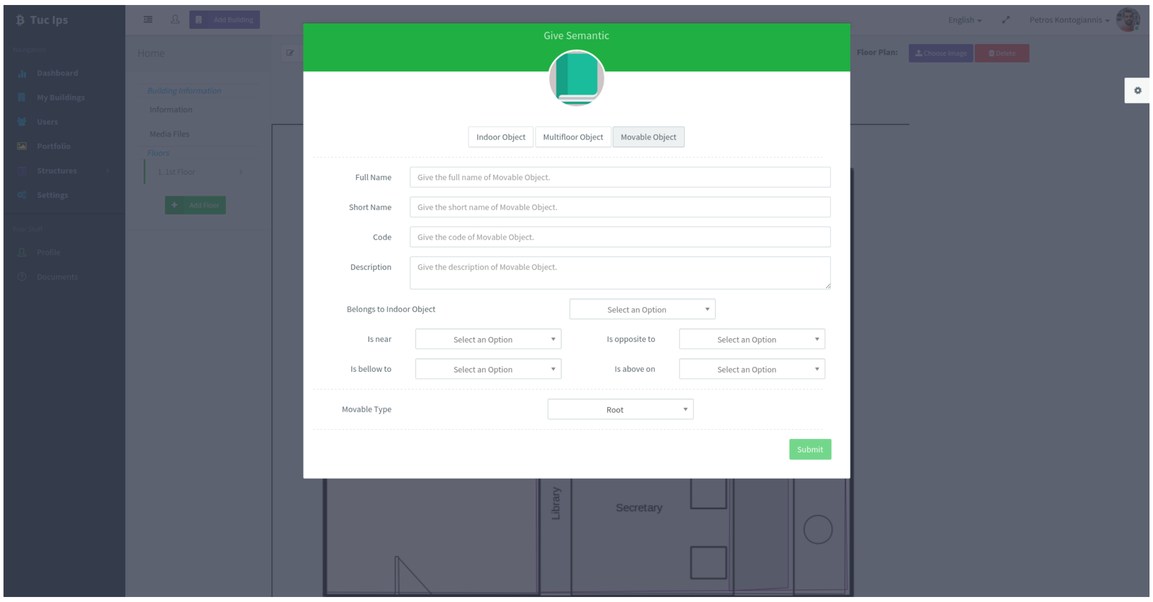
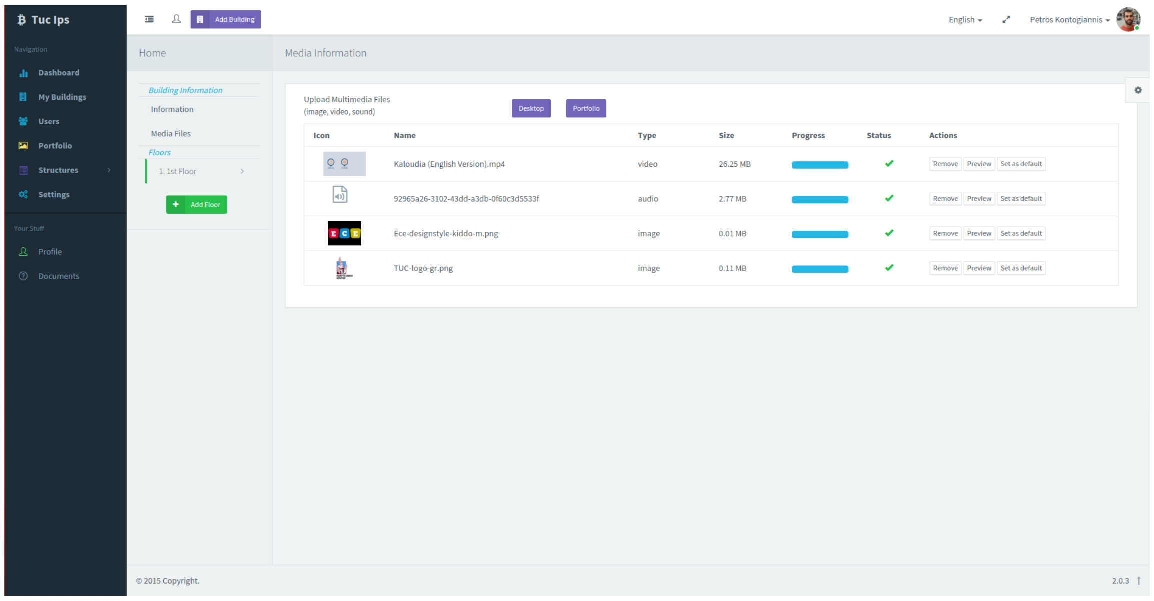


Figure 8.21: Editor Tool - Give Semantic - step 1.



Full Name: Give the full name of Movable Object.
Short Name: Give the short name of Movable Object.
Code: Give the code of Movable Object.
Description: Give the description of Movable Object.
Belongs to Indoor Object: Select an Option
Is near: Select an Option
Is opposite to: Select an Option
Is bellow to: Select an Option
Is above on: Select an Option
Movable Type: Root

Figure 8.22: Editor Tool - Give Semantic - step 2.



Upload Multimedia Files
(image, video, sound)

Icon	Name	Type	Size	Progress	Status	Actions
	Kaloudia (English Version).mp4	video	26.25 MB		✓	Remove Preview Set as default
	92965a26-3102-43dd-a3db-0f60c3d5533f	audio	2.77 MB		✓	Remove Preview Set as default
	Ece-designstyle-kiddo-m.png	image	0.01 MB		✓	Remove Preview Set as default
	TUC-logo-gr.png	image	0.11 MB		✓	Remove Preview Set as default

Figure 8.23: Editor Tool - Upload Media.

following figures(Figure 8.24 and Figure 8.25), around the floor and then he should connected to each other.

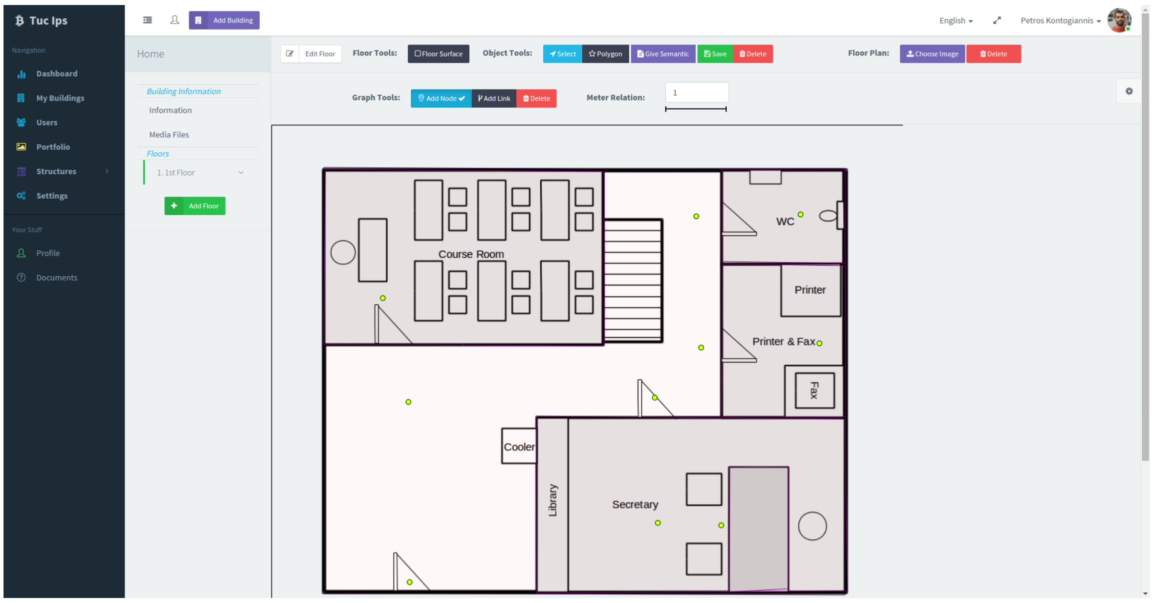
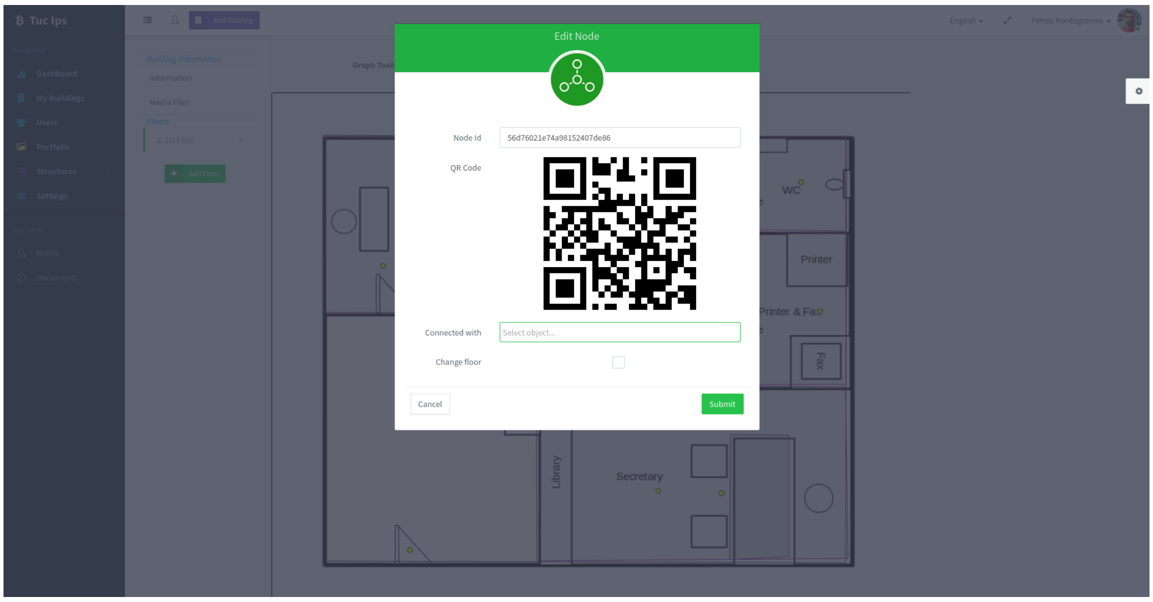


Figure 8.24: Editor Tool: Add Nodes.


Edit Node

Node Id: 56d76021e74a98152407de86

QR Code

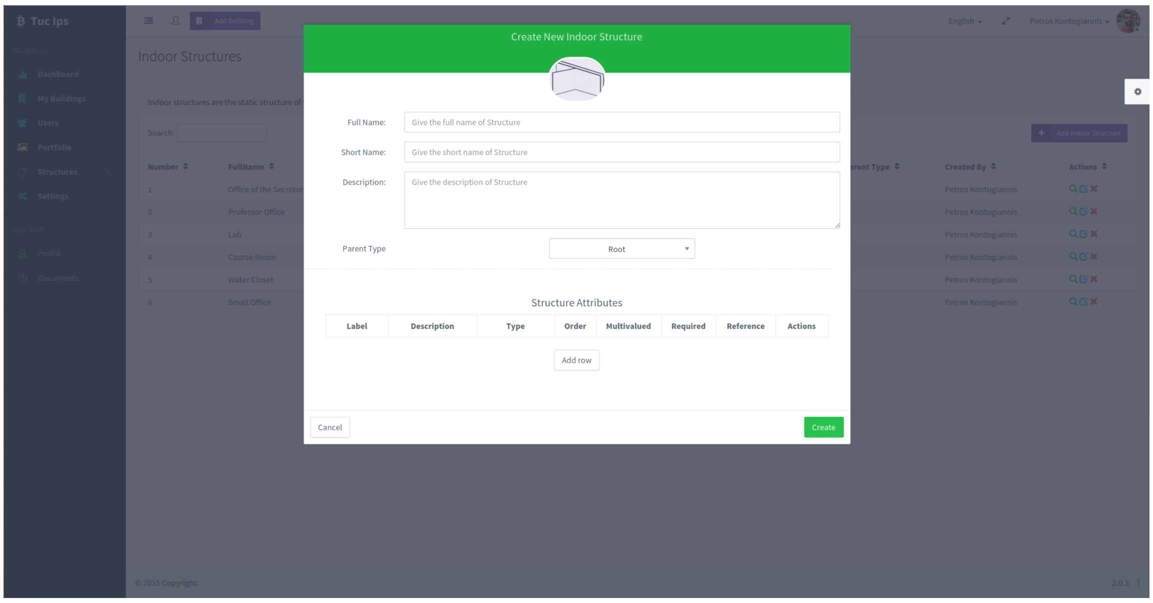
Connected with: Select object...

Change floor: ☐

Figure 8.25: Editor Tool: View Node.

Create Structure

Figure 8.26 presents a screenshot of the user interface that is displayed in case the user wants to create a new structure. In this stage the user has the ability to add the desired kind of structure, such as indoor, multi-floor or movable.


Create New Indoor Structure

Full Name: Give the full name of Structure

Short Name: Give the short name of Structure

Description: Give the description of Structure

Parent Type: Root

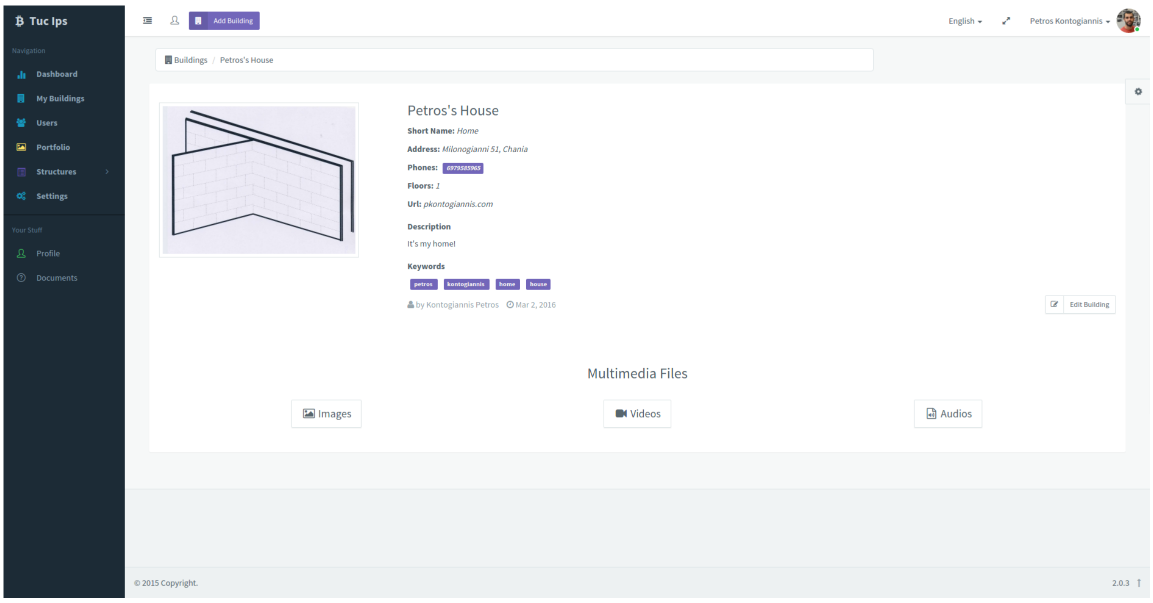
Structure Attributes

Label	Description	Type	Order	Multivalued	Required	Reference	Actions

Figure 8.26: Editor Tool: Create Structure.

Building

The presentation mode of an already created building is shown in the user interface screenshot of Figure 8.27. It provides an overview of the building's characteristic including the full name, description and other useful information.



Petros's House

Short Name: *Home*

Address: *Milonogianni 51, Chania*

Phones: *6979585965*

Floors: *1*

Url: *pkontogiannis.com*

Description

It's my home!

Keywords

petros kontogiannis home house

by Kontogiannis Petros Mar 2, 2016

Figure 8.27: Editor Tool: View Building.

Portfolio

The user interface of the user's personal multimedia object library is presented in Figure 8.28. Through this interface the user is able to upload new material, categorize the already existing multimedia objects and also filter them based on their type.

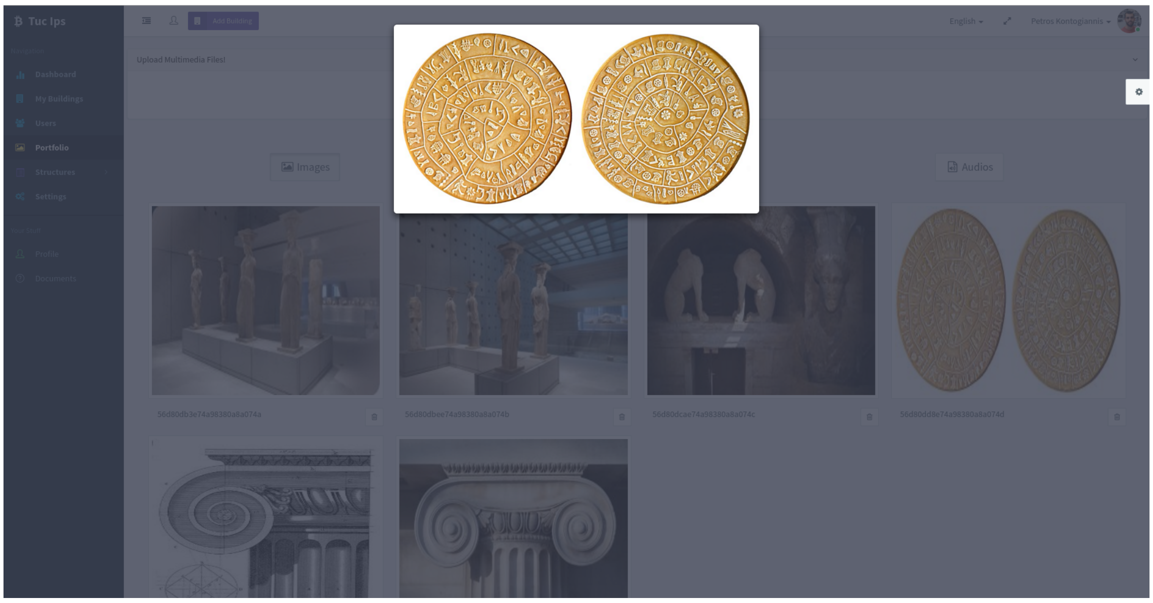


Figure 8.28: Editor Tool: View Portfolio.

8.3.2 Mobile Application

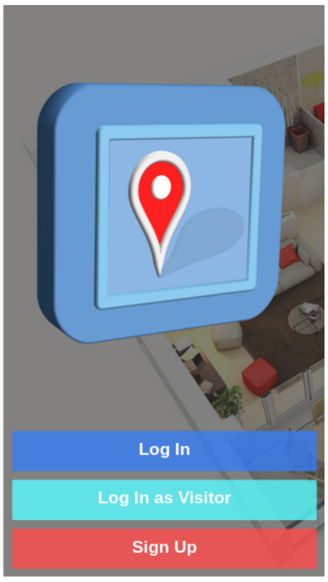
This section presents several screenshots of the mobile application's user interface that we implemented. Additionally, it provides some basic information and descriptions on how to perform certain actions.

Landing Page

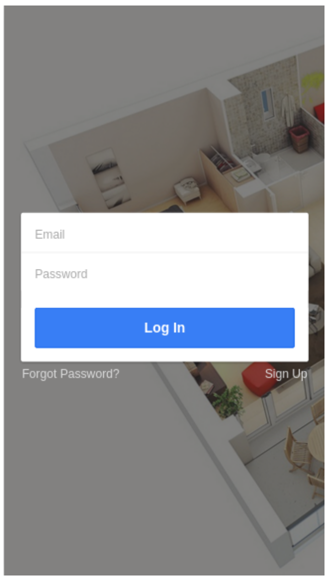
When the user visits a building and opens our application for the first time, we presented the Figure!8.29a below. In this screen give the ability to the user to log in if he already has an account, sign up or simple login as visitor.

Log In

When the user selects the Login button the screen (Figure 8.29b) is presented, prompting him to enter his credentials before continuing to the home screen of the application. The user also has the option to create a new account and there is option to restore his password if user has forgot it.



(a) Landing.

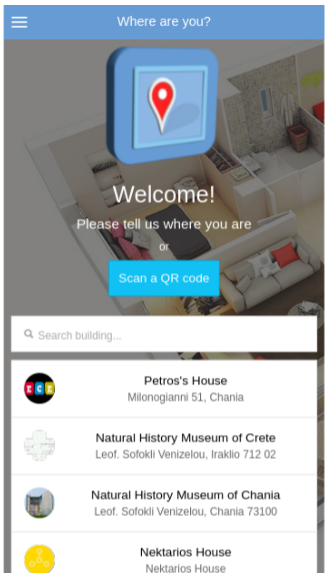


(b) Log in.

Figure 8.29: Mobile Application.

Find Building

The screenshot of Figure 8.30 is presented when the user opens our application for the first time, he is able to search and choose the desired building or simply scan a QR code in order to retrieve automatically his position.



Petros's House
Milonogianni 51, Chania

Natural History Museum of Crete
Leof. Sofokli Venizelou, Iraklio 712 02

Natural History Museum of Chania
Leof. Sofokli Venizelou, Chania 73100

Nektarios House
Nektarios House

(a) Welcome Page.

Figure 8.30: Mobile Application (cont'd).

Create Account

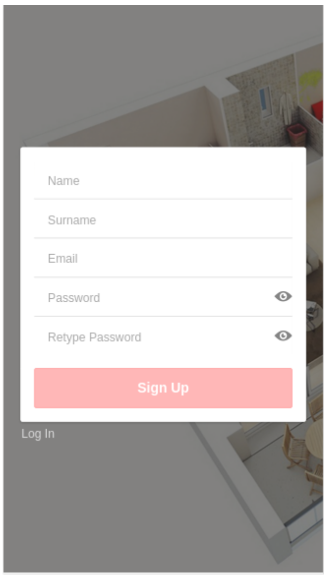
When the user selects the Signup button the screenshot of Figure 8.31a is presented in order to give the ability to the user signup to our application. The user also has the option to login, in case that he has already an account to our system.

Left Side Menu

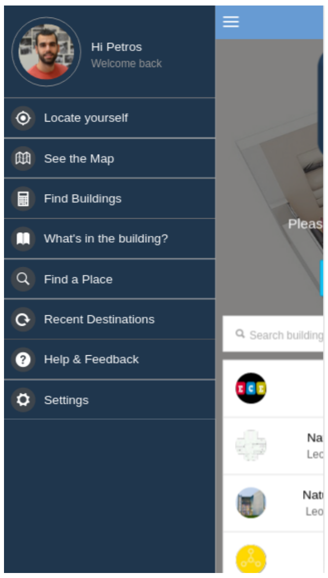
The left side menu of our mobile application is presented in Figure 8.31b. The available options of this menu are presented below along with some basic description.

- *Profile*: Presents the personal information and settings of the user that is currently logged in.
- *Locate Yourself*: Give the ability to the user to user his camera device in order to scan a QR code that is located near to him.
- *See the Map*: Presents the current position of the user on the appropriate floor map.
- *What's in the Building*: Shows a list with all building containing objects.
- *Find a Place*: Through this screen the user is able to search and retrieve a place inside the building.

- *Recent Destination*: In this screen are presented the recent destination of the user.
- *Help and Feedback*: Presents some information about the way that user can use our application.
- *Settings*: Shows the application's settings.



(a) Create account.



(b) Left side menu.

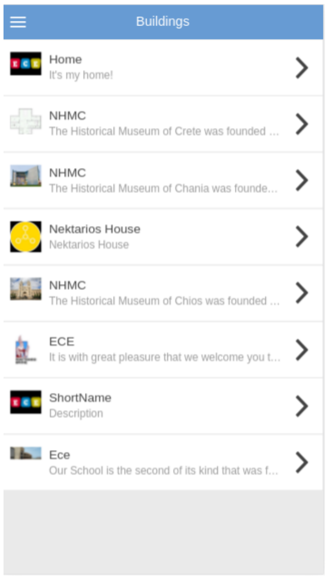
Figure 8.31: Mobile Application (cont'd).

All Buildings

Figure 8.32a presents a screenshot of the available buildings in our application. The user has the option to inspect a specific building by simply clicking on it. By swiping left, the user is able to see the left side menu of our application.

View Building

In this screen (Figure 8.32b) the user is able to see information about the selected building. In this user interface the user is able to inspect some descriptive information about the building and its containing objects.


Buildings

Home
It's my home!

NHMC
The Historical Museum of Crete was founded ...

NHMC
The Historical Museum of Chania was founde...

Nektarios House
Nektarios House

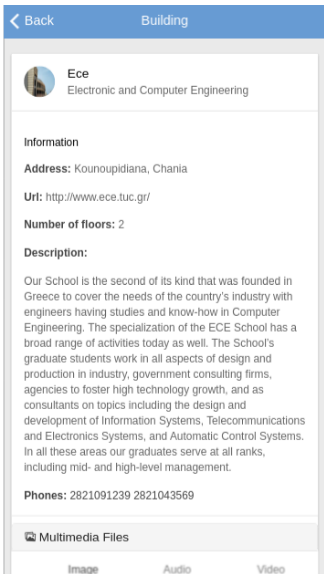
NHMC
The Historical Museum of Chios was founded ...

ECE
It is with great pleasure that we welcome you t...

ShortName
Description

Ece
Our School is the second of its kind that was f...

(a) All buildings.


Back Building

Ece
Electronic and Computer Engineering

Information

Address: Kounoupidiana, Chania

Url: http://www.ece.tuc.gr/

Number of floors: 2

Description:

Our School is the second of its kind that was founded in Greece to cover the needs of the country's industry with engineers having studies and know-how in Computer Engineering. The specialization of the ECE School has a broad range of activities today as well. The School's graduate students work in all aspects of design and production in industry, government consulting firms, agencies to foster high technology growth, and as consultants on topics including the design and development of Information Systems, Telecommunications and Electronics Systems, and Automatic Control Systems. In all these areas our graduates serve at all ranks, including mid- and high-level management.

Phones: 2821091239 2821043569

Multimedia Files

Image Audio Video

(b) Building.

Figure 8.32: Player Application (cont'd).

Settings

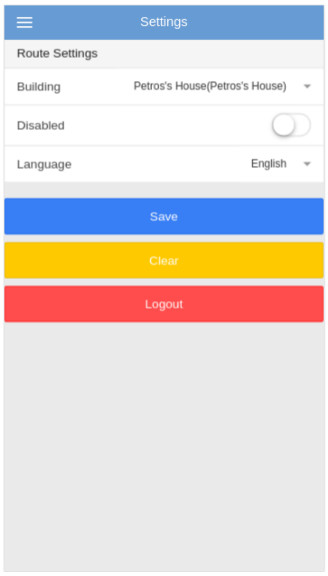
Figure 8.33a presents a screenshot of the settings of our application. The user is able to select a building as default in order to see information about the selected building. Additionally he has the option to select if he is person with special needs in order to take into consideration through generation of instruction process. Finally he has the choice to select the desired language of our application.

See the Map

Figure 8.34 presents a screenshot of a floor map of specific floor. In this screen the user has the option to locate himself inside the building or simple see information about the floor's objects. Additionally, he is able to click on these objects in order to retrieve more information about the selected object, either is a movable object or multifloor and indoor.

Take Instructions

Figure 8.35 presents two of stages which are presented when the user selects to navigate to an object. In the bottom of the map are presented the appropriate instructions in order to navigate the user to his destination. With clicking on the arrows the user is able to retrieve the next command.



Settings

Route Settings

Building Petros's House(Petros's House)

Disabled

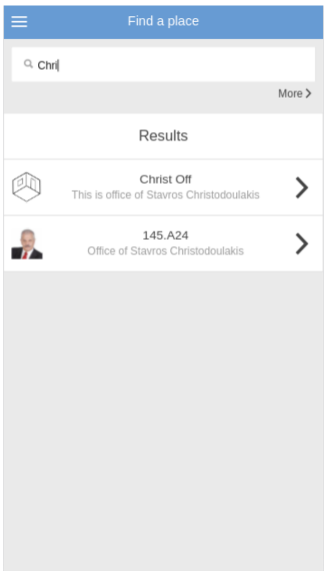
Language English

Save

Clear

Logout

(a) Settings.



Find a place

Chri

More

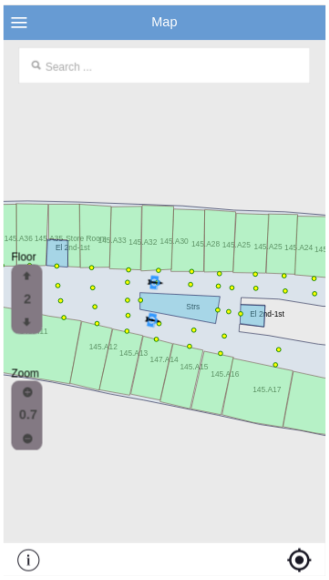
Results

Christ Off
This is office of Stavros Christodoulakis

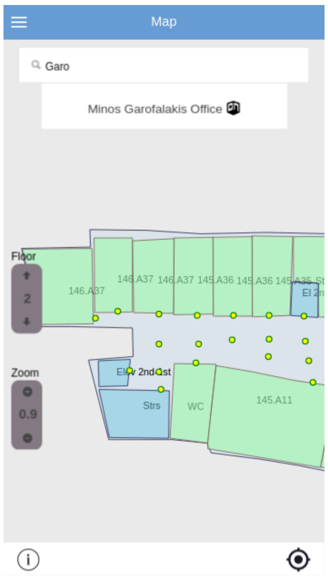
145.A24
Office of Stavros Christodoulakis

(b) Find Place.

Figure 8.33: Mobile Application (cont'd).

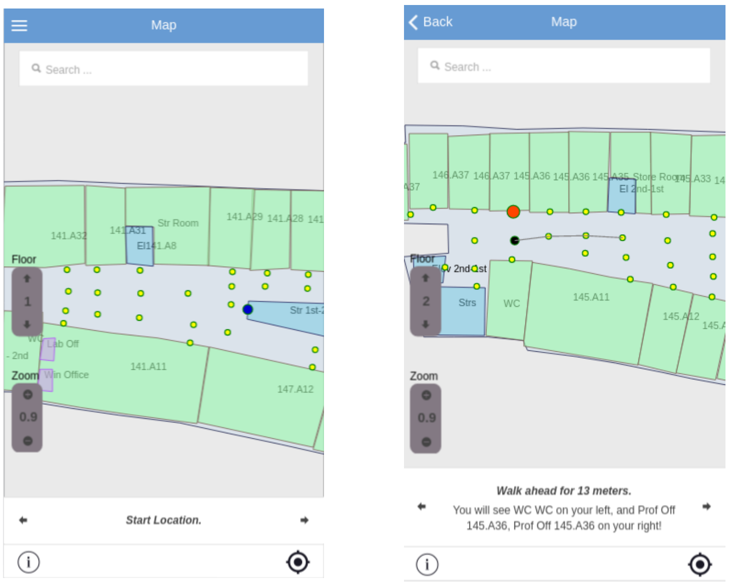


(a) See the Map.



(b) See the Map - Search.

Figure 8.34: Mobile Application (cont'd).



Start Location.

Walk ahead for 13 meters.

You will see WC WC on your left, and Prof Off 145.A36, Prof Off 145.A36 on your right!

(a) Start Location. **(b)** Command

Figure 8.35: Mobile Application.

Chapter 9

Conclusion & Future Work

In this thesis we presented the design and implementation of a platform for creating, designing, annotating content and management of interior spaces, as well the navigation inside them. Our main objectives were to: (*a*) support an easy and fast way to create and design the map of building, (*b*) support a wide variety of possible buildings through the freedom which characterize our model, (*c*) familiarize visitors of building with created interior spaces, and (*d*) provides instructions to the user in order to navigate inside a building.

More specifically, we designed and implemented a platform that consists of two applications, an editor tool and a mobile application. Through the editor tool, we supported the creation, design, object's annotation, and management of internal spaces of building, including building's objects editing. We tried to give the ability to everyone to create and design whichever kind of building he wants, such as museum, supermarkets, retails, hotels etc. On the other side, we developed and implemented a mobile application supporting the searching and the navigation of a user inside the created building in order to familiarize with a new building. The mobile application is compatible with most state-of-the-art mobile devices/operations, while both tools have been designed with flexibility and extensibility in mind. The underlying model supports the creation and design of great variety of building in order to create a great range of buildings that a user is able to navigate inside and retrieve their content. To this end, our tools can be used in various domains including museum, supermarkets, stadiums, hotels, ships etc.

Both the editor tool and mobile application have been evaluated for their usability though the user of three types of evaluation (heuristic evaluation, pluralistic walkthrough evaluation and think aloud evaluation). Finally, it is worth mentioning that we used state-of-the-art technologies to implement our system in order to be compatible with all modern platforms.

It is a fact that we want to continue the work that we developed in this thesis and we have many upgrades and extensions in our mind in order to give a more complete system to our users. More specifically our future work will include multiple steps in order to extend our functionality and system's quality. Firstly we want to improve the procedure during the creation and design of the floor plan. In the current version the user draws the floor plans from scratch using an editor, but it would be better if this procedure is shorter. For example it would be very useful to retrieve the desired floor plan from an uploaded image using image recognition algorithms or construct an algorithm which transforms a design that is produced using CAD tools, in order to avoid the drawing of floor plan. Furthermore, we want to work on the generation of instruction about the navigation to the user and take advantage of all our data about the objects in order to generate instruction which are connected with object inside building. It is a undeniable fact that in the current supported model we save extended information for our objects, such as if an object is above, below, opposite to another object. Regarding navigation instructions, our goal is to give to the form of voice instructions in order to give the user the ability not to see the mobile device and be able to navigate inside building only using these instructions. Another point that we want to focus on is to implement our mobile application using native language in order to take pleasure of every feature that is offered by mobile devices and improve experience of our users. It will be interesting to use wireless or bluetooth technologies in order to track the mobile position in real time and generate instruction on the fly.

Besides them, we want to give more features to creators of building. For example we think that it is very useful for the museum staff to give them the ability to create a tour inside the building. In this way a visitor is able to take part in scenario in order to see the whole museum according to the museum staff guide. Also we think that it is a significant improvement to give the ability to the administrator to manage and watch the interior spaces from another aspect. Specifically, it is so important for administrators to know how many people are inside a room or how visitors move inside building, or even create and organize events inside a reserved room.

In case of museum building, it may be useful for visitors to give recommendations about the visitors exhibits in order to improve their sightseeing tour inside the museum, using their attitude towards the objects which they have already seen. For example, if someone stands in front of an exhibit for some time, is possible to be interested in these category of exhibits.

Additionally, in the near future we want to extend our model in order to connect an interior space with a person who has a specific role (professor, lab director, staff member), for example a professor with his office or his lab.

The routing system currently uses Dijkstra's algorithm to find the shortest path between two points. Using different algorithms, for example A*, can decrease the time needed to calculate a route. Route calculation could also be performed on the device to decrease the reliance on a internet connection. Especially, a visitor might not always have the needed credentials to use the Wi-Fi system or even be aware of it's existence. The reception from the cell phone network is not always reliable and especially in underground levels unreliable at best.

Finally, we want to include in our model, connectivity with existing systems which have been developed in our lab, such as educational games in case of student visits. People may discover objects that enable them to proceed in the next step, they may get as a result hints on where the next object to be discovered is, and eventually find the winning object which gives you the price. Through the organization of educational games, we are able to familiarize children with a new building and discover the desired objects.

In the long term, it is desirable to extend our application to another sector. It would be impressive for us, if we were able to provide a 3D view of the interior. In this way, we could take off the user experience and it would be too easy for the user to navigate inside the building. The second sector that we are able to extend our application is the augmented reality. It is a too different sector but it would be fantastic to enrich our application with this feature.

As you can understand, our main goal is to continue the development of this thesis in order to create a platform that can be used for an extended variety of buildings through various ways.

Bibliography

[1] Heuristic Evaluation. https://en.wikipedia.org/wiki/Heuristic_evaluation.

[2] Ionic Framework. http://ionicframework.com/docs/.

[3] Pluralistic walkthrough. https://en.wikipedia.org/wiki/Pluralistic_walkthrough.

[4] Spring RESTful Services. http://spring.io/guides/gs/rest-service/.

[5] Think aloud protocol. https://en.wikipedia.org/wiki/Think_aloud_protocol.

[6] 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.

[7] F. Aebi. *Autonomous Indoor Navigation*. Msc thesis, University of Fribourg (Switzerland), 2012.

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

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

[10] AngularJS. https://docs.angularjs.org/guide.

[11] S. Anjum. *Place Recognition for Indoor Blind Navigation*. Msc thesis, Carnegie Mellon University, 2010.

[12] D. M. Applications. https://msdn.microsoft.com/en-us/library/ee658108.aspx.

[13] A. Bachmeier. *Wi-Fi based indoor navigation in the context of mobile services*. Msc thesis, University of Fribourg (Switzerland), 2013.

[14] J. Behr, Y. Jung, J. Keil, T. Drevensek, M. Zoellner, P. Eschler, and D. Fellner. A Scalable Architecture for the HTML5/ X3D Integration Model X3DOM. 2010.

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

[16] I. Biederman. Recognition-by-Components: A Theory of Human Image Understanding . 1987.

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

[18] A. Chandgadkar. *An Indoor Navigation System For Smartphones*. Msc thesis, Imperial College London, 2013.

[19] 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.

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

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

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

[23] 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.

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

[25] 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.

[26] H. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. 2011.

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

[28] 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].

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

[30] 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.

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

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

[33] S. R. GANDHI. *A REAL TIME INDOOR NAVIGATION AND MONITORING SYSTEM FOR FIREFIGHTERS AND VISUALLY IMPAIRED* . Msc thesis, ETH Zurich, 2012.

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

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

[36] T. Gothivarekar, H. Ajay, C. Pathak, S. Yadav, A. Perupalli, and P. Gauda. Indoor Navigation System. 2015.

[37] 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.

[38] M. Hegarty. The Cognitive Science of Visual-Spatial Displays: Implications for Design. 2011.

[39] 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.

[40] M. Holcik. *Indoor Navigation for Android*. Msc thesis, MASARYK UNIVERSITY FACULTY OF INFORMATICS, 2012.

[41] 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.

[42] H. Huang and G. Gartner. A Survey of Mobile Indoor Navigation Systems. 2009.

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

[44] 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.

[45] R. Mautz. *Indoor Positioning Technologies* . Msc thesis, ETH Zurich, 2012.

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

[47] D. L. Moody. The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. 2009.

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

[49] 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.

[50] R. Nielsen, J; Mack. *Usability Inspection Methods, John Wiley and Sons Inc.* 1994.

[51] 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.

[52] 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.

[53] B. Peterson and R. Darken. Representation and recognition of the spatial organization of three-dimensional shapes. 1977.

[54] A. Petrenko. *GENERATION OF AN INDOOR NAVIGATION NETWORK FOR THE UNIVERSITY OF SASKATCHEWAN*. Msc thesis, University of Saskatchewan, 2013.

[55] Phonegap. `http://cordova.apache.org/docs/en/5.0.0/index.html`.

[56] 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.

[57] B. Pricope. *Positioning using terrestrial wireless systems*. Msc thesis, Jacobs University, 2013.

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

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

[60] J. Rehana. *Model Driven Development of Web Application with SPACE Method and Tool-suit*. Msc thesis, Norwegian University of Science and Technology, 2010.

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

[62] G. Skevakis, C. Tsinaraki, I. Trochatou, and S. Christodoulakis. A Crowdsourcing Framework for the Management of Mobile Multimedia Nature Observations. 2014.

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

[64] R. Snook. MSOA Design Patterns and Mobile. Technical report, IBM Software, Rational, 2013.

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

[66] Spring. `http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/overview.html`.

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

[68] 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.

[69] 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.

[70] L. P. WEN, C. W. NEE, K. M. CHUN, T. SHIANG-YEN, and R. IDRUS. Application of WiFi-based Indoor Positioning System in Handheld Directory System. 2010.

[71] J. Xiao and Y. Furukawa. Reconstructing the World's Museums. 2012.

[72] F. Y. Xiao J. Reconstructing the World's Museums, 2012.

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

[74] Y. Zhuang, H. Lan, Y. Li, and N. El-Sheimy. PDR/INS/WiFi Integration Based on Handheld Devices for Indoor Pedestrian Navigation. 2015.

Appendix A

Model Implementation Samples

This appendix provides samples of the implementation of the model, as persisted in the database, transferred through the services and used by the components. The database contains 8 main collections: (*a*) *"user"* collection holds the details for the users of the system who have an account, (*b*) *"building"* collection holds all buildings, (*c*) *"floor"* collection holds all floors of buildings, (*d*) *"generalObject"* collection holds the containing objects of floors, (*e*) *"objectType"* collection holds the structures, (*f*) *"node"* collection holds the containing nodes of structures, and (*g*) *"multimedia"* collection holds the multimedia which are stored in our system.

Apart from these, there are some additional collections holding pieces of data, like the collections that contain the data for authentication of users. In the following sections we will present sample data for the most important collections.

Users

```
{
    "_id" : ObjectId("562cdcb531da12efc40cab0e"),
    "_class" : "gr.tuc.music.tucips.core.model.User",
    "firstName" : "Petros",
    "lastName" : "Kontogiannis",
    "email" : "pkontogiannis4@gmail.com",
    "password" : "*&%^%^JHDST$56*&54",
    "dateOfBirth" : ISODate("2015-10-25T13:44:21.267Z"),
    "portfolioId" : ObjectId("562cdcb531da12efc40cab0f"),
    "createdBuildingsIds" : [],
    "createdGeneralObjectsIds" : [],
    "favoriteBuildingsIds" : [],
    "favoriteGeneralObjectsIds" : [],
    "joinedOn" : ISODate("2015-10-25T13:44:21.267Z"),
    "roleId" : ObjectId("562cdb3a31dae773cb074920"),
    "phone" : "",
    "activated" : true
}
```

Buildings

```
{
    "_id" : ObjectId("56bf449ee74a985ef488c08c"),
    "_class" : "gr.tuc.music.tucips.core.model.stable.Building",
    "fullName" : "Electronic Department",
    "shortName" : "ECE",
    "address" : "Kounoupidiana , Chania",
    "url" : "www.ece.tuc.gr",
    "description" : "It is with great pleasure that we welcome you to
       the web pages of the School of Electronic and Computer
       Engineering (ECE) of the Technical University of Crete
       (TUC)...",
    "creatorId" : ObjectId("562cdcb531da12efc40cab0e"),
    "floorsIds" : [
        ObjectId("56bf7052e74a98494001d2d1"),
        ObjectId("56bf706fe74a98494001d2d2"),
        ObjectId("56bf708ae74a98494001d2d3")
    ],
    "grade" : 0,
    "ratingsIds" : [],
    "multimediaDefaultId" : ObjectId("56c88bd0e74a98138c5591d0"),
    "multimediaIdsList" : [
        ObjectId("56bf6f95e74a98494001d2cd"),
        ObjectId("56bf6f95e74a98494001d2ce")
    ],
    "createdDate" : ISODate("2016-02-13T14:58:38.994Z"),
    "lastChangeDate" : ISODate("2016-02-13T14:58:38.994Z"),
    "lastChangedByUserId" : ObjectId("562cdcb531da12efc40cab0e"),
    "keywords" : [
        "ece",
        "tuc"
    ],
    "phones" : [
        "2821091239"
    ],
    "numberOfFloors" : 3
}
```

Floors

```
{
    "_id" : ObjectId("56c0c38ae74a9866bfef38bd"),
    "_class" : "gr.tuc.music.tucips.core.model.stable.Floor",
    "fullName" : "First Floor",
    "shortName" : "1st",
    "description" : "First Floor of ECE",
    "buildingId" : ObjectId("56bf98f2e74a9874499c5f23"),
    "creatorId" : ObjectId("562cdcb531da12efc40cab0e"),
    "level" : 3,
    "surface" : [],
    "scale" : 0,
    "multimediaIdsList" : [],
    "multimediaIdMap" : ObjectId("56c0c38ee74a9866bfef38be"),
    "temporaryObjects" : [],
    "containsObjectsIds" : [],
    "createdDate" : ISODate("2016-02-14T18:12:26.577Z"),
    "lastChangeDate" : ISODate("2016-02-15T17:18:36.839Z"),
    "lastChangedByUserId" : ObjectId("562cdcb531da12efc40cab0e")
}
```

General Object

```
{
    "_id" : ObjectId("56bf96c2e74a9874499c5f07"),
    "_class" :
        "gr.tuc.music.tucips.core.model.stable.IndoorLocationObject",
    "containsIndoorObjIds" : [],
    "containsMovableObjIds" : [],
    "containsMultifloorObjIds" : [],
    "type" : "indoor",
    "fullName" : "English Course Room",
    "shortName" : "145.P.42",
    "description" : "English Course Room Description",
    "creatorId" : ObjectId("562cdcb531da12efc40cab0e"),
    "floorId" : ObjectId("56bf708ae74a98494001d2d3"),
    "objectTypeId" : ObjectId("56bf413fe74a985ef488c07c"),
    "values" : [
        {
            "objectTypeAttributeId" :
                ObjectId("56bf413fe74a985ef488c07b"),
            "value" : 12
        }
    ],
    "multimediaIdsList" : [],
    "surface" : {
        "type" : "polygon",
        "color" : "#512864",
        "accessibility" : false,
        "accessPoints" : [
            {
                "x" : 81,
                "y" : 70,
                "len" : 5.300339611760741
            },
            {
                "x" : 84,
                "y" : 335,
                "len" : 8.420213774008353
            },
            {
                "x" : 505,
                "y" : 338,
                "len" : 5.38
            },
            {
                "x" : 505,
                "y" : 69,
                "len" : 8.480023584872862
            },
            {
                "x" : 81,
                "y" : 70,
                "len" : 8.480023584872862
            }
        ],
        "height" : 0,
        "opacity" : 0.2000000029802322
    },
    "scale" : 0,
    "createdDate" : ISODate("2016-02-13T20:49:06.963Z"),
    "lastChangeDate" : ISODate("2016-02-13T20:49:06.963Z"),
    "lastChangedByUserId" : ObjectId("562cdcb531da12efc40cab0e"),
    "publicAccessibility" : false
}
```

Structures

```
{
    "_id" : ObjectId("56bf4060e74a985ef488c072"),
    "_class" : "gr.tuc.music.tucips.core.model.extension.ObjectType",
    "type" : "indoor",
    "fullName" : "Office of the Secretary",
    "shortName" : "Sec Off",
    "description" : "Office of the Secretary Description",
    "creatorId" : ObjectId("562cdcb531da12efc40cab0e"),
    "objectAttributes" : [
        {
            "_id" : ObjectId("56bf4060e74a985ef488c06f"),
            "label" : "Phone",
            "description" : "Phone Description",
            "type" : "text",
            "multivalued" : false,
            "required" : true,
            "hasReference" : false,
            "order" : 0
        },
        {
            "_id" : ObjectId("56bf4060e74a985ef488c070"),
            "label" : "Fax",
            "description" : "Fax Description",
            "type" : "text",
            "multivalued" : false,
            "required" : false,
            "hasReference" : false,
            "order" : 1
        },
        {
            "_id" : ObjectId("56bf4060e74a985ef488c071"),
            "label" : "Office Hours",
            "description" : "Office Hours Description",
            "type" : "text",
            "multivalued" : false,
            "required" : false,
            "hasReference" : false,
            "order" : 2
        }
    ],
    "subTypesIds" : [],
    "levelType" : 0,
    "isUsed" : false,
    "usedByGeneralObjectsIds" : []
}
```

Nodes

```
{
    "_id" : ObjectId("56bfadb6e74a9874499c5f69"),
    "_class" : "gr.tuc.music.tucips.core.model.stable.Node",
    "nodeName" : "27744884-5a7f-4e43-8b72-19d24650e3c1",
    "point" : {
        "x" : 548,
        "y" : 352,
        "len" : 0
    },
    "creatorId" : ObjectId("562cdcb531da12efc40cab0e"),
    "connectedWithNodes" : [
        {
            "_id" : ObjectId("56bfadd0e74a9874499c5f70"),
            "distance" : 3.640054941177368,
            "disabled" : true
        },
        {
            "_id" : ObjectId("56bfadc0e74a9874499c5f6a"),
            "distance" : 0.854283332824707,
            "disabled" : true
        },
        {
            "_id" : ObjectId("56bfadc6e74a9874499c5f6d"),
            "distance" : 1.283900260925293,
            "disabled" : true
        },
        {
            "_id" : ObjectId("56bfad7ae74a9874499c5f67"),
            "distance" : 0,
            "disabled" : true
        }
    ],
    "connectedWithIds" : [],
    "floorId" : ObjectId("56bfabbae74a9874499c5f31")
}
```

Multimedia

```
{
    "_id" : ObjectId("56bf92fce74a98734b9a54bb"),
    "_class" : "gr.tuc.music.tucips.core.model.multimedia.Multimedia",
    "creatorId" : ObjectId("562cdcb531da12efc40cab0e"),
    "name" : "test.mp3",
    "contentType" : "audio/mpeg",
    "size" : "2905266",
    "path" : "1393b961-c974-45b4-bf59-41025bffa5c4",
    "type" : "audio",
    "createdDate" : ISODate("2016-02-13T20:33:00.636Z")
}
```

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) [29]. The data format supported is JSON, although the services can be easily extended to support other formats as well. The services have been classified in three categories:

- "User": describes services that are relevant to the user management.
- "Building": contains services that handle the buildings.
- "Structures": contains services that handle the structures of the system.

B.1 User

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

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

```
[
    {
  "data": [
    {
      "id": "562cdcb531da12efc40cab0e",
      "firstName": "Petros",
      "lastName": "Kontogiannis",
      "email": "pkontogiannis4@gmail.com",
      "userName": "",
      "dateOfBirth": 1445780661267,
      "portfolioId": "562cdcb531da12efc40cab0f",
      "createdBuildingsIds": [],
      "createdGeneralObjectsIds": [],
      "favoriteBuildingsIds": [],
      "favoriteGeneralObjectsIds": [],
      "joinedOn": 1445780661267,
      "roleId": "562cdb3a31dae773cb074920",
      "role": {
        "id": "562cdb3a31dae773cb074920",
        "roleName": "Student",
        "description": "Student",
        "buildingCategory": "University"
      },
      "phone": "",
      "activated": true
    },
    {
      "id": "56bf4566e74a985ef488c08d",
      "firstName": "Filio",
      "lastName": "Alvanou",
      "email": "filioalv@gmail.com",
      "userName": "",
      "dateOfBirth": 1455375718755,
      "portfolioId": "56bf4566e74a985ef488c08e",
      "createdBuildingsIds": [],
      "createdGeneralObjectsIds": [],
      "favoriteBuildingsIds": [],
      "favoriteGeneralObjectsIds": [],
      "joinedOn": 1455375718755,
      "roleId": "562cdb3a31dae773cb074920",
      "role": {
        "id": "562cdb3a31dae773cb074920",
        "roleName": "Student",
        "description": "Student",
        "buildingCategory": "University"
      },
      "phone": "",
      "activated": true
    }
  ],
  "code": "200"
}
]
```

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/554f1f7644ae93bee5fee7b5`

```
{
  "data": {
    "id": "562cdcb531da12efc40cab0e",
    "firstName": "Petros",
    "lastName": "Kontogiannis",
    "email": "pkontogiannis4@gmail.com",
    "userName": "",
    "dateOfBirth": 1445780661267,
    "portfolioId": "562cdcb531da12efc40cab0f",
    "createdBuildingsIds": [],
    "createdGeneralObjectsIds": [],
    "favoriteBuildingsIds": [],
    "favoriteGeneralObjectsIds": [],
    "joinedOn": 1445780661267,
    "roleId": "562cdb3a31dae773cb074920",
    "phone": "",
    "activated": true
  },
  "code": "200"
}
```

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

URL	/user/id
Method	PUT
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/562cdcb531da12efc40cab0e`

```
{
  "data": {
    "id": "562cdcb531da12efc40cab0e",
    "firstName": "Petros",
    "lastName": "Kontogiannis",
    "email": "pkontogiannis4@gmail.com",
    "userName": "",
    "dateOfBirth": 1445780661267,
    "portfolioId": "562cdcb531da12efc40cab0f",
    "createdBuildingsIds": [],
    "createdGeneralObjectsIds": [],
    "favoriteBuildingsIds": [],
    "favoriteGeneralObjectsIds": [],
    "joinedOn": 1445780661267,
    "roleId": "562cdb3a31dae773cb074920",
    "phone": "",
    "activated": true
  },
  "code": "200"
}
```

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/562cdcb531da12efc40cab0e`

B.2 Building

Table B.5: RESTful Service: Get all Buildings.

URL	/building
Method	GET
Description	Return the buildings by our system.
Example	`GET /building`

```
[{
  "data": [
    {
      "id": "56bf449ee74a985ef488c08c",
      "fullName": "Electronic Department",
      "shortName": "ECE",
      "address": "Kounoupidiana , Chania",
      "url": "www.ece.tuc.gr",
      "description": "It is with great pleasure that we welcome you to
          the web pages of the School of Electronic and Computer
          Engineering (ECE) of the Technical University of Crete (TUC),
          and we wish you a pleasant navigation through these pages.
          Our School is the second of its kind that was founded in
          Greece to cover the needs of the country's industry with
          engineers having studies and know-how in Computer
          Engineering.",
      "creatorId": "562cdcb531da12efc40cab0e",
      "floorsIds": [
        "56bf7052e74a98494001d2d1",
        "56bf706fe74a98494001d2d2",
        "56bf708ae74a98494001d2d3"
      ],
      "grade": 0,
      "ratingsIds": [],
      "multimediaDefaultId": "56c88bd0e74a98138c5591d0",
      "multimediaIdsList": [
        "56bf6f95e74a98494001d2cd",
        "56bf6f95e74a98494001d2ce"
      ],
      "createdDate": 1455375518994,
      "lastChangeDate": 1455375518994,
      "lastChangedByUserId": "562cdcb531da12efc40cab0e",
      "keywords": [
        "ece",
        "tuc"
      ],
      "phones": [
        "2821091239"
      ],
      "numberOfFloors": 3
    },
    {
       ...
    }
  ],
  "code": "200"
}
```

Table B.6: RESTful Service: Get Building's details.

URL	/building/id
Method	GET
Parameters	*id*: the building's id
Description	Return the details of the building.
Example	`GET /building/558beedcb7601b0fa8e03256`

```
{
  "data": {
    "id": "56c6cb7fe74a98102d6fe3e5",
    "fullName": "Nektarios House",
    "shortName": "Nektarios House",
    "address": "Nektarios House",
    "description": "Nektarios House",
    "creatorId": "562cdcb531da12efc40cab0e",
    "creator": {
      "id": "562cdcb531da12efc40cab0e",
      "firstName": "Petros",
      "lastName": "Kontogiannis",
      "email": "pkontogiannis4@gmail.com",
      "userName": "",
      "dateOfBirth": 1445780661267,
      "portfolioId": "562cdcb531da12efc40cab0f",
      "createdBuildingsIds": [],
      "createdGeneralObjectsIds": [],
      "favoriteBuildingsIds": [],
      "favoriteGeneralObjectsIds": [],
      "joinedOn": 1445780661267,
      "roleId": "562cdb3a31dae773cb074920",
      "phone": "",
      "activated": false
    },
    "floorsIds": [
      "56c6cc0fe74a98102d6fe3e6"
    ],
    "floors": [
      {
        "id": "56c6cc0fe74a98102d6fe3e6",
        "fullName": "Underground",
        "shortName": "Underground",
        "description": "Underground",
        "buildingId": "56c6cb7fe74a98102d6fe3e5",
        "creatorId": "562cdcb531da12efc40cab0e",
        "creator": {
          "id": "562cdcb531da12efc40cab0e",
          "firstName": "Petros",
          "lastName": "Kontogiannis",
          "email": "pkontogiannis4@gmail.com",
          "userName": "",
          "dateOfBirth": 1445780661267,
          "portfolioId": "562cdcb531da12efc40cab0f",
          "createdBuildingsIds": [],
          "createdGeneralObjectsIds": [],
          "favoriteBuildingsIds": [],
          "favoriteGeneralObjectsIds": [],
          "joinedOn": 1445780661267,
          "roleId": "562cdb3a31dae773cb074920",
          "phone": "",
          "activated": false
        },
        "level": 1,
        "surface": [],
        "scale": 0,
        "multimediaIdsList": [],
        "multimediaIdMap": "56c88b85e74a98138c5591cf",
        "temporaryObjects": [],
        "containsObjectsIds": [
          "56c6d280e74a98102d6fe3f6"
        ],
        "containsObjects": [
          {
            "type": "indoor",
            "id": "56c6d280e74a98102d6fe3f6",
            "fullName": "asdf",
            "shortName": "fasdf",
            "description": "asdfasdf",
            "creatorId": "562cdcb531da12efc40cab0e",
            "code": "asdfasdf",
            "floorId": "56c6cc0fe74a98102d6fe3e6",
            "url": "",
            "objectTypeId": "56bf4060e74a985ef488c072",
            "values": [
              {
                "objectTypeAttributeId": "56bf4060e74a985ef488c06f",
                "objectTypeAttributeName": "Phone",
                "value": "4352345"
              },
              {
                "objectTypeAttributeId": "56bf4060e74a985ef488c070",
                "objectTypeAttributeName": "Fax",
                "value": "234523452345"
              },
              {
                "objectTypeAttributeId": "56bf4060e74a985ef488c071",
                "objectTypeAttributeName": "Office Hours",
                "value": "sdfgsdfg"
              }
            ],
            "multimediaIdsList": [],
            "surface": {
              "type": "polygon",
              "color": "#512864",
              "accessibility": false,
              "accessPoints": [
                {
                  "x": 86,
                  "y": 76,
                  "len": 2.800017857085915
                },
                {
                  "x": 87,
                  "y": 356,
                  "len": 4.53004414989523
              },
              {
                "x": 540,
                "y": 358,
                "len": 2.8800694436072196
              },
              {
                "x": 542,
                "y": 70,
                "len": 2.55
              },
              {
                "x": 287,
                "y": 70,
                "len": 2.010895322984267
              },
              {
                "x": 86,
                "y": 76,
                "len": 2.010895322984267
              }
            ],
            "height": 0,
            "opacity": 0.2
          },
          "scale": 0,
          "createdDate": 1455870592156,
          "lastChangeDate": 1455870592156,
          "lastChangedByUserId": "562cdcb531da12efc40cab0e",
          "publicAccessibility": false,
          "containsIndoorObjIds": [],
          "containsMovableObjIds": [],
          "containsMultifloorObjIds": []
        }
      ],
      "createdDate": 1455868943534,
      "lastChangeDate": 1455870592200,
      "lastChangedByUserId": "562cdcb531da12efc40cab0e"
    }
  ],
  "grade": 0,
  "ratingsIds": [],
    "multimediaDefaultId": "56c6cb7fe74a98102d6fe3e4",
    "multimediaIdsList": [],
    "createdDate": 1455868799666,
    "lastChangeDate": 1455868799666,
    "lastChangedByUserId": "562cdcb531da12efc40cab0e",
    "phones": [
      "697585856"
    ],
    "numberOfFloors": 3
  },
  "code": "200"
}
```

B.3 Structure

Table B.7: RESTful Service: Get all structures.

URL	/structure
Method	GET
Description	Returns the structures created in our system.
Example	`GET /structure`

```
{
  "data": [
    {
      "id": "56bf4060e74a985ef488c072",
      "fullName": "Office of the Secretary",
      "shortName": "Sec Off",
      "type": "indoor",
      "description": "Office of the Secretary Description",
      "creatorId": "562cdcb531da12efc40cab0e",
      "objectAttributes": [
        {
          "id": "56bf4060e74a985ef488c06f",
          "label": "Phone",
          "description": "Phone Description",
          "type": "text",
          "multivalued": false,
          "required": true,
          "hasReference": false,
          "order": 0
        },
        {
          "id": "56bf4060e74a985ef488c070",
          "label": "Fax",
          "description": "Fax Description",
          "type": "text",
          "multivalued": false,
          "required": false,
          "hasReference": false,
          "order": 1
        },
        {
          "id": "56bf4060e74a985ef488c071",
          "label": "Office Hours",
          "description": "Office Hours Description",
          "type": "text",
          "multivalued": false,
          "required": false,
          "hasReference": false,
          "order": 2
        }
      ],
      "subTypesIds": [],
      "levelType": 0,
      "used": false
    },
    {
      ...
    }
  ],
  "code": "200"
}
```

Table B.8: RESTful Service: Get structure's details.

URL	/structure/id
Method	GET
Parameters	*id*: the structure's id
Description	Returns the structure's details.
Example	`GET /structure/558c11deb7601b0fa8e035b9`

```
{
  "data": {
    "id": "56bf4060e74a985ef488c072",
    "type": "indoor",
    "fullName": "Office of the Secretary",
    "shortName": "Sec Off",
    "description": "Office of the Secretary Description",
    "creatorId": "562cdcb531da12efc40cab0e",
    "objectAttributes": [
      {
        "id": "56bf4060e74a985ef488c06f",
        "label": "Phone",
        "description": "Phone Description",
        "type": "text",
        "multivalued": false,
        "required": true,
        "hasReference": false,
        "order": 0
      },
      {
        "id": "56bf4060e74a985ef488c070",
        "label": "Fax",
        "description": "Fax Description",
        "type": "text",
        "multivalued": false,
        "required": false,
        "hasReference": false,
        "order": 1
      },
      {
        "id": "56bf4060e74a985ef488c071",
        "label": "Office Hours",
        "description": "Office Hours Description",
        "type": "text",
        "multivalued": false,
        "required": false,
        "hasReference": false,
        "order": 2
      }
    ],
    "subTypesIds": [],
    "levelType": 0,
    "usedByGeneralObjectsIds": [],
    "used": false
  },
  "code": "200"
}
```