TÉCNICO LISBOA

DynamicPOI

Middleware for developing exhibition navigation applications

Diogo Miguel Neves Salgueiro

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. João Nuno de Oliveira e Silva

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. João Nuno de Oliveira e Silva
Member of the Committee: Prof. Miguel Filipe Leitão Pardal

September 2020

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

In the first place, I would like to thank my supervisor, Prof. João Silva, for his readiness and willingness to help during our first conversations about this work's topic. His availability, guidance, and support throughout the different stages of this work were fundamental to its completion.

To my parents, Elsa e Carlos, for all guidelines and values transmitted during my entire life and for allowing me to pursue my dreams and achieve my goals.

To my girlfriend, Rita, for being an essential part of my life and always being there for me. Your constant motivation, positivity, and support made it possible to overcome my most difficult challenges.

Finally, to all my friends and colleagues, for being a vital part of my academic and personal journeys.

Abstract

Museums and exhibition venues have been growing in the last decades, allowing a greater diversification both on their layouts and on the displayed artwork. On the other hand, the mobile application market growth allowed for these venues to provide better exhibition support and navigation applications to their visitors. However, each application is developed specifically for a single exhibition and the developer must implement a suitable exhibition structure and configure the necessary location systems.

One contribution of this work is the implementation of a generic schema that allows the definition of entire exhibition structures by establishing relationships between each of their points-of-interest, ranging from physical areas to displayed artwork items.

The other contribution is a middleware that improves the development process of exhibition navigation applications. This middleware aggregates several location systems transparently to the developer and automatically determines the most relevant POI of the exhibition, according to the visitor's location. Furthermore, an automatic services' management was also included to assure that only the necessary location services are active during an entire visit.

The middleware was implemented in the Android environment by creating a library package and separate location service classes. Its operation was evaluated by testing its features during demonstration visits and the energy savings of an application using this middleware were also determined.

Keywords

Points of interest, exhibitions, middleware, application, development, location services

Resumo

Os museus e locais de exposições têm crescido nas últimas décadas, permitindo uma maior diversidade tanto na sua organização como no tipo de peças exibidas. Por outro lado, o crescimento do mercado de aplicações móveis possibilitou a estes locais oferecerem melhores aplicações de suporte e navegação aos seus visitantes. No entanto, cada aplicação é desenvolvida especificamente para cada exposição e o programador tem de implementar a estrutura da exposição e configurar os sistemas de localização necessários.

Uma das contribuições deste trabalho é a definição de um esquema genérico que permite a definição de toda a estrutura de uma exposição, estabelecendo relações entre cada um dos seus pontos de interesse, desde áreas físicas às peças de arte exibidas.

A outra contribuição é um middleware que melhora o processo de desenvolvimento de aplicações para a navegação em exposições. Este middleware agrega vários sistemas de localização de forma transparente para o programador e determina automaticamente o ponto de interesse mais relevante da exposição, dada a localização do visitante. Além disso, o middleware efetua uma gestão automática dos serviços para garantir que apenas os serviços de localização necessários estão ativos durante toda a visita.

O middleware foi implementado no ambiente Android com a criação de uma biblioteca e classes de serviço de localização independentes. O seu funcionamento foi avaliado testando as suas funcionalidades durante visitas de demonstração e possíveis poupanças de energia para aplicações que utilizem este middleware.

Palavras Chave

Pontos de interesse, exposições, middleware, aplicação, desenvolvimento, serviços de localização

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AOA	Angle of Arrival
AP	Access Point
API	Application Programming Interface
AR	Augmented Reality
BDS	BeiDou Navigation Satellite System
BLE	Bluetooth Low Energy
BS	Base Station
GLONASS	Globalnaya navigatsionnaya sputnikovaya sistema
GML	Geometrical Markup Language
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HAL	Hardware abstraction layer
JSON	JavaScript Object Notation
MS	Mobile Station
NFC	Near Field Communication
POI	Point of Interest
QR	Quick Response
RSSI	Received Signal Strength Indicator
TDOA	Time Difference of Arrival
TOA	Time of Arrival
UI	User Interface
URL	Uniform Resource Locator
W3C	World Wide Web Consortium

1

Introduction

Contents

1.1 Motivation

Cultural related venues, *e.g.* museums and exhibitions, have been growing steadily in the last decades. This growth was accompanied by a greater diversification on the types of cultural heritage available to the general public. Furthermore, the technological evolution, particularly of mobile devices, allowed for better exhibition experience, *i.e.* how an exhibition's artwork is presented to its visitors [1].

Considering the growing concern with improving exhibition experience and visitor interaction, many museums have developed tour guides for their exhibitions [2]. These guides can have multiple technologies and purposes, ranging between simple audio guides to fully developed applications implementing several services (*e.g.* artwork information or augmented reality).

Furthermore, exhibition supporting applications can be very heterogeneous and support a wide variety of technologies. The developers must explicitly create each of these applications and their components, data structures and technologies.

1.2 Problem statement

Regarding the diversity of exhibition applications, two major problems can be outlined:

- Each exhibition must have suitable representation for the venues' artwork, Points of Interest (POIs) and areas. The nonexistence of a common solution that supports multiple types of exhibitions implies that each venue must answer this requirement independently. As a result, when a programmer creates a guide or support application to each venue, it must obey and adapt to the representation guidelines previously defined. In addition, these types of representations are not scalable or do not have any type of hierarchy between their elements.
- In order to determine the user's position relative to an exhibition, one or more location services must be configured and accessed. The existence of different types of venues and areas inside those venues implies that a given location service that suits some areas may not be used accurately on others. Again, when developing solutions that need to determine the visitor's position inside a heterogeneous exhibition, the programmer must configure each location service individually and combine the information of multiple services. After that, the programmer must take the visitor's location and find the nearest POI by accessing the POI structures described above.

1.3 Objectives

As a consequence to the problems outlined for an exhibition's helper application, identified on the previous section, there are two main objectives defined for this work.

The first objective is the definition of an generic representation for POI's structures. The main requirement for this representation is to be able to describe a large variety of POI structures, independently of their size, complexity and nature of the venues (indoor, outdoor or both). This implies that several location types must be considered as the most suitable one varies with each locale characteristics and location accuracy requirements. Other desired feature is not to be restricted to a single platform or data type or, in other words, to be able to easily be loaded into any relevant type of platform (*e.g.* web or mobile applications). This representation model must also be extensible with respected to the supported location types. At the same time, each POI structure that obeys to this representation's guidelines must be able to easily add, delete or modify its elements.

The second objective for this work is the development of a middleware that aims to simplify the development process of exhibition helper applications. The main purpose of this middleware is to determine the most relevant POIs identifier and make it accessible to any application that implements the middleware. This module must combine the information of an exhibition structure with the necessary location services' callbacks in order to determine the nearest POI at any given time. This modules' operations should be completely concealed not only from the application layer (that receives the nearest POI updates) but also from the layer that provides all the sensor information needed for the different location services. Considering the previous objective of having a extensible representation for POI structures, the library module should be also allow the future integration of new services and the support of new location types.

1.4 Accomplishments

The first major contribution of this work is the definition of schema that sets guidelines for the definition of POI' structures. These structures can be used in several kinds of exhibitions or venues, allowing to easily catalogue POIs or artwork in every area, regardless of scale.

The other major contribution is a middleware that transparently combines several location systems and determines the most relevant POI for an given location. This simplifies the development process of any programmer that wishes to develop applications that assist visitors, regardless of exhibition or location typing. This middleware abstracts the programmer from configuring each necessary location system and mapping each of their responses into a relevant POI on the exhibition's structure.

In addition to the abstraction provided for every application that uses the developed middleware, an efficient management of the location services was integrated into it. By evaluating the user's position related to a given exhibition location, *e.g.* inside a specific building or room, only the necessary location services will be accessed.

As a consequence to the library module's ability to easily add new types of location services, a new

service based on QR codes was developed and integrated. This service allowed for a more accurate definition of the nearest POI, specially on venues with several artworks in a small area (where the conventional location services don't have the necessary accuracy).

1.5 Thesis Outline

This thesis has seven main chapters and their content is summarized as follows:

The Introduction (Chapter 1) gives a general overview and motivation for this work and briefly describes its objectives, contributions and how it is structured.

The Related Work (Chapter 2) analyzes the state of the art of the relevant topics for this work and assess some related solutions developed to tackle similar problems.

In the DynamicPOI Design (Chapter 3) the general requirements for the solution are described as well as the required modules for the solution's architecture.

The DynamicPOI Implementation (Chapter 4) explains how the proposed library module was developed to respond to the proposed objectives. The module's constituent classes are characterized as well as which location services were integrated. In order to demonstrate the operation of the developed module, Demonstration (Chapter 5) describes the development process of a demonstration application and the integration of a new location service.

In Evaluation (Chapter 6), the proposed solution is evaluated against the initial goals. In addition, some demonstration visits were designed using the proposed proposed solution and the results obtained after those visits are presented.

Finally, in Conclusion (Chapter 7), the key outcomes of this work are presented and further improvements and future work recommendations are suggested.

2

Related Work

Contents

This chapter presents an overview of the most relevant topics for this work. Section 2.1 describes several types of exhibitions and new technologies used to support them. Section 2.2 describes the most commonly used location identifiers. The most relevant location systems for mobile devices are described on section 2.3. Lastly, sections 2.4 and 2.5 presents POI data models and the necessary programming models when developing exhibition navigation applications, respectively.

2.1 Exhibitions

The ability to pass on knowledge and cultural information between generations was one of the most crucial processes for the human evolution. This continuous process can be observed on the vast and diverse cultural heritage that is available nowadays. By definition, cultural heritage comprises not only all tangible cultural expressions such as artwork, buildings and landscapes but also intangible culture like knowledge and traditions [3].

Considering the extensiveness and diversity of the physical component of the cultural heritage, exhibitions play an important role by easily group, display and preserve of its constituents. By definition, an exhibition is an organized group of items to be presented to an audience. The scope of an exhibition may vary tremendously as it can span between the artwork of a single person to artifacts of an entire civilization [4].

2.1.1 Types of exhibition venues

Given the extent of possible exhibition scopes, one consequence is the large variety and dimensions of venues that can be designed to accommodate them. The most elementary exhibition could be located inside a single room, with varying dimensions suitable for the displayed items. On the other hand, art galleries and museums can have several spaces dedicated to hosting single or a set of exhibitions. Some of these venues are part of art foundations that usually have large gardens and other notable areas for displaying exhibition items.

The Gulbenkian [5] and Serralves [6] foundations are two examples of multi layered exhibition sites. Their gardens display several art sculptures but they also have noteworthy museums, each containing several themed exhibitions comprised of multiple rooms.

Widening the scope of exhibition venues, Parques de Sintra groups 10 of the most relevant venues of Sintra's municipality, including palaces, castles and gardens. Each on this venue can be considered as an element of a complex exhibition (instead of isolated ones) as they are related to each other, despite being scattered across the Sintra-Cascais natural park [7].

Lastly, if a more extensive notion for culturally relevant heritage is considered, one can include very

large areas such as geological parks as complex and widespread exhibitions. UNESCO Global Geoparks currently lists 161 geological parks, 5 of them in Portugal [8].

This variations of complexity and size for these types of exhibitions pose several challenges for cataloging its constituent parts (both venues and artwork items) and providing proper navigation solutions to their visitors.

2.1.2 New technologies used in exhibitions

The smartphone market has grown tremendously in the last decade and became one of the most important type of devices worldwide. This was motivated, not exclusively, by the ability to run specific applications for each of the users' needs. Museums and other exhibition venues have directly benefited from this technological breakthrough, as it allowed them to personalize and diversify the visitors' experience inside their venues [9] [10].

Prior to the use of mobile phones inside exhibitions, visitors only had access to predefined routes, in form of paper guides or when accompanied by the exhibition's audio guides. Nowadays, several museums have their own applications, capable of displaying different types of tour information directly on the visitors' phone. Some of these solutions can even create personalized tours accordingly to visitor preferences. Real time positioning and step-by-step navigation can also be provided, by using indoor location techniques enabled by the smartphones' sensors, with or without extra equipment installed on the exhibition itself (*e.g.* item or room identifiers).

By utilising the smartphones' multimedia capabilities, exhibitions could now provide a better visitor experience. One key aspect when visiting an exhibition is how the item's related information is presented to the visitor. Instead of the classical audio guides (via small devices given at the beginning of the visit), visitors can now have on-demand access to videos and audio descriptions, directly from their smartphones. Likewise, textual information can also be give in a similar fashion, providing a faster and more personal experience.

One of the most recent technologies to be used to enhance the visitor's experience is Augmented Reality (AR) [11]. This technology allows live rendering of 3D objects in the smartphone's viewfinder while the user pans around an area. Nassar *et al.* [12] approaches several challenges in adopting this technology in mobile devices and proposes an AR system to be used in iOS devices. Desai [13] further examines AR in museums, particularly as a tool for recreating historical artifacts. Bone Hall [14], one of the permanent exhibits at the Smithsonian Museum, has a dedicated AR application that displays 3D models of corresponding animals when users point their devices towards the exhibit's skeletons.

Lastly, gamification and storytelling frameworks can be used to further improve the experience and make the user feel like he is an active part of the exhibition and not just a spectator. The usage of these frameworks are specially relevant for younger audiences as classic types of exhibitions fail to engage

them [15]. Again, given their application versatility, smartphones can be crucial in materializing these mechanisms as some solutions have been proposed [16] and several museums have started to incorporate these techniques in their applications. However, these applications are individually developed for each museum or exhibition and their existing technologies.

2.2 Location identifiers

The basis for any location system is the type of location identifier it uses to describe its determined locations. Three categories of location identifiers are described below:

- **Geographical or absolute coordinates** - the location is described by a set of coordinate system whose origin point is the Earth's center. The most commonly used coordinate systems are ellipsoidal coordinate systems, which describe locations using two angles relative to two orthogonal reference planes: latitude, the angle relative to the Equator plane and longitude, the angle relative to the Greenwich meridian plane [17]. If necessary, an altitude value can also be provided along with these two coordinates;

- **Relative or local coordinates** - This type of coordinates differ from the geographical ones by recurring to an referential other than the Earth's center. Therefore, these coordinates are accompanied by which referential they are relative to, *e.g.* a map or a predetermined physical point;

- **Symbolic identifiers** - a location is described by an identifier or human understandable name, *e.g.* locating an object simply as "inside a room", not specifying its position with discrete coordinates.

2.3 Location systems

Nowadays, one of the demands of mobile phone's users is to easily know their location. In order to have this functionality, mobile phones can use several location systems depending on the user's surroundings. In this subsection, an list of location systems (available on mobile phones) is presented.

2.3.1 GNSS

Global Navigation Satellite Systems (GNSSs) are location systems that are comprised of a group (constellation) of satellites and ground control stations. The currently operating GNSSs are briefly described below:

- GPS (Global Positioning System) - The first GNSS to be operational and globally accessible (1994). Owned and operated by the United States government, it currently has 31 operational satellites on its space segment [18].
- GLONASS (Globalnaya navigatsionnaya sputnikovaya sistema) - Russian operated GNSS, is composed by 24 orbiting satellites distributed along 3 orbital planes and achieved global coverage on 1995 [19]
- Galileo - Created by the European Union, aimed in providing a GNSS independent from GPS and GLONASS, has a planned constellation of 30 satellites (24 operational and 6 active spares). This system is operational since 2016, with 26 satellites deployed at the moment [20] [21].
- BeiDou Navigation Satellite System (BDS) - Chinese three-phase satellite system, with the latter phase launched in 2015 and aimed at global availability. The last of the 35 currently orbiting satellites was successfully launched in June 2020 [22].

GNSSs have a common operating principle for the positioning of client devices. By design, all of their constituent satellites are synchronised and their orbital distribution ensure that, at any location on Earth, several satellites visible for client devices. By having several satellite signals available, the client device determines the signal travel times and then the distance between the device and each satellite (as the signal velocity is constant). The final position is calculated as the intersection of the ”imaginary” spheres based on the calculated distances.

This location technique requires line-of-sight between a group of satellites and the receivers. When this condition is met (for instance, on open areas) the GNSSs can produce very accurate positioning, with accuracy of a few meters. Several GNSSs can be used simultaneously to further increase the accuracy [23].

On the other hand, on places with a large amount of tall buildings and indoors, the accuracy of these systems plummets and can even stop operating under these conditions. In these environments, the GNSSs are generally used in conjunction with other location technologies.

2.3.2 Cellular networks

Cellular networks are complex mobile networks that provide wireless communication to devices inside its constituent cells. Although these types of network were not originally designed for positioning, their Base Stations (BSs) allow for several positioning techniques of the connected Mobile Stations (MSs) (*e.g.* mobile phones) [24]:

- Cell identification - Knowing a BS’s location, the position for connected MSs can be assessed to be inside its radius. Can be suitable for very small cells (*e.g.* urban areas), in larger cells positioning

via this method is mediocre.

- Signal Strength - Measures of several BS signal strengths may be used to triangulate a MS position, *i.e.* as the intersection of three or more BS's computed radius. In areas with buildings or relevant landmarks, complex signal models might be needed to have satisfactory results.

- Angle of Arrival (AOA) - By evaluating the angle of arriving signals for two or more BSs, locations can be determined for connected MSs. This technique requires antenna arrays on the BS and clear line-of-sight between the BSs and MS is very desirable for accurate positioning.

- Time of Arrival (TOA)/Time Difference of Arrival (TDOA) - Based on time measurements of exchanged signals between BSs and MSs. In TOA, the time measurements are absolute relatively to a single station as inTDOA time difference between two ore more stations are considered. There are several TOA and TDOA algorithms, some specific to the cell network protocol.

A brief performance comparison between some of the described techniques was presented by [25]. Campos [26] also presents the positioning techniques in cellular networks and, most importantly, describes their evolution from 2G through 4G cell networks.

These networks' positioning capabilities are almost exclusively used on outdoor environments and can only determine locations with a maximum accuracy of several meters. Thus, cellular networks based location systems are usually paired with GNSSs to obtain more accurate locations.

2.3.3 WiFi

WiFi is one of the most relevant technologies not only for wireless communication but also for mobile devices location. This technology encompasses several protocols and is present in almost all modern mobile devices.

As one of the most common uses on this technology is the ability to have wireless networks centered on WiFi Access Points (APs). As almost all of these APs are fixed, by having a database correlating them with their physical location, one can infer one approximate location of mobile devices connected to them. Some databases are freely accessible like the Wireless Geographic Logging Engine [27] which database accepts user contribution and has over 600 million WiFi entries. Others, like Google's AP database, are not publicly accessible but it is used by its location services to improve or validate user locations.

On the other hand, as a radio signal technology, WiFi capable devices can be located using similar methods as the described in section 2.3.2 such as AOA, TOA/TDOA and Fingerprinting. Fingerprinting techniques present the advantage of not requiring a clear line-of-sight between WiFi devices in order to compute physical locations. This is significant as WiFi networks are primarily located indoors, where

guaranteeing line-of-sight between all devices might not be possible. Instead, these techniques rely on Received Signal Strength Indicator (RSSI) readings to operate. However, signal refraction and multi-pathing must be taken into account in order to obtain consistent positioning. Therefore, fingerprinting methods are heavily studied and improved in order to have good positioning outcomes [28] [29].

2.3.4 BLE

Bluetooth Low Energy (BLE) is a protocol built from the original Bluetooth specifications, aimed at reducing energy consumption while maintaining a similar range of communication [30]. This reduced energy consumption allowed the creation of a set of devices called beacons, that could advertise different types of information for long periods of time.

BLE devices can be detected by mobile devices and, by analyzing the RSSI for each one, distances can be estimated. However, the accuracy of this metric might not present satisfactory results if used directly to determine one's location: first, the nature of radio signals result in very difficult signal intensity models when used indoors and secondly BLE devices use the same frequency range as WiFi and other Bluetooth devices, which can lead to signal interference in crowded environments.

In order to achieve better positioning using RSSI measurements, the fingerprint method is often used. This method essentially consists of two phases: an offline phase - several reference points are defined in the installation area and a radio map is built with the signal strengths measured for each device at each reference point; an online phase - the location of a user is then determined by using algorithms that try to find the most suitable fingerprints of the radio map from its live signal strength measurements. Faragher *et al.* [31] evaluated this location technique when applied to BLE devices. Several parameters, both on the fingerprinting method as on the beacons' configurations, were evaluated on how they can affect the accuracy of the positioning results. Pu *et al.* [32] focused on evaluating different matching algorithms for the latter phase of the fingerprinting technique, *i.e.* mapping the user position from a previously defined radio map.

Despite the efforts described above at determining at maximizing the location accuracy of BLE devices, at some venues a mere symbolic approach might be sufficient *i.e.* knowing which beacon is the nearest to the user.

2.3.5 NFC

Near Field Communication (NFC) is a bidirectional wireless communication technology, that enables information exchange between several types of devices when placed a few centimeters from each other. The most common applications of NFC are (i) reading/writing NFC tags, (ii) exchange data between two devices (*e.g.* smartphones), (iii) card emulation [33].

NFC tags are small chips that can store data that can be accessible when a compatible device is close to it. These chips are passive *i.e.* their antennas are not actively powered. Instead, they use magnetic induction from the device attempting to access them in order to power the chip. This NFC tags' property allows virtually no operational costs after setting up the information on them.

Regarding this technology as a component for location systems, the immediate approach is to use NFC tags as identifiers for a given positioning system. As the reading process implies that the user is very close to the NFC tags, if their location is known *a priori*, when a user reads a tag, it's position is determined very accurately. Ozdenizci *et al.* [34] proposed an indoor navigation system using NFC tags, encoding both an indoor map and location information on them. These tags were then read by a mobile phone running an application that provided directions to the user.

2.3.6 QR codes

Quick Response (QR) codes are bi-dimensional bar codes that can encode information in a machine-readable format. Originally designed for tracking automotive parts, QR codes nowadays are used widely across several industries and with different purposes [35]. The growth of this type of identifiers followed the smartphone market boom as these devices have cameras and software to decode the information.

As a mean for encoding information, QR codes can be used to locate a user by two approaches:

1. A QR can directly encode its position relative to a known referential *e.g.* geographical coordinates;

2. A QR code can be an unique identifier of an item inside a venue. If the position of the item is known, when scanning its identifier, one can assess that the user is close to the item's position. Figure 2.1 displays a user scanning a QR of an exhibition artwork.

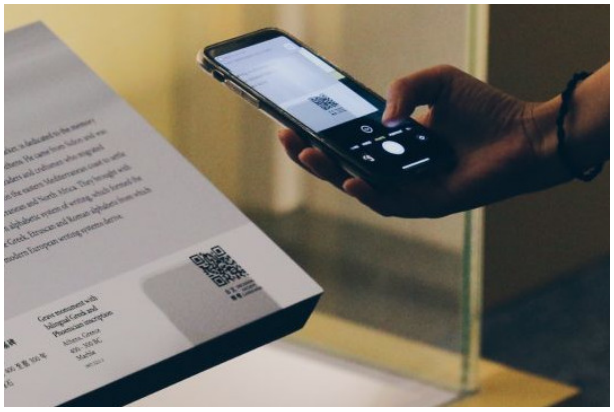


Figure 2.1: A visitor scanning an item's QR code using a smartphone. [1]

[1] ware.hk/project/aol-british-museum-exhibition/

2.4 Points of Interest

Informally, a Point of Interest is any place that is relevant and/or useful to someone. This definition generally encompasses different types of venues and facilities but also natural sites. However, in this work, a wider scope for the POI definition will be considered, ranging from entire areas to single exhibition items, as these latter can also be considered relevant points (but on a smaller scale). Therefore, this approach accounts for the possibility of POIs to exist inside other POIs, which is rarely considered when developing existing POI based applications (as described on the following subsections).

2.4.1 POI data models

When developing an application, one of the most fundamental steps is to define a data model that backs all the information that is processed by it. Considering applications that make use of POIs, a convenient data model to represent them must be defined. The following paragraphs present two approaches used in designing a suitable data model for POIs.

W3C POI

The World Wide Web Consortium (W3C) is a organization that aims to develop several standards to be used in the World Wide Web. The ”Point of Interest Working Group” was created by the W3C in September 2010 to deliver guidelines for the representation of POIs [36]. The last draft of this group was published in March 16th 2012 [37] and the UML diagram of this proposed data model is presented in figure 2.2.

The ”POIBaseType” acts, as its name suggests, as base type for almost every other entity in the data model and stores all of their internal information. ”POIType” entity stores the ”real world” information for a given POI such as its name (”label”), description, category and a time reference. The ”POI” entity then extends from the ”POIType” and adds a reference for a ”Location” entity that adds spacial information to it. ”Location” supports the representations of the Geometrical Markup Language (GML) (point, line or polygon) and allows a relation to another ”POI” via the ”Relationship” entity.

Regarding the possible connections between several POIs, two aspects must be pointed out:

1. The ”Relationship” element only concerns to the spacial relation between two POIs. For instance, this connection allows for a POI to be contained inside another but it does not allow any further relations between other POIs.

2. The ”POIS” element only allows for a list of several ”POI” elements, *i.e.* no nested lists can be constructed using this model.

Considering the previously described aspects of this data model definition, one can conclude that

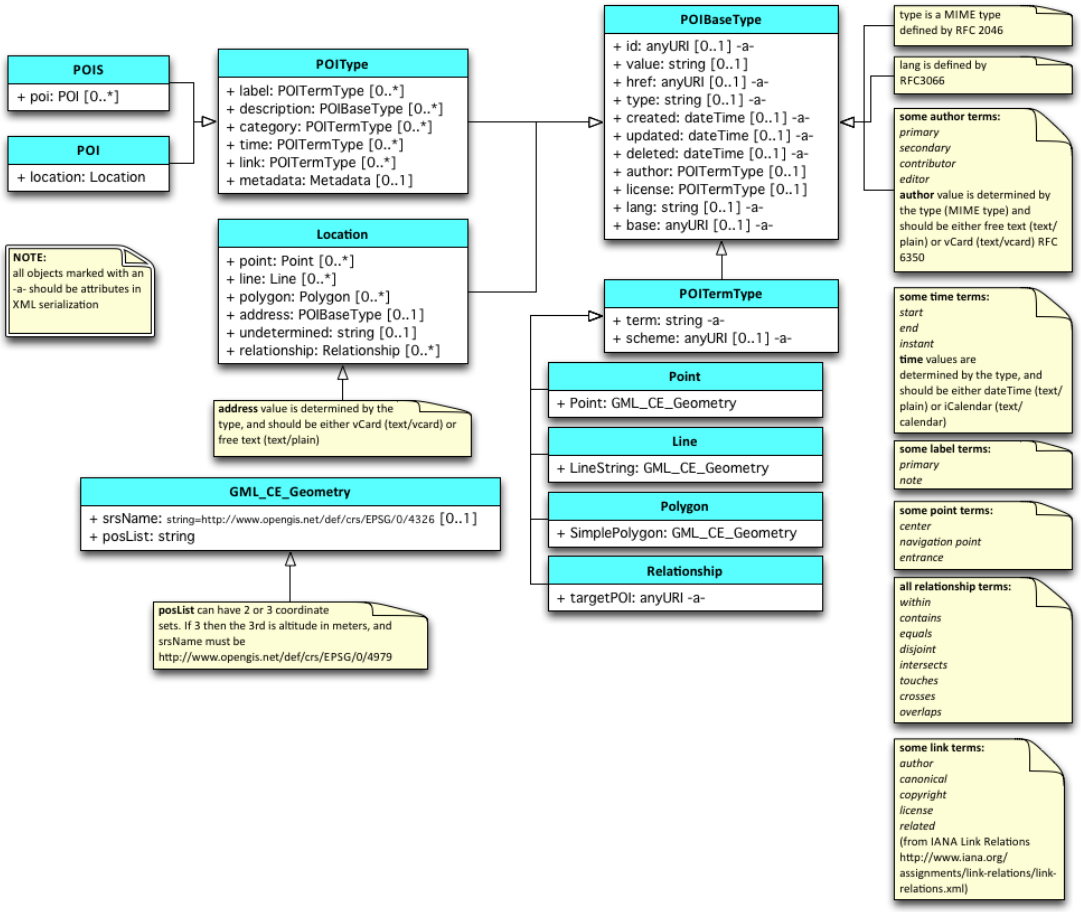


Figure 2.2: W3C's Point of Interest UML diagram.

this data model is a good starting point when modelling POIs and is suitable for building categorized lists. However, it doesn't favor the creation of more complex POI structures, *e.g.* a hierarchical network, that could be used as base for multi-venue exhibitions or large areas.

FIWARE POI

FIWARE is a open source initiative endorsed by the FIWARE Foundation that aims to provide a set of standards and components for the development of smart solutions. Its catalogue contains a large number of data models, including a proposed model for representing POIs [38].

Their POI definition is based on the JavaScript Object Notation (JSON) and consists on a single entity that has a set required fields, described below. In order to validate JSON data sets that follow this specification, a JSON schema was also defined.

- Required
  - id: a unique identifier for the POI;

  - type: has the predefined value of ”PointOfInterest”;
  - name: self explanatory, designation of the POI;
  - category: numeric value of a predefined set of values, each corresponding to a POI category, with some added values with this specification.

- At least one is required
  - location: can have one of the possible ”Geometry” structures defined in the GeoJson format [39]
  - address: can have a textual value or a set of properties that constitute a physical address, *e.g.* country, locality, street and postal code.

The rule set for the ”address” and ”location” fields favors a more flexible approach in terms of positioning the POIs as some can use only an address, others a geographical reference or use both.

There are also a set of optional fields that allow for further detailing of the POI entry, such as a description, its source and timestamps for its creation and/or modification date.

Evaluating the available fields of this data model, an immediate observation is the lack of any kind of connection between any other POIs. Although this model provides the possibility of creating lists of detailed POIs, when considering a single element, there is no way to directly assess if it is related to another one. Furthermore, this lack of relationship between different POIs impedes the creation of more complex structures and the usage of a hierarchical notion.

2.4.2 POI applications

After describing above some approaches for POI data modelling and how fundamental a suitable data model can be when developing an application, some applications that use POIs are presented in the following paragraphs.

The CitySDK project was created to ease the development of Smart City solutions, by having unified and open data interfaces for several cities in Europe. One of its three Application Programming Interfaces (APIs), the TourismAPI [40] was targeted at the different intervening parts of tourism solutions: the tourists themselves, the solutions developers and the information providers of touristic venues. This API provides standardized data models for POIs, Events and Itineraries as well for their categories. These four data models were based on the W3C’s POI data models: the POI was mapped directly from the specification and the remaining ones were adaptations of some of its base entities.

As stated in 2.1.2, tour guide applications have become common inside modern museums. One extension of this is the ability to recommend POIs inside an exhibition while a user visits it. Hashemi *et al.* [41] explored this thematic and how it can relate to IoT sensors for better user tracking. Regardless

of the main focus of this study is the recommendation system, a good structural representation on how the POIs are organized inside a museum is a key aspect for this type of solutions.

Nitti *et al.* [42] proposed a IoT based architecture for tourism applications in Smart Cities. This architecture takes advantage of IoT devices placed at POIs in order to easily have relevant information for a touring application, *e.g.* queue times. The use case presented had the goal to minimize the wasted time when visiting a set of POIs. Despite not explicitly refer which data models were used, one can assume some kind of association between a POI representation and the possible IoT devices had to be made.

Taking into consideration the three POI-based applications presented above, one of the outcomes is the ability to have both indoor and outdoor POIs but rarely this two types are considered at the same time inside an application. Furthermore, these applications didn't present complex relationships between their base POIs, as simple listings were considered.

2.5 Programming models

As stated on section 2.1.2, the growing capabilities of mobile devices coupled with the need for better and diversified exhibitions ensued the development of mobile applications targeting exhibition visitors. One fundamental aspect of these applications, is the ability to correctly locate the user within an exhibition. As exhibitions can have a multitude of configuration and environments, the application must combine several location technologies. In addition, the application needs be aware of the exhibition's layout and its items.

The most relevant functionalities for exhibition applications and how their developers can incorporate them are described in the following subsections. This section will only consider the Android mobile development, as it is currently the most used mobile operating system [43]. Nonetheless, these procedures are very similar when developing Apple's iOS applications, with very few exceptions.

2.5.1 Location libraries

Nowadays, mobile phones have access to several location systems in order to be able to locate the users in different environments. As previously described in 2.3, each of this location systems can have several protocols and means of operation. In addition, several manufacturers might have their own implementations of these protocols which results in being impractical to obtain location data without a set of libraries available to the developer.

On the Android environment, locations are be obtained using a "Fused Location Provider", which combines the mobile devices' sensors on a unified location callback. The location provider can be configured to only use GNSSs data to determine the user's location (which can be limited in indoor

environments) or combine this data with the WiFi and Bluetooth systems. These latter systems are then used to scan nearby APs and use the Google's AP database to better locate the user (specially indoors).

Regarding location libraries available to the developers, the Android native "location" library [44] allows the developers to obtain meaningful location data, *e.g.* a set of geographical coordinates, without having to directly target the phone's sensors. This library was introduced on the first Android version and besides providing location data it also allowed to configure and retrieve data of the GNSSs supported on the device.

However, this API was deprecated in favor of the Google Location Services API [45]. This newer API provided further abstracted the positioning algorithms from the developer and improved the location accuracy. The location data can now be obtained from a new "Fused Location Provider", which seamlessly aggregates the devices' location systems.

2.5.2 Beacon programming

In order to add BLE beacon scanning capabilities to an application, the simplest step is to use the native Bluetooth API of Android [46]. This library implements a large set of functionalities, both of the "classic" Bluetooth and BLE, being one of them the ability to scan BLE devices and obtain their broadcast information and RSSI. The two main advantages when using the native library is the fact that is already included in the core Android libraries and it is well supported and documented.

One alternative to the native library approach is the AltBeacon's library and its APIs [47]. Aiming at providing an open specification for beacon advertising messages, Radius Networks created the AltBeacon standard. In addition, an Android library was developed to support not only devices compliant with the AltBeacon standard but also with other popular beacon standards, *e.g.* iBeacon and Eddystone.

Lastly, several companies had developed their own beacons along with proprietary protocols for communication and configuration. External libraries or SDKs provided by these companies must be included in the application code in order to support their devices. Aruba [48] and Kontak.io [49] are two companies that supply public accessible SDKs to help developers integrate their products. Sousa [50] developed a mobile middleware solution which integrated the Aruba Beacons SDK on a positioning component.

2.5.3 QR code and NFC programming

As stated in section 2.3, both QR codes and NFC tags can be used as item identifiers when scanned at short distances by a compatible device. The inclusion process for these two feature inside an application is similar as several native libraries exist to support them.

The QR codes scanning process in Android phones requires access to two different set of libraries:

one for accessing the device's camera and other to decode the QR code information from the its viewfinder live feed. In this particular functionality, third-party libraries for decoding these type of codes were the first to appear. The Zxing ("Zebra Crossing") project developed a Java based library for decoding one-dimensional and two-dimensional barcodes [51] and has being compiled for several programming languages and platforms. Later, Google included libraries for reading barcodes [52], in which QR Codes are included, as part of the Mobile Vision API.

With the inclusion of the NFC technology, mobile phones gain the ability to read and write NFC tags, peer-to-peer communication and emulate NFC cards. Developers can add these functionalities to their applications via a set of native libraries. Considering the NFC tag reading process, a developer can program an application to execute different actions when the user reads a tag with a know structure, *i.e.* a museum can codify a set of tags' identifiers to trigger specific actions on its own application.

One final aspect for these two technologies is the fact they both need some kind of user action in order to operate correctly. Contrary to the GNSS, Wifi and BLE systems (which automatically provide a position), a user must actively point a camera to QR code or approach the device to a NFC tag to trigger an action. This inherent aspect must be taken into account by the developer when programming the interactions with these technologies.

2.5.4 POI data sources

Another relevant topic to be considered when developing exhibition applications is how these applications can obtain, manage and store the POI information. Two main approaches can be taken on the development process to obtain the POI's structure:

1. The application could have the entire exhibition structure compiled along with the application code. Despite being pragmatically simpler, this approach implies that, in the event of an exhibition structure modification, a new version of application must be created.

2. The application could interact with an external source, generally a web service, to obtain the exhibition structure. By following this approach, the exhibition layout will always be loaded at runtime, effectively decoupling possible exhibition structure updates from updating the application itself.

More complex approach can utilize persistent local database solutions in conjunction with web services when updates on the exhibition structure are available. On the Android environment, two of the most popular local database libraries available to developers are SQLite [53] and Room [54]. Besides providing simple mechanisms to persistently store data, they can also share the database format of remote databases (of web services), simplifying their integration process.

In addition, the POI information can be subdivided into POI identifiers and extra details for those POIs, allowing the application developer to chose the most suitable mechanism for obtaining each one. The

Second Canvas project [55] develops media enhanced applications for European museums. including. For instance, the application for the Museu Nacional de Arte Antiga [56] has a precompiled exhibition structure but the extra information, such as the image resources, is loaded from a web service on the first application execution.

2.5.5 Programming challenges

Despite having several libraries that help developers accessing the location systems of the target devices and providing data storage capabilities, there are still several challenges when developing an exhibition navigation application.

In the first place, when adding a companion application to an exhibition site, some sort of mapping must be made, *i.e.* the physical properties of the exhibition must be represented by a data structure, readable by the application. These physical properties can refer to the exhibition's areas, buildings and rooms but also where the different items are placed within the exhibition and how they can be located. As there is no generic structure to describe exhibitions, the developers must find the most suitable way to represent each exhibition. This task becomes more tiresome if the developer needs to outline different kinds of exhibitions as a suitable format for a venue might not be appropriate for another, *e.g.* a museum description contrasts with the description of a large outdoor park. Furthermore, the chosen data structure must be able to be easily adapted to changes in the physical properties of the exhibition.

Considering the different means to locate both areas and their items of a venue, the developer must guarantee that all the necessary location systems are integrated. However, if a new location system is added to the exhibition, *e.g.* BLE beacons identifying rooms, the developer has to openly integrate this new system on the application. This might imply changes on the previously developed application structure and location algorithms as they now have to consider this new location system.

Lastly, the developer is also responsible for managing all the location systems in these applications, *i.e.* determine when they should be active and with which parameters should they operate. This process is crucial in terms of power consumption as some of these systems (GNSS and cellular networks) can noticeably drain the device's battery. Therefore, an efficient management of the location systems is an imperative feature for these applications.

3

DynamicPOI Design

Contents

This chapter presents the design of a middleware that improves the development of exhibition navigation applications. The main features for these middleware are:

- Combine multiple location systems;
- Determine the most suitable POI for the visitor;
- Automatically activate/deactivate location services;
- Allow for exhibition layout modifications or addition of new feature with minimal changes on the applications.

Starting by identifying its requirements on section 3.1, the solution's generic architecture is outlined on section 3.2 as well as the function of each of its main components. The section 3.3 presents guidelines for a suitable data model and the main methods and callbacks of the solution's API are summarized on section 3.4.

3.1 Requirements

3.1.1 Functional requirements

When considering an application for exhibition navigation, one can consider three different concerned user classes: the exhibition curator - who defines the exhibition layout; the developer - who programs the application; and the visitor - the final user of the application and visitor of the exhibition. This categorization is important as each of these user classes has a different role and, therefore, can have different requirements.

Exhibition curator

This user class comprises not only exhibition and museum curators but all the entities responsible for an exhibition venue, *e.g.* municipalities. The functional requirements for this user class when considering exhibition navigation applications are:

FR1 - Define relationships between the constituent areas of an exhibition - The curator must be able to describe all of the areas of an individual exhibition and how they are related to each other. Both indoor and outdoor spaces must be supported;

FR2 - Attribute item positions - The curator should be able to place each item inside its respective area;

FR3 - Assign location identifiers to rooms and their items - The curator must have the ability to provide an identifier for each of the exhibition's elements (both areas and items);

FR4 - Support a variety of location systems - The curator must have access to a large variety of location systems to describe where items and areas are located, as a suitable system for a given area might not appropriate for another;

FR5 - Modify a previously defined exhibition structure - As exhibitions are physically mutable, *i.e.* areas or items could be added, removed or modified, the curator should be able to update their corresponding data structures.

Developer

The developer user class considers, straightforwardly, all the developers of these type of applications. A set of expected available functionalities to this user class, during the development process, is described below:

FR6 - Develop applications that allow information access about items and areas - The developer must have the necessary tools to develop this type of applications;

FR7 - Modify the areas and items structure without recompiling the application - The developer must be able to decouple the exhibition structures from the application code, allowing for future exhibition updates without modifying the application;

FR8 - Easily access to exhibition's identifiers - The developer must be able to easily obtain the identifiers for both the exhibition's areas and items;

FR9 - Integrate several location systems - The integration process of different location systems should be simplified for the developer, *i.e.* technical and implementation details for each system should be concealed to the developer;

FR10 - Easily add new location system without rewriting the application - The integration process of a new location system must not imply any application code modifications;

FR11 - Seamlessly management of the necessary location services during a visitation - The developer should not be directly responsible for activate and deactivate each of the location systems supported by the application;

FR12 - Automatically display widgets - The developer should have mechanisms to automatically display widgets on appropriate moments during the visitor's tour.

Visitor

This user class regards all final users that use these applications while visiting an exhibition. The required application's functionalities available to the visitors are stated below:

FR13 - Automatic notifications for needed user actions - The application should automatically notify the visitor when it needs further user actions for specific location systems;

FR14 - Seamlessly turns on the necessary location services during a visitation - The visitor should not be responsible for activating or deactivating the device's location systems while using the application.

3.1.2 Non functional requirements

In addition to functional requirements described above, two non functional requirements can be considered for the proposed solution. By definition, these type of requirements are not crucial for the solution development, but can be considered desirable features and provide and added value for these type of applications.

- **Efficient battery consumption** - regarding the high energy consumption of several device's location systems, particularly GNSSs and cellular networks, it is desirable to minimize the up time of these services. Therefore, the proposed solution could have mechanisms to ensure only the necessary location services are active, by evaluating the current visitor's position within an exhibition;

- **Minimize impact on application size** - The incorporation of the middleware must not result in a noticeable increase on the final application size.

3.2 Architecture

Considering the requirements presented in section 3.1, the two main components of the proposed solution are a schema for the definition of POI structures and a middleware that simplifies the development process of exhibition navigation applications. Figure 3.1 presents a generic architecture of the solution and where these two components are integrated.

The middleware, as its denomination suggests, will be place between the Application layer and the Hardware abstraction layer (HAL). The HAL provides a set of programming interfaces is commonly used in mobile architecture as it allows for a simpler access to important connectivity hardware, *e.g.* GNSSs and Bluetooth. Therefore, an application that uses this middleware will not need to directly interact with the different components of the HAL. The most relevant components of the generic architecture are outlined below.

POI Tree

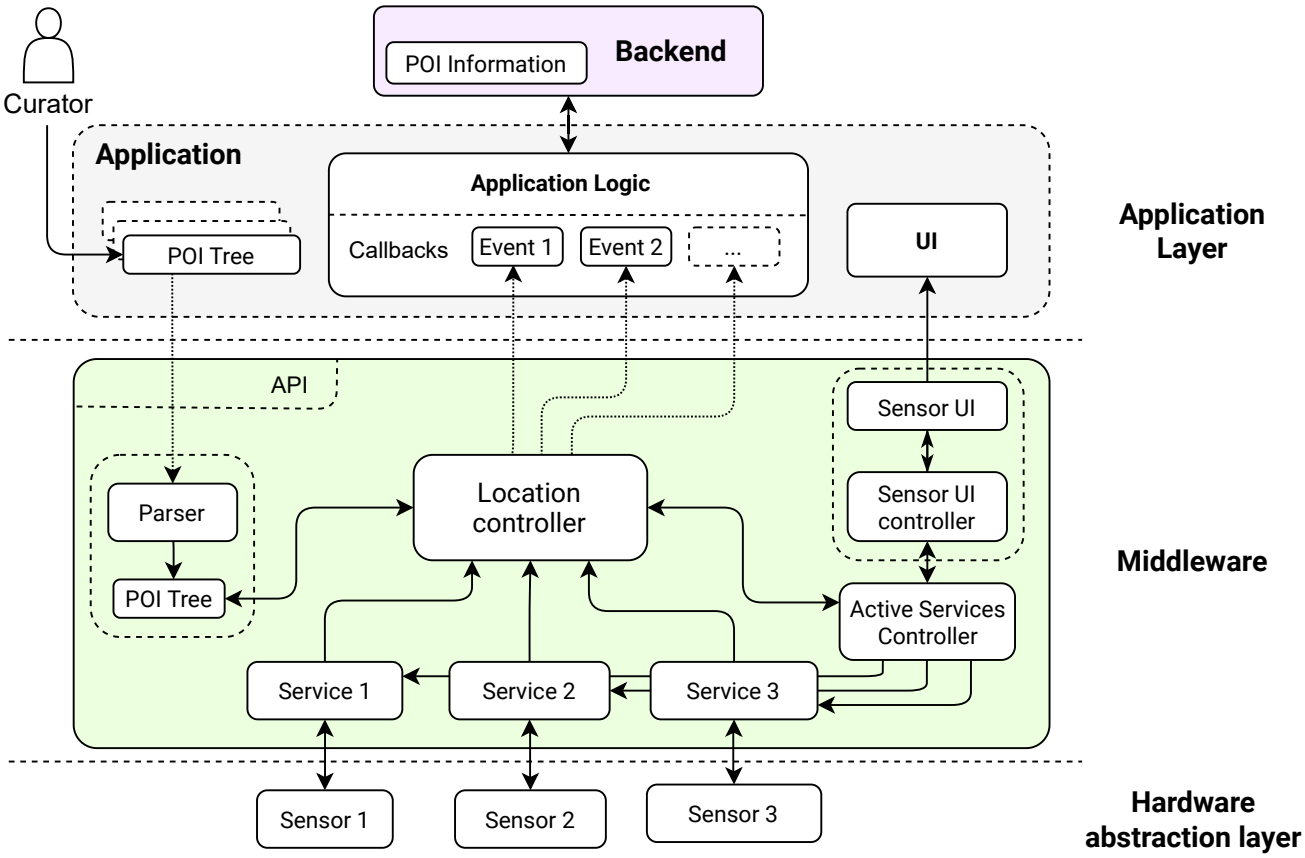


Figure 3.1: Generic architecture for a middleware aimed at exhibition navigation applications.

The generic model for this POI Tree will be described on section 3.3. A set of these structures must be loaded into the middleware by the application via a provided API. The proposed POI structure guideline is designed to be platform independent or, on other words, can be used by several tools, libraries and services. This flexibility enables applications to obtain POI structures from different sources, such as imported files or web services. Thus, as stated before, it is the application responsibility to obtain and load these structure into the middleware.

Parser

After a POI structure is loaded into the middleware, it needs to be parsed into runtime data in order to be easily accessible by the other middleware components. Straightforwardly, the Parser module has decoding mechanisms that obtain data objects from the externally loaded structures. This module also ensures that only compliant structures are loaded into the middleware (as the parsing methods are solely successful with the predefined format).

Location controller

The main purpose of the Location controller is to determine the most relevant POI to the application's user (visitor). The controller continuously receives location updates from several location services and, by analyzing the runtime POI structure, tries to match each update into a POI's area condition (further explained on section 3.3). If the controller determines the visitor is inside a POI region, it becomes the new active POI and an event is sent to the application layer.

Furthermore, upon changing to a new active POI, the Location controller also determines the necessary location services for the newly determined position inside the exhibition. This information is then passed to the Active Services controller.

Active services controller

The Active Services controller manages the activity of all the location services configured on the middleware. Upon knowing from the Location controller which services are necessary to be active, it starts (or maintains) only those and stops the unnecessary ones.

While the majority of the location services on mobile devices can operate in a seamless fashion, *i.e.* automatically provide location callbacks without user interaction, some services might require specific user actions in order to operate correctly. In those cases, when required by the Location controller, the Active Services controller must notify a separate controller in order to request further user action.

Sensor User Interface (UI)

This component of the middleware is responsible for displaying widgets and prompt the user to perform specific actions when needed, *e.g.* simply approach the device to an identifier or scan a code.

There are two main constituents of this module, a controller and the UI resources. The Sensor UI controller receives requests from the Active Services controller describing which necessary user interaction is needed. The Sensor UI controller then invokes the correspondent UI resource that will be displayed to the user and prompting him to perform the required action. After that, the result of the interaction is sent back to the Sensor UI controller and then will be used by the Location controller to assess necessary location updates.

3.3 POI data model

Regarding the identified requirements on section 3.1, specially those related with an exhibition curator, one can derive the requirements of a generic POI data model:

- Hierarchical;
- Support multiple location systems;

- Have a property for POI identifiers;

- Extensible.

Given the coverage of the POI definition, specifically the possibility of being both items or areas, one immediate conclusion is the possibility to have nested POIs. This recursive notion is one of the core motivations for the proposed solution and, consequently, for the data model supporting it. In figure 3.2, a generic data model is presented. The next paragraphs describe each of the constituent entities with a brief explanation about their properties.

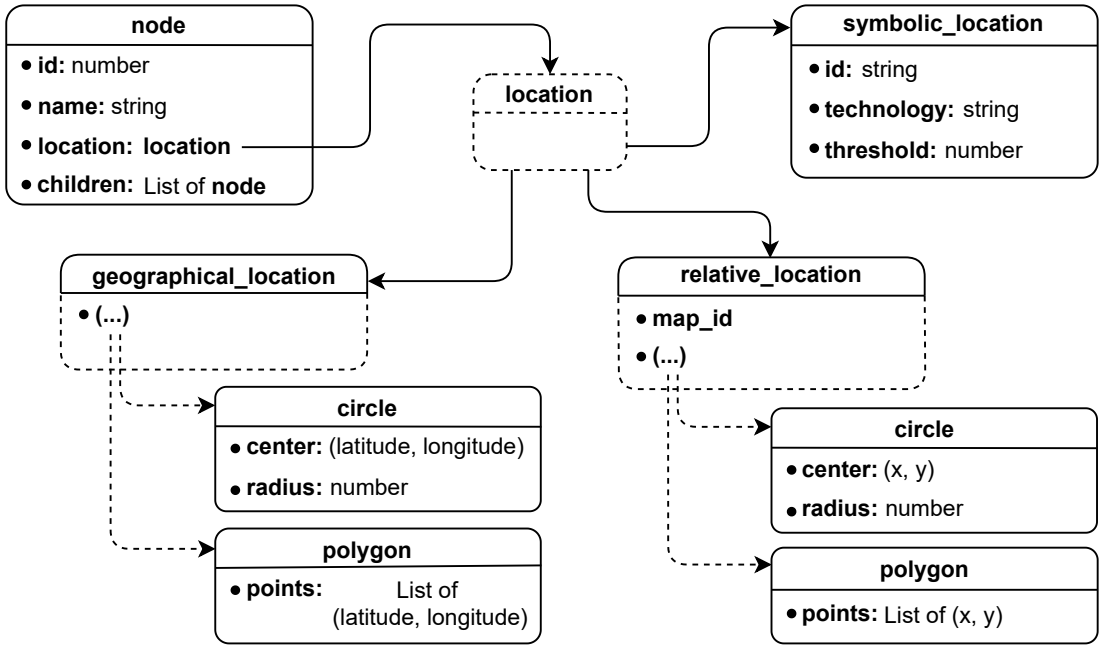


Figure 3.2: Generic POI data model diagram.

3.3.1 POI nodes

To achieve a hierarchical representation for a group of diverse POIs, a recursive approach was taken. The ”node” entity, that represents a POI, is the fundamental element in any POI structure. This element has the ”identifier” and ”name” properties, as well as a property for describing its location (presented on the next subsection). As some POIs can have other POIs inside them, the ”children” property allows the possibility for a ”node” to have a list of inner ”nodes” (with an identical structure). Therefore, using a set of recursive ”nodes”, a tree of POIs can be constructed, with a varying depth accordingly to the corresponding physical layout of the POI group.

3.3.2 Location

As stated above, each POI node has a ”location” property which stores its positioning information. POIs can have different types of location identifiers, generally dependent on the chosen referential, *e.g.* positioned on a map or using geographic coordinates. This multitude of location types imply its support by the data model.

There is another requirement regarding a POI's location: in order to provide the most relevant POI to the visitor, an area of ”significance” must be defined. An initial approach could be to consider the most relevant POI for the visitor, at a given time, to be its nearest POI. Considering this approach, a simple point representation for a POI's location would suffice as the nearest POI could be found by computing the shortest distance to the current visitor's position. However, if a group of POIs is scattered around a large area, the nearest one could still be out of sight and, therefore, not relevant to the visitor. Thus, a more complete approach is to define a region for each POI to be considered relevant to the visitor. In addition, this defined area provides a clear entering/exiting condition for obtaining the activePOIs, that would become relevant on the implementation phase of the solution.

Considering this two guidelines, the ”location” property can be one of three types: absolute, relative or symbolic and each one has properties that define an active region. The differences between these three types of locations are described below.

Geographical locations allow to locate anything on Earth, using geographical coordinates, such as latitude and longitude. In order to support geographical areas for POIs, the ”geographical location” on the proposed data model can be defined by a circle, with a single point and radius, or a polygon, with an array of three or more points.

Relative locations have a similar principle to geographical location but the set of coordinates is relative to a specific referential, such as a building or room. Therefore, instead of using pairs of latitude and longitude, relative locations use 2D coordinates and an identifier for the chosen referential.

Symbolic locations differ from the previous two types by the absence of a unequivocally position on a referential. Instead, they identify an area, generally indoors such as rooms or hallways. This limitation does not hinder its usefulness, as some POIs might not require more precise location descriptions or the facilities themselves cannot provide more accurate location systems. The proposed data model for this location type has an ”identifier” property for the corresponding area and ”technology” and ”threshold” properties for possible technologies differentiation and a threshold condition.

3.4 API

In order to an application use the proposed library, a communication mechanism must be implemented between the two components. The library itself must have a publicly accessible API, composed

of several methods calls that the application integrating the library could execute. On the other hand, the library will also have to provide callbacks or event responses in order to send information back to the application.

As this library will be used as an application component, there is the possibility to have synchronous requests, *i.e.* when the application executes an API method, the response can be directly obtained as the return value of the method. However, as a way to allow more flexibility to both API and applications, all of the API calls do not expect any return value. Instead, the library will trigger several events to the application, containing the requested information. This approach also allows for isolated update events without the need for their explicit requests.

On the next subsections, both the library's API methods and callback events will be described.

3.4.1 Method calls

The available API calls are presented below.

- `init` - This method initiates the runtime components of the library by providing a POI structure. This structure has to comply to the predefined set of rules in order to be initialize correctly;
- `getNearestPOI` - request the library for the nearest POI relative to the visitor.

3.4.2 Callbacks/events

The set of events from the library to the application using it are described below.

- `enteredPOI` - notifies the application that the visitor entered a new POI region;
- `exitedPOI` - notifies the application that the visitor had exited a previously entered POI region, *i.e.* deviated from its defined region;
- `nearestPOI` - returns the identification of the nearest POI to the application.

On a final note, the existence of these callbacks illustrate one of the advantages of this asynchronous approach: the API is able to send ”entered”/”exited” POIs events when the update actually occurs. This eliminates the obligation of periodic checks by the application to know which is the current POI.

4

DynamicPOI Implementation

Contents

Regarding the requirements and architecture design defined on the previous chapter, this chapter describes the implementation process of a solution that attempts to fulfill them. Section 4.1 describes the JSON schema that defines a set of guidelines for POI structures. In section 4.2, an overview of an Android middleware implementation is presented. The defined external location services are described on section 4.3 and how their location updates can be used to determine the most relevant POI are described on section 4.4. Lastly, the section 4.5, defines a simple approach to only activate the strictly necessary location services, according to the visitor's location.

4.1 JSON Schema

It was decided to use a JSON Schema in order to represent the supported data model as well as its properties and restrictions. A JSON Schema is written as a regular JSON file but instead of containing object information it defines the expected structure for the JSON files used by an application or service [57].

As a multi-platform standard by definition, almost every platform has some kind of JSON parsing abilities, via native libraries or third party addons. Thus, the process to generate data structures that comply to the expected input format for the solution is greatly simplified.

Lastly, the existence of an available template for the expected data format allows for *a priori* validations. By validating the input data before loading it into the solution, no time nor resources at runtime will be used in parsing unsupported data. One example of using this preemptive validation are online JSON validators, such as Newtonsoft's JSON validator [58]. By inserting the defined JSON schema, a user can easily validate the compliance of a POI database before loading it into the solution.

As stated above, a JSON schema is defined as regular JSON file. Instead of describing data, the key-value pairs define a set of rules for the expected format of data JSON files. One of the core features of a schema is specifying attributes for an object's properties, *e.g.* its type, if it is required or provide a default value. This is further extended by enabling the definition of entire object structures and even providing conditionally defined structures. Each of these features used on the proposed JSON schema will be further described.

The creation of a JSON schema starts with the definition of several header properties:

- `id` - URI for the schema and is the base URI for other definitions inside it. Generally corresponds to a publicly accessible file;
- `schema` - draft version used on this schema. Draft 7, find date;
- `title` - optional, defines a title for the schema;
- `description` - optional, provides a brief description of the schema's purposes;

After specifying these headers, the schema's typings, custom objects and rules can now be defined. Some of these guidelines can be placed inside the schema's `definitions` property and then be referenced within the schema itself. Each entry on the `definitions` portion on this schema can indeed be considered a "subschema", as the same rules and syntax apply. Summarily, the `definitions` block allows for better schema structuring and avoids code duplication.

The whole proposed JSON is visually represented on figure 4.1. Dotted lines represent direct definition referencing as dashed lines represent definitions attributed via a condition. The most relevant aspects of this schema are described on the next paragraphs.

4.1.1 POI node

The core structure of the schema is the `node` object (listing 4.1). This element solely aims to provide the relationships between several POIs. Each `node` has two required properties: an `id`, storing the unique identifier for the POI and a custom defined `location` reference (described below). In addition, an optional properties are allowed: `name`, to store the POI's designation and `children`. This last property provides the ability for a `node` to have several child `nodes`, sharing the same definition. Therefore, several tree-like recursive structures can be created, using the same building block.

It is important to emphasize that the purpose of this schema is to define hierarchical representations for POIs. Therefore, the defined properties for the `node` block focus on offering a hierarchical relation between POIs and not how to vastly describe them. However, external data sources can be used to map each POI identifier to a more complete description of it.

Listing 4.1: Node definition.

```
"node":{
      "type":"object",
      "properties":{
        "id":{
          "type":"number"
        },
        "name":{
          "type":"string"
        },
        "location":{
          "$ref":"#/definitions/location"
        },
        "children":{
          "type":"array",
          "items":{
            "$ref":"#/definitions/node"
          }
        }
      },
      "required":[
        "id",
        "location"
      ],
```

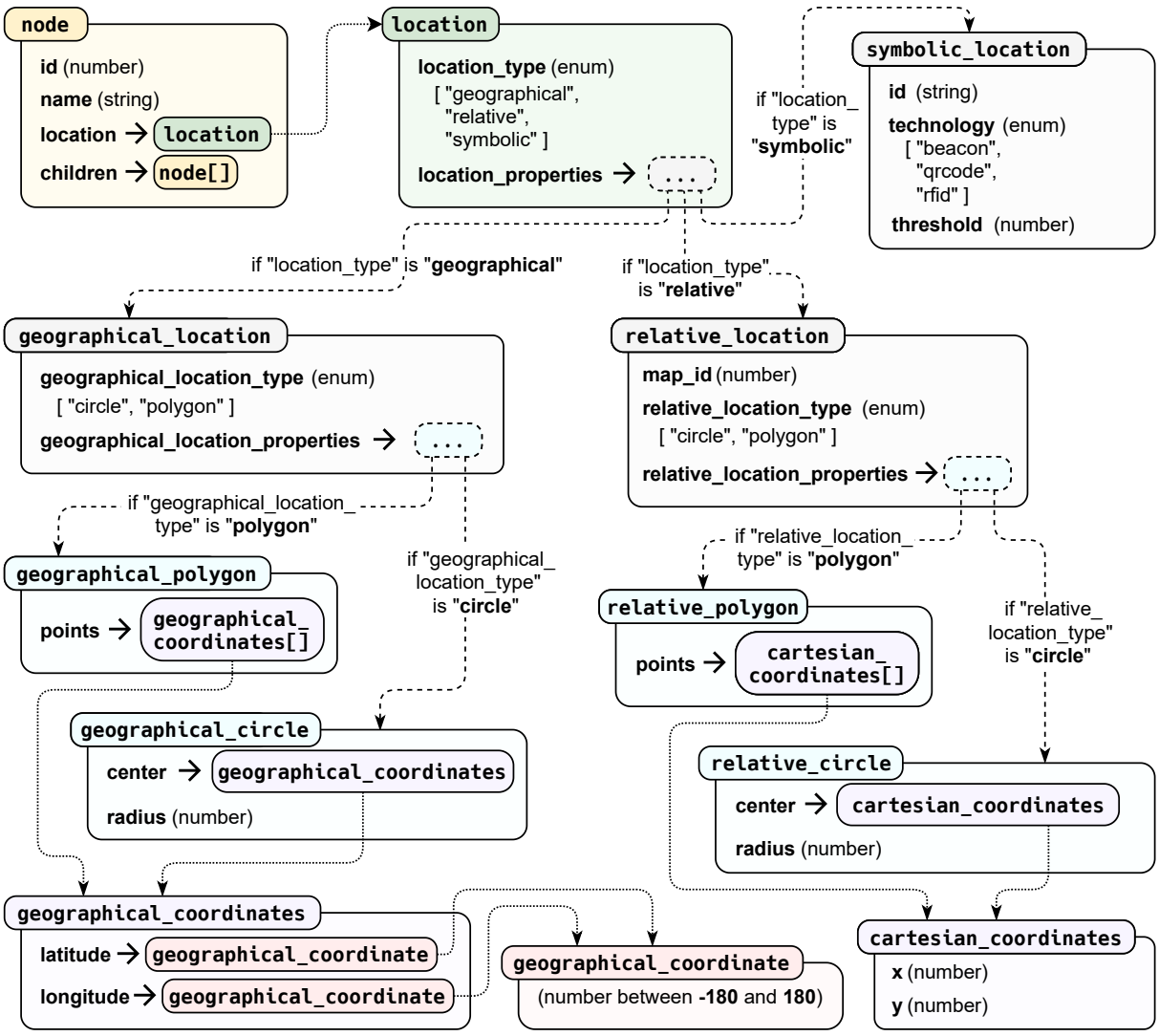


Figure 4.1: JSON schema diagram.

```
24          "additionalProperties":false
25      }
```

The whole schema could be, in fact, defined using the `node` definition as the schema's root element. However, for added flexibility the root element is an array of `node` type objects, called `nodes` (listing 4.2). This allows for one or several first level nodes instead of restricting to one initial node.

Listing 4.2: Schema's root element definition.

```
1   "type":"object",
2   "properties":{
3     "nodes":{
4       "type":"array",
5       "items":{
6         "$ref":"#/definitions/node"
7       },
8       "uniqueItems":true
9     }
10  }
```

4.1.2 Location definitions

The `node`'s `location` property is also a custom defined `location` object (listing 4.3). It has two obligatory properties: location type is defined as a enumerate, *i.e.* can only have a restrict set of values, being those "geographical", "relative" and "symbolic". The `location_properties` property will have a variable definition based on the value of `location_type`. To achieve this, the conditional block `allOf` was used with "`if then`" clauses (each one for each location). These clauses ensure if the `location_type` is one of the allowed values, the `location_properties` will have a specific definition, `geographical_location`, `relative_location` and `symbolic_location`. The `allOf` keyword must be used in this case to validate all of the three possible location "subschemas".

Listing 4.3: Location definition.

```
1   "location":{
2     "type":"object",
3     "properties":{
4       "location_type":{
5         "enum":[
6           "geographical",
7           "relative",
8           "symbolic"
9         ]
10      },
11      "location_properties":{
12
13      }
14    },
15    "allOf":[
16      {
17        "if":{
18          "properties":{
19            "location_type":{
```

```
20              "const":"geographical"
21            }
22          }
23        },
24        "then":{
25          "properties":{
26            "location_properties":{
27              "$ref":"#/definitions/
                    geographical_location"
28            }
29          }
30        }
31      },
32      (...)
33    ],
34    "required":[
35      "location_type",
36      "location_properties"
37    ],
38    "additionalProperties":false
39  }
```

The `geographical_location` objects define a POI's location by delimiting areas using geographical coordinates (listing 4.4). The usage of areas instead of single coordinated points provides a more clear relevance threshold for a given POI. In other words, if the visitor is inside the POI's defined area, it will become the most relevant to him. There are two types of geographical areas supported in the `geographical_location` definition: `geographical_circles` and `geographical_polygons`. To differentiate between these possible geographical location structures, the same approach as the location definition was taken: define an enumerate property, `geographical_location_type`, with the possible values of "circle" and "polygon" and then the `geographical_location_properties` is defined by "`if then`" clauses inside an `allOf` block.

Listing 4.4: Geographical location definition.

```
1   "geographical_location":{
2         "type":"object",
3         "properties":{
4           "geographical_location_type":{
5             "enum":[
6               "circle",
7               "polygon"
8             ]
9           },
10          "geographical_location_properties":{
11
12          }
13        },
14        "allOf":[
15          (...)
16        ],
17        "required":[
18          "geographical_location_type",
19          "geographical_location_properties"
20        ],
21        "additionalProperties":false
22      }
```

The `geographical_circle` object has a coordinates ”center” property and a numeric ”radius” property, while `geographical_polygon` has the property ”points”, an array of coordinates with a minimum of three unique entries (listing 4.5). In both cases, all of properties are obligatory and the coordinates are defined by another subschema, `geographical_coordinates`. This subschema groups a pair of latitude and longitude, that are themselves custom defined as `geographical_coordinate`. This latter definition ensure only valid values for latitude and longitude are accepted, *i.e.* numbers between -180 and 180.

Listing 4.5: Geographical circle and polygon definitions.

```
"geographical_circle":{
  "type":"object",
  "properties":{
    "center":{
      "$ref":"#/definitions/
          geographical_coordinates"
    },
    "radius":{
      "type":"number"
    }
  },
  "required":[
    "center"
  ],
  "additionalProperties":false
},
"geographical_polygon":{
  "type":"object",
  "properties":{
    "points":{
      "type":"array",
      "items":{
        "$ref":"#/definitions/
            geographical_coordinates"
      },
      "minItems":3,
      "uniqueItems":true
    }
  },
  "required":[
    "points"
  ],
  "additionalProperties":false
}
```

The `relative_location` object definition is very similar to the `geographical_location`'s, but uses cartesian coordinates instead of geographical ones (listing 4.6). This coordinate set implies the inclusion of a explicit referential or origin point. Given the multitude of mechanism for maps representation, a more generic approach was taken and the `relative_location` only stores a numeric identifier for a map, `map_id`. Thus, the position is defined by an area relative to a given map and it is the relative location system responsibility to evaluate if this conditions are met.

Listing 4.6: Relative location definition.

```
"relative_location":{
  "type":"object",
  "properties":{
    "relative_location_type":{
```

```
5         "enum":[
6           "circle",
7           "polygon"
8         ]
9       },
10      "relative_location_properties":{
11
12      },
13      "map_id":{
14        "type":"number"
15      }
16    },
17    "allOf":[
```

Similarly to the `geographical_location`, the `relative_location` definition accounts for two possible area definitions, circles or polygons. Once again, "`if then`" conditions inside an `allOf` block ensures that the `relative_location_properties` have the definition specified by the enumerate value of `relative_location_type`. Both `relative_circle` and `relative_polygon` have the same structure of the geographical ones but the coordinate subschema is `cartesian_coordinates` instead of `geographical_coordinates`. This definition simply has two `x` and `y` coordinates that can have any numeric value.

The symbolic location definition describes POIs positions as inside a representative location, *e.g.* a room, instead of a set of coordinates (listing 4.7). Therefore, the only required property for a symbolic location is its identifier (`id`). However, as these type of location can be used with location systems with variable accuracy, *e.g.* BLE beacons, two optional properties are also allowed. The first one, `technology`, is an enumerate type that distinguishes between the major location systems that might use this approach, with the possible values of "beacon", "rfid", "qrcode". To ensure clear enter and exit conditions, similarly to the other two location types, a `threshold` property is supported. Lastly, additional properties are allowed.

A practical example could be a symbolically identified room through a BLE beacon In this representation, the `technology` property would have the value "beacon" and the `threshold` property would have a suitable value for RSSI measures that distinguish when the visitor is near enough of the beacon for this room to be relevant for him.

The full JSON schema definition is available on Appendix A, at the end of this document.

4.2 Android middleware implementation

The proposed solution was implemented on the Android environment (using the Kotlin programming language), given its wide availability on the smartphone market and its well documented libraries and services. Figure 4.2 provides an overview of the implemented solution.

Listing 4.7: Symbolic location definition.

```
"symbolic_location":{
  "type":"object",
  "properties":{
    "id":{
      "type":"string"
    },
    "technology":{
      "enum":[
        "beacon",
        "qrcode",
        "rfid"
      ]
    },
    "threshold":{
      "type":"number"
    }
  },
  "required":[
    "id"
  ],
  "additionalProperties":true
}
```

The middleware component was implemented inside a single library package and by external location services. The orange colored components are runtime components that enable all the middleware's functionalities. The green colored components correspond to location services, each one implemented as a Service class (described on section 4.3). Despite not being included inside the library itself, they play a crucial role by allowing the runtime components to determine the visitor's location relative to an exhibition. Furthermore, the execution of this services is controlled by an Active Services Controller, detailed on section 4.5.

The library's core class is named PoiManager. This class is responsible for aggregating several components, with varying levels of complexity but, most importantly, is the contact point between the middleware and the application layer. In order to operate the developed library, an application must instantiate a PoiManager object and comply to its public interfaces. These interfaces are detailed on section 4.2.1.

The PoiManager is instantiated by the application through an API call containing a POI tree. In order for it to be available at run-time, it must parsed into memory. This process is detailed on section 4.2.2 and its outcome is then stored on a TreeController instance, described on section 4.2.3. Besides storing an object containing the entire POI tree, this controller helps determining the most relevant POI to the visitor. The algorithms that constitute the Location controller and ascertain the most relevant POI for the visitor, based on the location callbacks sent by the location services, are presented on section 4.4.

The PoiManager class was also developed to facilitate its integration on the application layer by

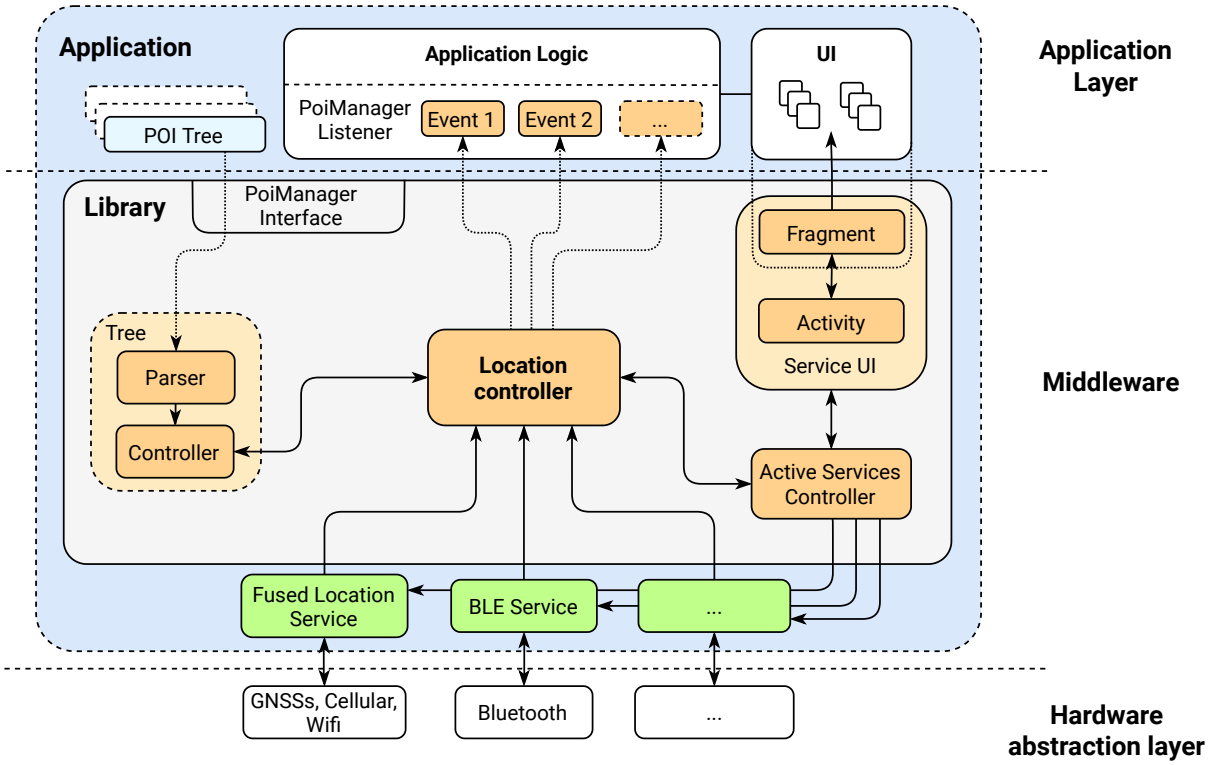


Figure 4.2: Diagram of the implemented Android middleware.

providing two useful features. The first one the integration of a Context wrapper, which simplifies the Context management by the application, specifically when starting and stopping external services. The other feature is an integrated Permissions manager: the necessary Android environment permissions for the library's runtime components and services are already defined on the library's manifest and can be automatically requested to the user.

Lastly, the ability to include services that incorporate explicit user interactions on their operation is provided by the Service UI component, described on section 4.2.4.

4.2.1 API calls and events

In order to define a bidirectional communication between the proposed middleware, two public interfaces were defined on the library package:

PoiManagerInterface

The purpose for this interface is to define which are the publicly accessible methods for the middleware and their respective parameters. This interface defines two method calls:

- **init(jsonContent)** - responsible for the middleware initialization process. A POI tree is provided

via the ”jsonContent” parameter, which will be parsed into runtime objects;

- **retrieveCurrentPoi()** - requests the middleware for the most relevant POI, considering the current visitor's position.

PoiManagerListener

On the other hand, this interface specifies the callbacks' format sent by the middleware to the application layer. This callback interface defines three events:

- **enteredNode(nodeId)** - triggered event each time a new node location condition is met, updating the current POI;
- **exitedNode(nodeId)** - triggered event each time the determined visitor's location invalidates a previously active location condition, *i.e.* when a node ceases to be the most relevant for the visitor;
- **currentPOI(nodeId)** - returns the current POI's identifier, if it exists.

The application developer must create an instance of the PoiManager in order to instantiate the middleware components. As the PoiManager class implements the PoiManagerInterface, after the application creates its reference, the PoiManagerInterface's public methods can now be invoked on the application side. On the other hand, the developer must ensure the class which will hold the PoiManager instance implements the PoiManagerListener interface's methods. Therefore, upon being initialized, the PoiManager object will hold a listener reference corresponding to the application component that created it. This listener reference will then be used to invoke the defined callbacks, resulting on the invocation of the implemented methods on the application.

4.2.2 JSON parsing

The first step to correctly instantiate the runtime components of the solution, JSON information containing a POI hierarchy must be loaded, via the provided API. Attempting not to impose heavy restrictions on the encoding format expected by the library, the JSON content is passed simply as a UTF-8 encoded string. Thus, applications utilizing this middleware can obtain the JSON POI structures by different means, *e.g.* external files or web services, and just have to guarantee its content is simply converted to a string.

After obtaining the JSON content, it needs to be parsed into data objects, stored at runtime. The parsing process must match the schema's restrictions defined above in order to guarantee the correct operation of the solution's components.

The Android environment has several JSON parsing libraries and all share the same operation core: provide adapters that can serialize and deserialize JSON content. By directly supporting the Kotlin

language and having a straightforward syntax, the Moshi library was chosen to implement the JSON parsing procedures.

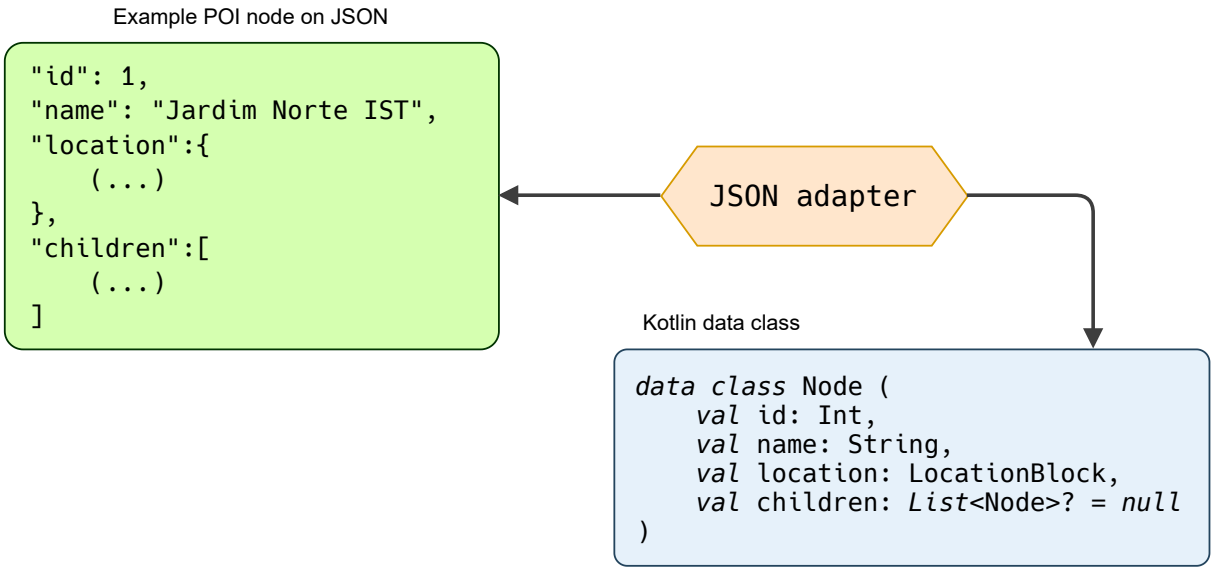


Figure 4.3: Example of the purpose of JSON adapters.

As illustrated on figure 4.3, the JSON adapters can convert between JSON information and data class objects. If the data classes share the same structure and property names as the JSON ones, a direct mapping can be performed. In these conditions, the Moshi library automatically provides the necessary adapters for the parsing process.

However, as the established JSON structure for POIs defines polymorphic properties, *i.e.* some object properties might have varying inner structures, specifically `location` and its sub-types, custom adapters had to be built to mirror the conditional behavior defined of the JSON schema. Concisely, when parsing JSON to an object, these custom adapters peek the value of the ”type” properties (*e.g.* `location_type`) and invoke the correct mapping for the corresponding data class.

After defining the direct and custom adapters for all expected JSON properties, a ”Tree” class object is obtained and stored in memory while the library is operating. As expected, this object holds the entire POI hierarchical representation and can be easily accessed to determine the visitor’s position within it.

4.2.3 Tree controller

One fundamental requirement for real time analysis of the loaded POI tree is having a structure that keeps track not only of the currently active POI but also of full path from the tree’s root to that node.

For simplicity purposes, a TreeController class was defined to store both the run time POI tree and the current node’s path as a private ArrayList of Node objects. Upon initialization, the PoiManager will

hold a reference to a TreeController object, which has the following public methods:

- **currentNode()** - returns the current node, if it exists. If the the ArrayList is empty, a null value is returned instead;
- **enteredChildrenNode(node)** - adds the supplied Node parameter to the ending of the ArrayList;
- **exitedNode(node)** - removes the supplied Node parameter from the ArrayList;
- **findParentCurrentNode(node)** - attempts to find the parent of the supplied node parameter, *i.e.* the node that was inserted immediately before the parameter node;
- **resetCurrentNode()** - clears the ArrayList by removing all of its elements.

These methods have a high relevance when determining which is the most relevant POI node to the visitor, given its location and consequent location callbacks sent by the location services.

4.2.4 Service UI

While the majority of the location services can operate seamlessly in the application's background and not requiring any user interventions, some services rely on user interactions with his surroundings to operate properly. The most common examples of this type of services are services based on short-range identifiers, such as QR codes or NFC tags.

Given the user interaction requirement for this type of services, some UI elements must be defined which will be presented to the visitor when needed. Regarding the Android environment, Fragments and Activities can be defined directly on the library package and then be accessed and displayed by the application. The interaction between these two types of components is described on figure 4.4

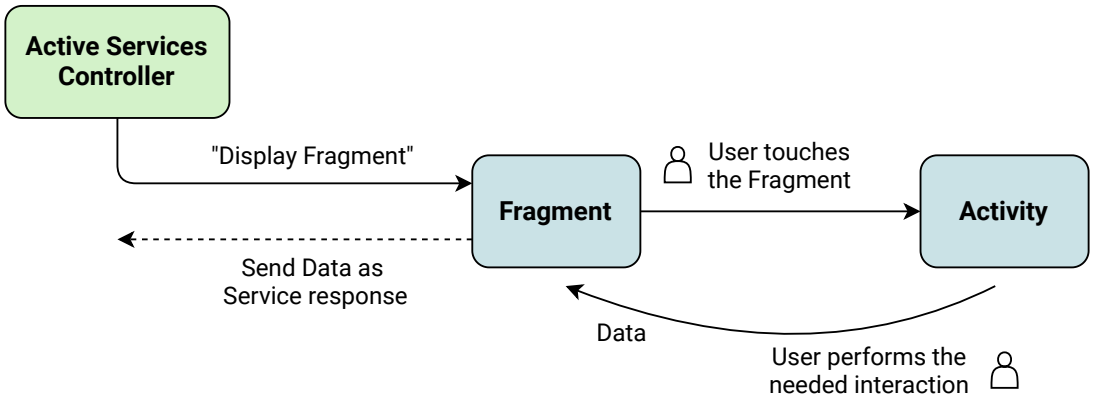


Figure 4.4: Interaction between Service UI components.

Regarding the implemented architecture, the defined Fragments could receive requests sent by the Active Service Controller in order to be visible (or not) on the applications UI. Then, upon user interaction, the Fragment could invoke extra Activities or execute the remaining operations to obtain the required information for the location service. Lastly, this information is sent back to the runtime library components and used as a regular update callback for a location service.

To include these components into an application, the developer simply has to define a placeholder on the application's UI layout indicating where the desired Fragment will appear (when necessary) and instantiate its class as a regular UI component. As stated above, the remaining logic of this component is implemented on the middleware, oblivious to the application's developer.

4.3 Location services

As stated on the previous sections, the first step to determine the visitor's location inside an exhibition is to obtain several location updates from different location systems. In order to achieve this, the PoiManager is responsible for managing various location services and receive their location updates.

The simplest mechanism to incorporate services on the Android environment is through service classes and receivers. With this approach, the service is independent from the component that started it and updates from the service are captured by a receiver. This two entities are further explained below:

- Service class - defines the service itself which can be started by calling the method startService(). Two on the most relevant methods available on Service classes are the onStartCommand() and onDestroy() methods, which can be overwritten to allow code execution when starting or stopping the service, respectively. When needed to provide any kind of updates, the service can broadcast events. To correctly identify the purpose and expected structure of these events, it is common to define public constants, used to characterize both ”Intent” objects and their internal properties (named ”extras”), which contain the update data. These Intent objects can then be broadcast;

- Broadcast receiver - in order to receive broadcast updates, a BroadcastReceiver must be registered. Upon registering the receiver, an IntentFilter is also provided, which specifies the relevant Intents for this receiver. Thus, by registering the receiver to receive the Intents defined by a specific service, the receiver will be able to process all of the service broadcast events.

Therefore, the services developed on this solution will follow the pattern presented above. In terms of location typing, two services were incorporated into the solution: one for obtaining geographical locations and other for obtaining estimated distances to nearby BLE beacons. These two services (and their receivers) are detailed on the following paragraphs.

4.3.1 FusedLocationProviderService

As stated on chapter 2, the Google Location Services API has a ”fused location provider” that seamlessly combines several location systems. A FusedLocationProviderClient can be easily configured on several Android application’s components in order to obtain periodic location updates from the Google Location Services.

Despite being commonly used on UI elements, the FusedLocationProviderClient can also be initialized inside any service class. Therefore, a custom FusedLocationProviderService class was created and the FusedLocationProviderClient was configured using the LocationRequest and LocationCallback objects, described below.

The LocationRequest object is used to configure the operation parameters of the FusedLocationProviderClient. Nevertheless, this location provider can fulfill several location requests at the same time, *e.g.* corresponding to different applications. A LocationRequest can specify the following parameters:

- **Priority** - specifies the operating mode for the LocationRequest, balancing between accuracy, power consumption and (indirectly) used technologies. In this case, value was set to PRIORITY_HIGH_ACCURACY, which attempts to obtain the maximum accuracy at the cost of combining several location services (GNSS, cellular networks and WiFi);
- **Interval** - specifies the expected time interval to receive location updates from the FusedLocationProvider, set to 4 seconds;
- **Fastest Interval** - given the shared nature of the FusedLocationProvider, some location updates might be obtained at a higher rate than the specified ”Interval” property. This parameter defines the maximum interval at which location updates will be sent to this LocationRequest. This value was set to 2 seconds.

LocationCallback objects provide a set of methods that can be invoked by the FusedLocationProviderClient, on pertinent location updates. In order to enable these method calls, a LocationCallback object is supplied when requesting location updates on the FusedLocationProviderClient.

Regarding this custom service, the most relevant method given by LocationCallback object is the onLocationResult. This method is called when new locations are available, which are placed inside a LocationResult object. Therefore, a custom LocationCallback was created with a modified onLocationResult method which iterates through the LocationResult and calls a custom function named ”onLocationChange”. This function as simple purpose: encapsulate Location objects inside an Intent and broadcast it.

As stated at the beginning of this section, a service can define public identifications for its broadcast Intents, to facilitate its reception on the correspondent receiver. Thus, the FusedLocationProviderService

defines a single Intent named ”LOCATION_CHANGED”, which can have an Location object (identified by ”EXTRA_LOCATION”) as its property.

On the PoiManager side, a BroadcastReceiver was defined to capture the location events sent by the FusedLocationProviderService. When these broadcast events are received, the Location object is extracted from the Intent and used by a mapping algorithm to determine the most relevant POI (explained on section 4.4).

4.3.2 BeaconService

Aimed at providing a location service for symbolically identified indoor areas, a service for evaluating BLE beacons' proximity was implemented. Based on the AltBeacon library [47], this service can estimate the distance to a set of beacons and determine which one is the nearest to the visitor. Instead of relying on a single service class, this service was divided into two components: the service itself and a separate thread, described on the following paragraphs.

Similarly to the previous service implementation, a service class was created named ScannerService. This class' purpose is to configure the operation of the external AltBeacon library. As mentioned on section 2.5.2, several identifier formats can be used on BLE beacons. Thus, the first step is to define the expected bacon identifier format for the AltBeacon library. It was decided to use the the Eddystone format, an open standard developed by Google. The definition of this format on the library's parser results on only being able to scan beacons that comply to it.

After defining the expected scanned format, the service is binds the library which enables a callback method (didRangeBeaconsInRegion) whenever the nearby beacons change. In order to maintain a runtime reference of the scanned beacons, two hash-maps were constructed (both indexed by the beacon's identifier) to store a time stamp of the last detection and an estimate of the distance to the visitor's device. Then, when the callback method is executed, both has-maps are updated. However, this method does not guarantee to provide updates at a regular time intervals.

To overcome the lack of regular updates to the beacon hash-maps, a separate tread was created that performs a set of actions at a constant time interval. The ScannerService initializes the thread, which saves a reference to the running service, allowing access to the nearby beacons' information and the service's public methods.

The thread was defined to verify the beacon's hash-maps at every 3 seconds and perform two tasks:

1. Removes hash-map entries corresponding to beacons not scanned on the last 9 seconds
2. Computes the nearest beacon to the user and prepares the a broadcast Intent containing it. If the hash-maps are empty, *i.e.* no beacons nearby, a specific broadcast is also constructed.

Regarding the necessary Intent definitions for this service, the following Intents were defined on the service class:

- ACTION_BEACON_BROADCAST - after evaluating both hash-maps, if the the running thread is able to determine the nearest beacon to the device, this Intent is used to broadcast its information. Two ”extras” are also defined to incorporate both the beacon’s identifier (EXTRA_ID) and its estimated distance (EXTRA_DISTANCE)
- ACTION_BEACON_NONE - when no beacon information is available at the service’s hash-maps, a broadcast containing this Intent is sent

Once again, a BroadcastReceiver was defined on the PoiManager, allowing the reception of the nearest beacon updates and consequent evaluation against the POI structure.

4.4 POI algorithms

Upon receiving a location update from a location service, algorithms must search the POI tree for the visitor’s most relevant node. Consequently, these algorithms are responsible for updating the tree controller’s current node reference. Given the flexibility (and consequent unpredictability) for the possible POI trees, the search algorithms must be independent of both trees’ depth and width.

On the following subsections, the matching algorithms for geographical locations and symbolical beacon locations are described. Despite both sharing a core operating principle, the inherent differences between these two types of locations imply different matching mechanisms.

4.4.1 Geographical nodes

Before the algorithm definitions, some assumptions were taken for the geographical location typing and its consequences on the nodes’ matching conditions:

1. Only geographically described nodes can be evaluated, *i.e.* of the current node does not has this location type, the geographical location update is irrelevant to assess its validity or search for a new current node.
2. A single geographical location update, *i.e.* a pair of geographical coordinates can match several POI nodes’ area conditions. These multiple possible matches imply some kind of recursive search algorithms to be applied in order to find the lowest tree level node whose area condition is matched.
3. Solely considering hierarchical well defined geographical areas, *i.e.* children nodes’ areas are contained inside their parent node area. Despite this condition is not strictly evaluated when defining the POI tree, the correct operation of the following algorithms is only guaranteed in these cases.

The whole evaluation algorithm for geographical locations can subdivided into three parts (a main algorithm and two inner search one), each one implemented by a single method. It is important to note that the whole algorithm is executed after each received geographical location update that will be used in all geographical area evaluations.

Starting with the main algorithm for geographical locations, executed after a successful location update sent by the FusedLocationProviderService, the first step is to evaluate the existence of "currentNode", *i.e.* an already active POI. If it exists, the algorithm will evaluate if the newly received location update matches any area conditions, starting on the current node. If no current node is found, a simple depth-first search is performed starting from the tree's root nodes. The main algorithm's steps are present on algorithm 4.1 and each relevant section will be detailed on the following paragraphs.

Algorithm 4.1: Main algorithm for evaluating geographical nodes' location conditions.

```
MatchGeographicalNodes (geographicalCondition)
    input: A geographicalCondition based on a set of geographical coordinates.
    currentNode ⟵ treeController.currentNode
    if currentNode is not null then
        if currentNode is not a geographical node then
            return
        if currentNode matches geographicalCondition then
            childNode ⟵
              SearchGeographicalNodeChildren(currentNode, geographicalCondition)
            if childNode is not currentNode then
                enteredNodeEvent()
            else
                return
        else
            newNode ⟵
              SearchGeographicalNodeParent(currentNode, geographicalCondition) if newNode
              is not null then
                enteredNodeEvent()
            else
                treeController.resetCurrentNode()
                exitedNodeEvent(currentNode.id)
    else
        foreach topNode ∈ tree.nodes do
            if topNode does not match geographicalCondition then
                continue
            else
                tree.Controller.enteredChildrenNode(topNode)
                SearchGeographicalNodeChildren(topNode, geographicalCondition)
                 enteredNodeEvent()
```

As presented on Algorithm 4.1), the main algorithm relies on two fundamental sub-algorithms: one

for searching the tree downwards (children nodes) and another for searching upwards (parent nodes). Both are designed to be called recursively (a necessary requirement as no depth limit is imposed for the POI structures) and their return value is fundamental to assess their search outcome.

Starting by the downwards search algorithm (Algorithm 4.2), it receives two parameters: the starting node and the geographical location. If the starting node as a group of child nodes, the algorithm iterates through it, evaluating if the geographical location matches any child nodes' area condition. In the event of a match, the treeController's current node is updated and the search continues one level lower, calling the algorithm recursively on the newly matched node and returning its value. The recursive nature of the downwards search algorithm effectively assures an depth-first search on each nodes' sub-tree, updating the current node every time a lower level node matches the location.

If the stating node does not have any children or after searching the entire node's sub-tree, the algorithm will exit by returning the starting node reference itself. Remembering the if clause on the main algorithm that compares the return result of the top most call of the searchChildren with the current node reference, the purpose of the return value is now clearer. If the starting node reference is returned, it means no lower level nodes matched the location condition. If a new node reference is return, a new currentNode is determined (already updated when searching downwards) and an event needs to be sent to the application layer.

Algorithm 4.2: Downwards search sub-algorithm for geographical nodes.

SearchGeographicalNodeChildren $(node, geographicalCondition)$

inputs: A $node$ from which to start a downwards search and a $geographicalCondition$ based on a set of geographical coordinates

output: If found, a new node that matches the $geographicalCondition$, the original $node$ otherwise.

$children \longleftarrow node.children$
if $children$ is $null$ **then**
  return $node$

foreach $child \in children$ **do**
  if $child$ matches $geographicalCondition$ **then**
    $tree.Controller.enteredChildrenNode(child)$
    return $SearchGeographicalNodeChildren(child, geographicalCondition)$

return $node$

The parent node (or upwards) search algorithm shares some similarities with the downwards search one: it is also designed to be recursive and its return value is also important for assessing its search outcome (described on Algorithm 4.3).

However, while a downwards search algorithm is based on one or multiple geographical area matches, when searching upwards, it is implicit that (at least) the current node's area condition is no longer valid. Therefore, the first step when starting the upwards (or parent) search algorithm is to verify if it possible

Algorithm 4.3: Upwards search sub-algorithm for geographical nodes.

```
SearchGeographicalNodeParent (node, geographicalCondition)
    inputs: A node from which to start a upwards search and a geographicalCondition based on
            a set of geographical coordinates
    output: If found, a new node that matches the geographicalCondition, null otherwise.
    parentNode ⟵ findParentNode(node)
    if parentNode is null then
        return null
    treeController.exitedNode()
    if parentNode matches geographicalcondition then
        foreach child ∈ parentNode.children do
            if child is node then
                continue
            if child matches geographicalCondition then
                tree.Controller.enteredChildrenNode(child)
                return SearchGeographicalNodeChildren(child, geographicalCondition)
        return parentNode
    else
        newNode ⟵ SearchGeographicalNodeParent(parentNode, nodeCondition)
        if parentNode is not null then
            return newNode
        else
            tree.Controller.exitedNode(parentNode) return null
```

to access the current node's parent. If affirmative, the algorithm starts by removing the no longer valid current node reference from the tree controller. Thus, the current node becomes the previous active node's parent, for now on referred as "parent node".

The next step is to evaluate if the parent node matches the area condition. If it does, a downwards search can be performed to further evaluate possible children matches, with special care to skip the now invalid starting node. Once again, the downwards search algorithm is used on a recursive fashion, updating the current node reference accordingly. After this search, if no new match is found on the parent node's children (besides itself), return this parent node reference.

The only missing step is when the parent node's area condition is not successfully matched by the received location. In this case, the algorithm tries to search upwards, calling itself recursively. If it obtains a new node, its reference is returned, if not, the parent node reference is removed and then returns a null value.

Once again, the return value is a valuable information for determining the inner searches' success on the main algorithm. In this case, if the upwards search algorithm returns a null reference, no parent was found for the current node *i.e.* the current node is a top level node. Combining this with the current

node's area condition becoming invalid with the newly received location implies that the tree's controller must be reset and an exit event must be sent to the application layer. On the other hand, when the parent search algorithm returns a node reference, it means that a new valid node was found (being a parent or any of its children) and a enter event must sent.

Going back to the final section of the main algorithm (Algorithm 4.1), when no current node exists, the tree's root nodes must be evaluated. A foreach loop iterates through the top level nodes, evaluating their area condition. If a valid node is found, it becomes the new current node and a recursive downwards search is executed. Despite the existence of further matches on lower level nodes or not, an enter node event will be sent to the application layer (as at least a top level node became the current node).

4.4.2 Beacon nodes

As described on section 4.3.2, the events sent by the BeaconService contain nearest beacon's identifier and an estimate for the distance to the device. Regarding the corresponding nodes' location structure (symbolic with the ”technology” property equal to ”beacon” and a ”threshold value), it is clear that two evaluations must be performed when searching the tree:

1. Beacon identifier - the received identifier must be matched to a node's symbolic location identifier;
2. Beacon distance - the received distance estimate must be lower than the node's threshold value for it to be considered relevant to the visitor.

In addition, it it clear that only a single matching node is expected to be found on the whole tree. Once again, the uniqueness of symbolic identifiers is not explicitly enforced by any schema rule, it is just an practical assumption when defining the POI tree. This symbolic identifier singularities will result on slightly different recursive methods, when compared to the geographical location ones. Summarily, when the matching node is found, no further searches are necessary.

On the geographical algorithms, one of the initial operational conditions was only evaluating geographical area conditions when the current node has a geographical location definition. Given the symbolic nature and the operating distances for the BLE beacons' technology, some node type flexibility was added when searching the tree. This subject will be further explained on the following paragraphs as well as justifications behind some decisions.

The main algorithm is presented on Algorithm 4.4. If a current node was already defined, the first step is to evaluate its typing. If it is a beacon described node, the two validations stated above are performed. Considering the case when received beacon identifier is the same as the current node's, if the estimated distance is still lower than the threshold, no further action is needed (the condition is still valid). If the distance surpasses the threshold, the current node is no longer valid and the tree will be

reset. Considering the opposite case, *i.e.* the received identifier belongs to another node, more complex evaluation steps are needed.

Similarly to the geographical algorithms, a recursive approach was implemented for searching the tree downwards for beacon nodes (algorithm 4.5). This downwards search algorithm was defined to only consider beacon defined children nodes. Despite being impractical to have hierarchical symbolic locations all defined through BLE beacons, the usage of recursive algorithms can support both single level beacon nodes as well as multi leveled beacon structures.

In order to correctly reflect the node path between the tree's root and the current node, the child iteration loop preemptive adds its reference to the tree's controller. If a match is found on any of this downwards search algorithm, the current node was already updated and the algorithm returns its reference. If not, the reference is removed and another child node is evaluated. If none of the starting node's children matched the received beacon's parameters (or there aren't any children), this algorithms returns a null reference.

Considering the cases where none of the lower level nodes match the received beacon conditions, higher level nodes can be evaluated. One of the implicit conditions for these algorithms was in fact to only evaluate nodes whose location structure corresponds to the received location typing. However, a more practical approach was considered when defining the upwards search algorithm.

Considering a common situation where a building, defined by a geographical area, has several rooms, symbolically defined by BLE beacons, one can assess that the building's node will have several children beacon nodes. Furthermore, when visiting an exhibition with this layout, the most recurrent situation is to go through several rooms, without leaving the building. If the typing restriction was applied when searching upwards through the tree, *i.e.* only be allowed to evaluate beacon nodes, when the visitor changes between rooms, the tree's controller will be reset after leaving the first room, the main building must be matched again and only then the second room would be considered.

In order to provide a more efficient operation in these conditions, it was defined the possibility to evaluate the current node's parent's children, even if it is not a beacon node. This addition is reflected on the main beacon search algorithm (Algorithm 4.4), where the current node and its children do not match the received beacon parameters. The upwards search algorithm for beacon nodes is detailed on Algorithm 4.6.

It is important to point out that this algorithm can only be recursively executed for beacon typed nodes, despite the fact that the first call (on the main algorithm) ignores the current node's parent typing. This ensures a simple search mechanism for finding beacon ”siblings” to the (now invalid) current node but does not allow an unrestricted upwards search (very inefficient and contradicts the whole location typed search algorithm guidelines).

A final remark regarding the search downwards algorithm call, which uses both the parent node and

Algorithm 4.4: Main algorithm for evaluating beacon nodes' location conditions.

```
MatchBeaconNodes (node, beaconCondition)
    input: A beaconCondition containing an identifier and estimated distance;
    currentNode ⟵ treeController.currentNode
    if currentNode is not null then
        if currentNode is a beacon node then
            if currentNode is beaconCondition.Identifier then
                if beaconCondition.Distance matches currentNode.threshold then
                    return
                else
                    treeController.resetCurrentNode()
                    exitedNodeEvent(currentNode.id)
            else
                child ⟵ SearchBeaconNodeChildren(child, beaconInformation)
                if child is not null then
                    enteredNodeEvent()
                    return
                else
                    newNode ⟵ SearchBeaconNodeParent(currentNode, beaconInformation)
                     if newNode is not null then
                        enteredNodeEvent()
                        return
                    else
                        treeController.resetCurrentNode()
                        exitedNodeEvent(currentNode.id)
        else
            if SearchBeaconNodeChildren(child, beaconInformation is not null) then
                enteredNodeEvent()
            return
    else
        foreach topNode ∈ tree.nodes do
            if topNode not a beacon node then
                continue
            if topNode matches beaconCondition then
                tree.Controller.enteredChildrenNode(topNode)
                enteredNodeEvent()
            else
                if SearchBeaconNodeChildren(child, beaconInformation is not null) then
                    enteredNodeEvent()
```

Algorithm 4.5: Downwards search sub-algorithm for beacon nodes.

```
SearchBeaconNodeChildren (node, beaconCondition, skipNode)
    inputs: A node from which to start a downwards search, a beaconCondition containing an
            identifier and estimated distance and an optional skipNode reference to a node to be
            skipped during the search;
    output: If found, a node that matches the beaconCondition, null otherwise.

    children ⟵ node.children
    if children is null then
        return null

    foreach childNode ∈ children do
        if childNode is skipNode or childNode is not a beacon node then
            continue

        treeController.enteredChildrenNode(child)

        if child matches beaconCondition then
            return child
        else
            newChild ⟵ SearchBeaconNodeChildren(child, beaconInformation)
            if newChild is not null then
                return newChild

        treeController.exitedChildrenNode(child)

    return null
```

the starting node as parameters in order for the downwards algorithm to be able to skip the starting node.

Related to the previously described practical case, when the current node is not a beacon described one, it is only allowed to search downwards (for beacon nodes). This results on the ability to evaluate symbolically defined children nodes for any kind of node. If this type flexibility were not to be considered, only single type POI tree could be considered which again defeats the whole diversity of location identifiers when describing POIs.

Lastly, when no current node was previously determined, the tree's root nodes are evaluated recurring the downwards search algorithm.

4.5 Active services' management

As mentioned on previous sections, one of the most fundamental components of this middleware is an Active Services controller. Its purpose is to ensure only the necessary services are activated, given the current visitor's location. Considering the previously described services' implementation, this controller must start (and stop) the services' classes and register (and unregister) the correspondent

Algorithm 4.6: Upwards search sub-algorithm for beacon nodes.

SearchBeaconNodeParent $(node, beaconCondition)$

inputs: A $node$ from which to start a upwards search and a $beaconCondition$ containing an identifier and estimated distance;

output: If found, a node that matches the $beaconCondition$, $null$ otherwise.

$parentNode \longleftarrow findParentNode(node)$

if $parentNode$ is $null$ **then**

  return $null$

$treeController.exitedNode()$

$childNode \longleftarrow SearchBeaconNodeChildren(parentNode, beaconInformation, node)$

if $childNode$ is not $null$ **then**

  return $childNode$

else

  if $parentNode$ is a beacon node **then**

    return $SearchBeaconNodeParent(parentNode, beaconInformation)$

  else

    return $null$

receivers.

Instead of creating a separate component to implement this controller, it was incorporated inside the PoiManager class, mainly for simplicity purposes but also to maintain the receivers' instances on the PoiManager running object. The active services controller operation is described on figure 4.5.

The first requirement for this controller is to store which services are currently active. In order to achieve this, a simple hash-map was created, where the keys are the service classes and the values are booleans. Thus, when needed to assess if a given service was started, it is only necessary to evaluate the boolean value of the hash-map for that service.

Secondly, some mapping algorithms must be defined in order to obtain the correspondent service class for a given location type. Considering the implemented services on the middleware, a correspondence was made between the geographical location objects and the FusedLocationProviderService class and between the symbolical location objects with ”beacon” technology and the ScannerService class.

Finally, the only missing step is to determine which location types are relevant given the current node. In order to be able to detect the current node's exit condition and its children (if exist) enter conditions, a list containing the current both node's and its children location types is built. This list is then mapped to a list containing the corresponding service classes which will be used to only activate the necessary services and their respective receivers.

After guaranteeing only the necessary services and their respective receivers are active, all the required service callback can be obtained in order to assess the validity of the current node's location

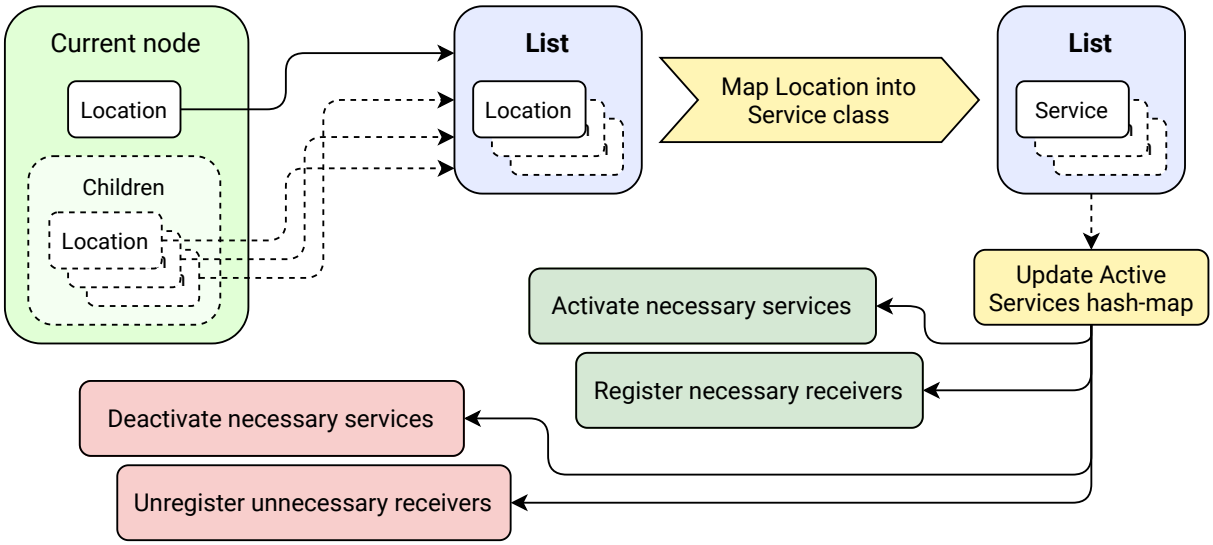


Figure 4.5: Active services' controller operation.

condition or, if it becomes invalid, to determine a new current node.

5

Demonstration

Contents

This chapter describes two different approaches in order to demonstrate the proposed middleware capabilities. Section 5.1 describes how a new user interaction based location service can be integrated into the solution, specifically, QR codes' scanning. Section 5.2 presents a demonstration application for operating and configuring the middleware parameters as well as display its events.

5.1 QR code scanning integration

As stated on the previous sections, some location services for the most relevant POI identification require some kind of user interaction to function properly. However, the proposed middleware is placed below the application layer and can not directly display any kind of UI without recurring to it. Therefore, the integration of a QR code scanning service without having to modify the application behavior required a more complex approach, when comparing to the other two location services implemented.

The different steps to implement this functionality are described on the next sections.

5.1.1 UI components

Regarding the UI components for a QR code scanning service, at least two elements must be considered: a user clickable button, which will be displayed when QR codes are available, and a camera scanning activity that decodes QR codes directly from the camera's viewfinder. Both of these components were defined inside a separate package on the library package.

IndicatorFragment

Buttons on the Android development environment can be implemented by several classes, according to its designed functionalities. A fragment class was chosen in this case, as it can ensure similar layout integration as a regular Views but can hold references to other important components of an application, such as service receivers (further explained on section 5.1.2).

In terms of UI, this fragment's layout holds a QR code icon inside a background element. Regarding its code functionality, a click listener was added to the button's layout, which launches the scanning activity, expecting a QR code result sent by this activity when it terminates.

QRCodeActivity

The Google Barcode API [52] has several demonstration applications, one of them that implements a set of barcodes scanner. This application was used as a base line, as the required QR code scanning feature is included on its source code (both activity's and library's integration).

The defined activity has defined to have a simple operational behavior: after the viewfinder becomes active, when the user points to a valid QR code, it immediately exits and sends the QR code value back

to the IndicatorFragment.

In sum, these two components define the whole UI part of the QR code scanning process and their integration will be presented on the following sections.

(a) IndicatorFragment **(b)** QRCodeActivity

Figure 5.1: UI elements presented to the visitor for QR code scanning.

5.1.2 Services integration

For simplicity purposes, the new QR code functionality on the middleware side was designed to follow the same node services pattern, *i.e.* use a service and a correspondent receiver when specific nodes are available.

Starting with the service definition, the QRCodeService class was created and associated to POI nodes which have symbolic locations with ”qrcode” technology. Thus, when the current node has one or more child nodes with this location typing, the service will be initialized.

The correspondent receiver, QRCodeServiceReceiver, has very simple structure: it simply captures Intents with a single ”QRCODE” parameter, which will be used inside a mapping function. Similarly to the other mapping functions, its main goal is to find if a child node of the current node matches the scanned QR code. It is important to note that QR code POI nodes are being considered to not have

any children, given their positioning accuracy. It is not a formal definition (neither it is enforced by the designed schema), it is just a practical assumption for this technology.

These two definitions would suffice if the QRCodeService was able to obtain QR codes by itself. However, as stated above, the user must scan a code in order to obtain it. To overcome this, a connection was made between the QRCodeService and the IndicatorFragment introduced in the previous section via broadcast events. The service simply mirrors its state (active/inactive) to the IndicatorFragment by broadcasting two events to the fragment, one for being visible on the application's UI and the other to be hidden. As expected, two additions had to be performed on the IndicatorFragment code:

1. Add a BroadcastReceiver to catch the QRCodeService broadcasts. Two public Intents were also defined for the service correctly identify the fragment action;
2. Add methods to change the indicator's visibility according to the received events from the service.

Recalling the UI operation of the IndicatorFragment and the QRCodeActivity, when a user scans a code, the IndicatorFragment receives it as result from the activity. This result, relevant for the PoiManager is then broadcasted using the QRCodeServiceReceiver's "QRCODE" Intent definition, allowing its reception and evaluation on the middleware. These interactions are illustrated on figure 5.2.

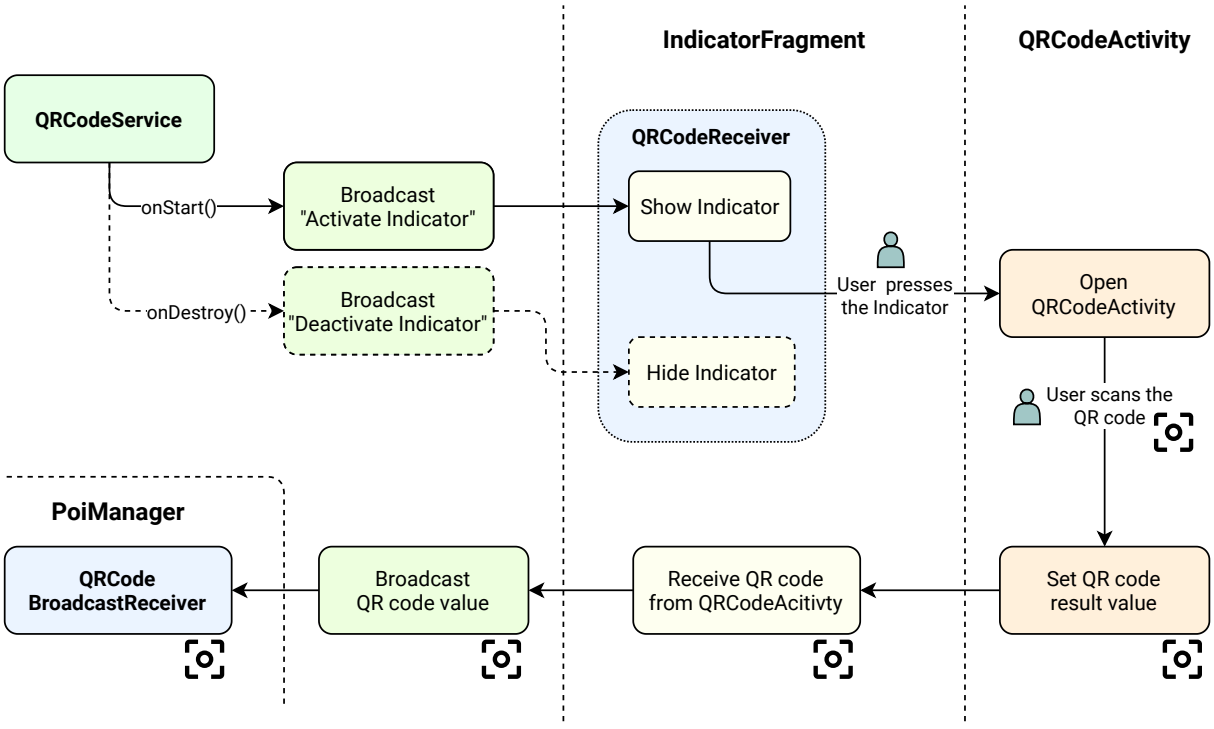


Figure 5.2: QR code scanning process.

Regarding the QR codes' value evaluation for current node updates, a fundamental aspect must be taken into account. As referred on several occasions in this work, the POI's possible location types

were designed to define clear entering and exiting conditions. When a POI identified by a QR code is determined to be the most relevant to the visitor, no exit condition is being defined: after the QR code scan, no extra information can change the current node, effectively locking the visitor to this POI. To conciliate this limitation with the inherent POI update needed on the application side after scanning a QR code, a specific behavior was defined for this technology: after successfully evaluating a QR code node to be the most relevant to the visitor, the ”enteredNode” event is sent but the currentNode reference is not updated. Thus, the current node for the PoiManager is still the QR code's node's parent (which can be easily modified by the other services) and the application is able to display the information related to the QR code's POI.

5.1.3 Application integration

The final step to complete the integration of the QR code service is to integrate the UI elements into any application. Considering the fact that all of the logic is already defined within the UI components, the only necessary additions were:

1. Define a placeholder, more specifically a FrameLayout, on any of the application's UI layouts to define the position of the IndicatorFragment;

2. Instantiate the IndicatorFragment inside the defined FrameLayout.

(a) No QR codes available

(b) QR codes available to the visitor.

Figure 5.3: Automatic display of the QR code indicator according to the visitor's location.

At this stage, the newly defined feature is completely integrated onto an application. Using the initially defined services and library architecture, a UI supported location service could be easily integrated into the application, with minimal modifications. This illustrates that even application layer components can be defined on a library package, which simplifies the development process. Figure 5.3 illustrates the automatic behavior of the service: when the visitor is near an area where QR codes can be scanned, the IndicatorFragment will automatically appear on the UI.

5.2 Application

Considering the nature of the developed solution, a middleware for aiding exhibition navigation applications, the most suitable environment to demonstrate its operation and integration is through a demonstration application. The application will allow the user to configure the middleware operation and exemplify how the middleware can support this type of applications. Some application features, not essential to the solution's normal operation, will require additional API connections to the library. These feature were designed to have a negligible impact on the solution's performance. Figure 5.4 present the application's logic diagram.

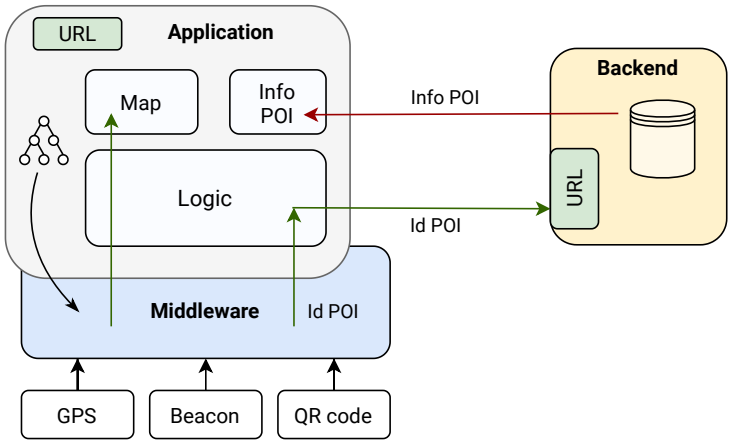


Figure 5.4: Application logic diagram.

The application starts by instantiate the middleware by loading a POI tree contained on a JSON file. The middleware is configured to use three location systems: GPS, BLE and QR codes. These systems will provide location callbacks to the middleware which will determine the most relevant POI and send its identifier to the application. In order to obtain further information about the POIs, the application has an Uniform Resource Locator (URL) reference to a backend to which it will send the POI identifiers. Lastly, after receiving the POI information from the backend, the application will display this information to the visitor.

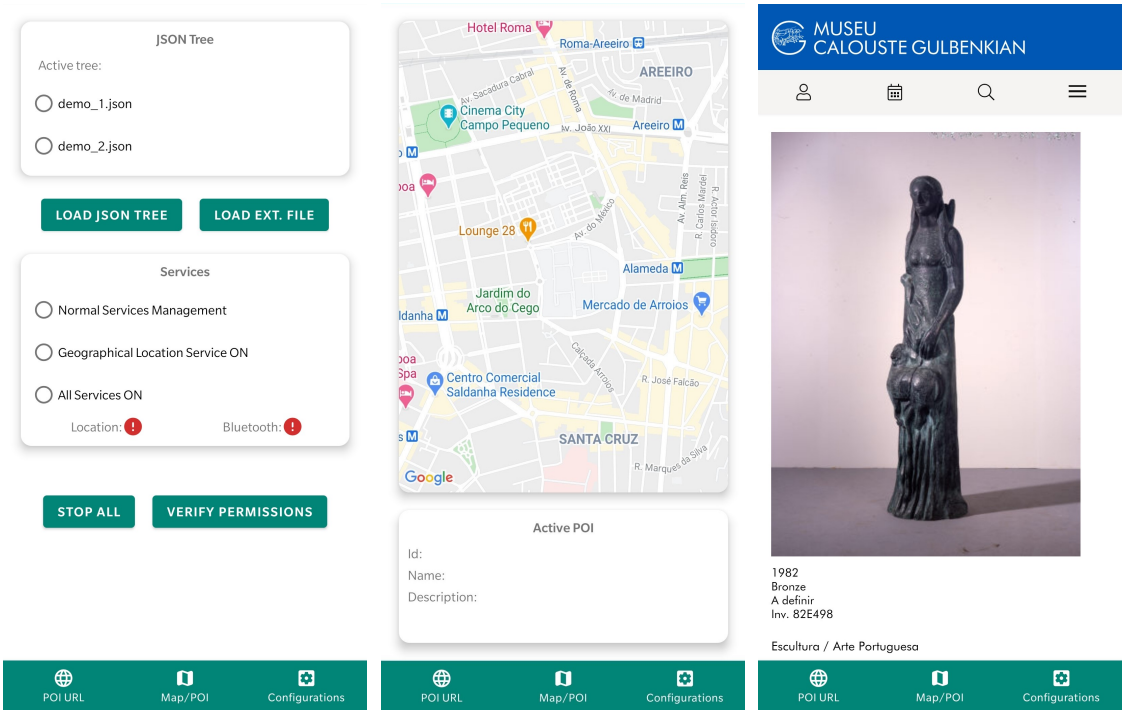
An example of a POI structure that utilizes the three supported location systems by the middleware is

presented on listing 5.1. It is relevant to point out each of the nodes can have additional children nodes with different location descriptions. For additional information about some items, the nodes identified by QR code also have an URL to the Gulbenkian Foundation artwork database [5].

Listing 5.1: Example of JSON file describing a POI tree.

```
"nodes":[
    {
        "id": 1,
        "name": "Pavilhao Central - Entrada/Saida",
        "location":{
            "location_type":"geographical",
            "location_properties":{
                "geographical_location_type":"circle",
                "geographical_location_properties":{
                    "center":{
                        "latitude": 38.736765,
                        "longitude": -9.139073
                    },
                    "radius": 8.21
                }
            }
        },
        "children":[
            {
                "id": 11,
                "name": "Sala 1",
                "location":{
                    "location_type":"symbolic",
                    "location_properties":{
                        "id": "0x80000080001000000aa00",
                        "technology": "beacon",
                        "threshold": 2.00
                    }
                },
                "children":[
                    {
                        "id": 1000,
                        "name": "Anjo - Gustavo Bastos",
                        "location":{
                            "location_type":"symbolic",
                            "location_properties":{
                                "id" : "82E498",
                                "technology": "qrcode",
                                "link": "https://gulbenkian.pt/museu/
                                    works_cam/anjo-156496/"
                            }
                        }
                    }
                ] (...)
            }
        ] (...)
    }
](...)
```

The application UI structure is quite simple, only having a single activity (MainActivity) which host three screens or fragments: one for displaying a map and the current POI (MapPoiFragment), other for configuring the middleware (ConfigsFragment) and one for displaying web pages (WebViewFragment). For user clarity, the application UI was developed by following the Material Design [59] guidelines and integrated some of its icons. Figure 5.5 provides an overview of the available screens.



(a) Configurations screen. **(b)** Map/POI screen. **(c)** WebView screen.

Figure 5.5: Available screens on the demonstration application.

A bidirectional communication was defined between the MainActivity and each of the fragments. As the activity instantiates all three fragments, a reference for each one can be held by the activity. With these fragment references, the activity is able to execute the fragments' public methods and, consequently, send information or execute changes on them. On the other hand, the fragments also define a public listener interface that specify callback methods for the activity. The MainActivity was defined to implement this interfaces, which allowed for the fragments to send information back to the activity. These relationships are illustrated on figure 5.6.

Each of the fragment's features will be described below. It is important to mention that the PoiManager reference is solely hold by the MainActivity. Therefore, some of the fragments' requests and callbacks needed to be routed through the activity fragment references or listeners.

Map/POI screen

As its name suggests, this screen is responsible for displaying a map and which the most relevant POI to the visitor.

The map section of this fragment was built using the Google Maps SDK components for the Android environment [60]. An API key had to be associated to the application package in order to gain access to

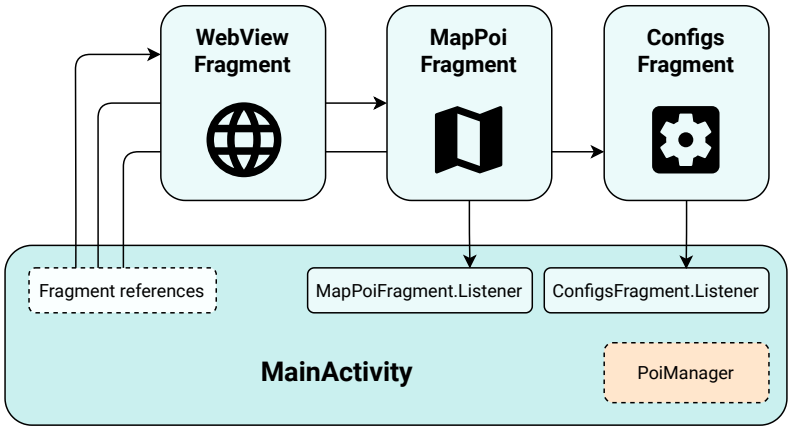


Figure 5.6: Relationships between the MainActivity and its fragments.

the Google Maps API. After this association, a MapView component was simply placed inside the map section layout and the view's initialization code was defined.

The POI section has a set of text boxes that display the current POI's identifier, name and description. It is relevant to point out that the middleware only notifies the application for POI identifiers. Therefore, the POI's name and description must be mapped on the application side (using any kind of data structure) before updating this fragment's text boxes.

Two extra features were added to the map component of this demonstration application:

- **Display geographical areas** - as previously stated, geographical located POIs can defined by geographical circles or polygons. By design, this information is not sent on the middleware's update events.

  To obtain this information, a helper method was added to the middleware's API, requesting a list of all geographical POIs. The middleware then performs a recursive search on the active POI tree and send the requested list via the correspondent helper callback. On the fragment side, the received list is iterated and for each POI a Maps "Circle" or "Polygon" object is created from the area's definition and it is added to map.

- **Display a current geographical location indicator** - the simplest approach to add this feature is to resort to the map's API itself, which as a simple parameter to display the user's current location. However, this method has a major drawback on the current solution context: it uses another instance of the FusedLocationProvider service, which will always be active while running the application. The existence of a permanently running service, external to the middleware's operation, will have a negative impact when evaluating the solution's performance (further detailed on chapter 6)

  To avoid the built-in indicator, a custom indicator was created. As the map component allows the

addition of custom markers, a new marker was added to it. The marker's location is then updated using the middleware's FusedLocationProvider internal service. Once again, as this service is hidden from the application layer, a custom event callback was added to allow discrete location updates on the application. This approach ensures that the indicator's location will only be updated when the middleware's geographical location service is running, which does not affect its normal operation.

It is important to point out that these features were added to facilitate the demonstration of the middleware and were design to have a minimal impact on its operation. The geographical areas and the current location indicator are displayed on figure 5.7

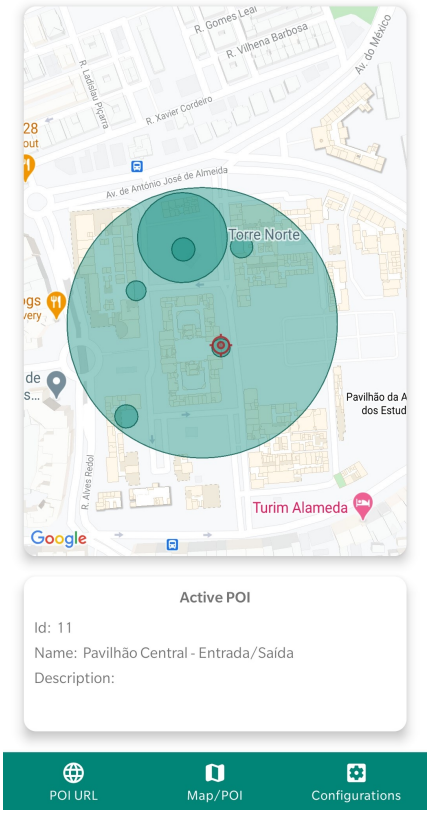


Figure 5.7: Improved Map/POI screen with geographical areas and current location indicator.

Configurations screen

The main purpose of this screen is to configure the middleware's operational parameters. The available options to the user are as follows:

- **Load JSON files** - allows the user to load a JSON file (from a set of pre-compiled application assets) into the middleware. The user can also import external JSON files. This loading action is

used to obtain the necessary JSON content for the initialization method of the middleware;

- **Stop the middleware** - a button is presented to the user to allow for a complete reset of the demonstration application, *i.e.* clears all references to the middle, resets all screens and stops all running services;
- **Configure services' behaviour** - For demonstration purposes, some alternative service management can be selected to ensure some services are kept active, despite the current POI node. Two extra service management were added to the PoiManager, one that keeps the geographical location service always active and other to keep all supported services enabled during the execution. The chosen mode can be selected on a radio button and will be included on the init method call;
- **Recheck necessary permissions** - As described on the Implementation chapter of this work, the permissions request process was integrated into the PoiManager, to simplify the application integration. Nevertheless, the initial permissions' requests can be re-executed by selecting a button on this screen.

WebView screen

The purpose of this screen is to present additional information to the visitor via web pages. This fragment's layout has a single WebView, an Android environment View that renders web content inside of it. Two public methods were defined to allow the MainActivity to execute two actions on this fragment:

1. Load a web page relevant to the user, by providing an URL. The relevant information displayed is related to the current POI: after an event from the middleware containing the POI node's identifier, if the application has an associated URL for this identifier, the URL is sent by the activity to this fragment.
2. Clear the WebView's content, when the displayed information is no longer relevant, generally when the visitor deviates from a POI.

6

Evaluation

Contents

This chapter presents, on section 6.1, how the implemented solution fulfilled the functional requirements defined of section 3.1. On section 6.2, two demonstration exhibition visits were defined in order to evaluate different middleware operations and their energy consumption.

6.1 Functional requirements validation

Considering all of the identified requirements on section 3.1 for exhibition navigation applications, more specifically for an exhibition curator, a developer and a visitor, the following items describe how the developed middleware and JSON schema attempted to fulfill them.

Exhibition curator

FR1 - Define relationships between the constituent areas of an exhibition - This requirement was fulfilled by the proposed JSON schema (section 4.1) as it provides a set of guidelines to define POI tree structures. The exhibition curator can then use these guidelines to hierarchically structure the constituent areas of an exhibition venue;

FR2 - Attribute item positions - In order to implement this requirement, the proposed JSON schema starts by considering a wide definition for POIs, considering both areas and artwork items as constituent elements of exhibition structures. Therefore, items can be easily defined as inner POIs inside a larger POI, as illustrated on the JSON structure example on section 5.2;

FR3 - Assign location identifiers to rooms and their items - This requirement was fulfilled by defining a location property on the core element of the POI structures (the Node object of the proposed JSON schema). The JSON example on section 5.2 presents three examples of different location identifiers: a geographical circle area and symbolic location identifiers using beacons and QR codes;

FR4 - Support a variety of location systems - In order to support several types of location systems, the Location property of the JSON schema was defined to support three location types: geographical, relative and symbolic. Therefore, the exhibition curator can attribute different location types to each POI which will be used by corresponding location systems. On this work, three location systems were integrated: GNSS, BLE and QR code based, described on chapters 4 and 5;

FR5 - Modify a previously defined exhibition structure - In order to make changes to an already defined exhibition structure, it only necessary to modify the corresponding JSON file

and load it into the application. Therefore, the application logic is completely independent of the loaded JSON structures.

Developer

FR6 - Develop applications that allow information access about items and areas - In order to facilitate the development of exhibition supporting applications, a middleware was developed on the Android environment (outlined on section 4.2).

FR7 - Modify the areas and items structure without recompiling the application - The POI trees (defined using the proposed JSON schema) are external to the application and loaded on-demand, *i.e.* are not included on the application code. Therefore, if the POI tree needed to be modified, the application code will remain unchanged and will not require to be compiled again;

FR8 - Easily access to exhibition's identifiers - In order to provide an easy access to the exhibition identifiers by an application, a set of event callbacks were included into the implemented middleware that return POI identifiers;

FR9 - Integrate several location systems - The developed middleware abstracts the whole location services logic from the developer. The location services are defined inside the middleware and solely managed by it. In this work, GNSS, BLE and QR codes were supported;

FR10 - Easily add new location system without rewriting the application - The developed middleware was defined to aggregate all necessary location systems and their services. Therefore, when a location service is added or modified, it is only necessary to compile a new version of the middleware and the developer simply has to include it on the application, leaving the application's code unchanged;

FR11 - Seamlessly management of the necessary location services during a visitation - In order to fulfill this requirement, an active services controller was added to the developed middleware (described on section 4.5). Thus, accordingly to the visitor's location at any given time, the middleware disconnects the location services that serve no purpose for such location, ensuring only the necessary services are kept running;

FR12 - Automatically display widgets - In order to satisfy this requirement, UI components and interconnecting logic was included into the developed middleware (as described on section 4.2.4). These components allow to automatically present buttons on the application's UI which activate specific location services. For instance, when the visitor is on a room with

items identifiable by QR codes, the middleware automatically presents a widget to the user to activate a QR code reader. The developer does not need to program this kind of interactions.

Visitor

FR13 - Automatic notifications for needed user actions - The implemented middleware includes a set of components which enable location services that rely user interactions. For instance, as described on section 5.1, when the visitor is inside a POI with available QR codes, an automatic button is displayed;

FR14 - Seamlessly turns on the necessary location services during a visitation - In order to only activate the necessary services in a seamless manner to the visitor, a controller was added to the implemented middleware (section 4.5). Therefore, the visitor does not need to manually turn on or off each one while visiting an exhibition.

6.2 Performance evaluation

This section describes two performance evaluations made to the middleware. The first one evaluated the energy consumption for different scenarios when simulating two exhibition visits and the other evaluated the size differences resulted by the inclusion of the middleware's code on the demonstration application.

6.2.1 Battery consumption

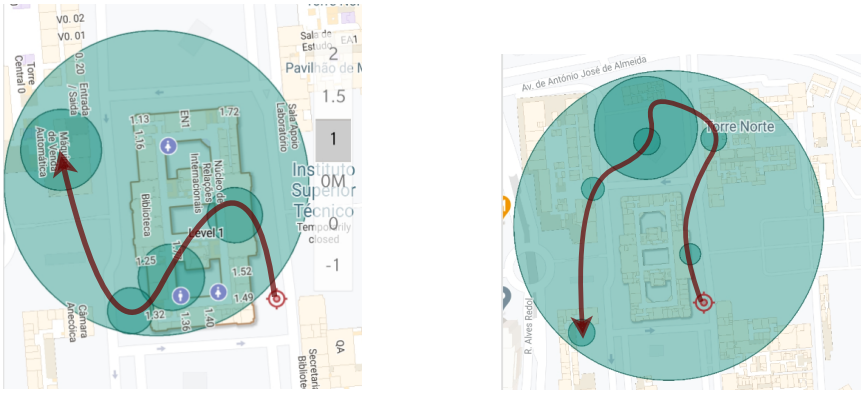
In order to evaluate the middleware's operation and its energy consumption, two demonstration exhibitions were created to mirror real exhibitions and possible paths followed inside them by a visitor. A JSON file was built for each exhibition and both had similar layout: one geographical root node containing the entire exhibition venue, some geographical nodes corresponding to different buildings and indoor rooms identified through BLE beacons. Lastly, some QR nodes were added to each room to simulate displayed artwork. Table 6.1 presents the number of nodes for each location type and the time spent on each node type during the defined visit for each exhibition.

The geographical nodes of both exhibitions were placed inside the Instituto Superior Técnico campus. Figure 6.1 presents the defined path taken while visit each exhibition. It is important to notice that only the geographical POI are presented on the map view of the application.

In order to simulate different BLE beacons, a Pycom [61] LoPy board was used. A Python program was compiled into the board that iterated through a list of beacon identifiers, changing the advertised

Table 6.1: Node typing and corresponding time spent for each demonstration exhibition.

	Demo 1		Demo 2	
Type of node	Number of nodes	Time spent (minutes)	Number of nodes	Time spent (minutes)
Geographical	5	15	7	20
Beacon	20	60	14	42
QR code	53	-	45	-
Total time		75		62



(a) Demo 1 visit path. **(b)** Demo 2 visit path.

Figure 6.1: Paths defined inside the demonstration exhibitions.

value of the simulated beacon. This identifier iteration allowed to simulate a regular exhibition visit through a set of indoor rooms, symbolically identified by BLE beacons.

As stated above, each of the simulated exhibitions' rooms had a set of QR codes as artwork identifiers. For simulating the retrieval of POI information from a web service, the Gulbenkian Foundation [5] artwork database was used, as referred on section 5.2. This database containing a large number of artworks and, most importantly, all have a public identifier. These identifiers were then converted into QR codes and enabled the display of extra information to the visitor, when scanned during the demonstration visits.

After defining the simulated exhibition visits, the next step was to define which were the scenarios for evaluating the middleware's performance. The chosen four scenarios were:

- **Idle** - for providing baseline measurements, the device was kept only with the screen on during the visits' duration;
- **Optimized without QR code** - middleware performs an active services' management, *i.e.* it deactivates unnecessary location services according to the visitor's location, without scanning any QR codes;
- **Optimized with QR code** - middleware performs an active service management and the available

QR codes are scanned;

- **All Services On** - all location services are kept active by the middleware and available QR codes are also scanned.

On all the scenarios, except for the ”Idle” one, the demonstration application was kept open and only the middleware managed the location services. The user followed the predefined path for each demonstration visit and simulated the entering and exiting of rooms by changing the advertised beacon identifier.

The specifications of the device used on all test runs are presented on table 6.2. To ensure equal testing conditions between both visits and all middleware operational modes, all background applications were closed and the screen brightness was kept at a fixed value.

Table 6.2: Mobile device specifications used of the battery consumption tests.

Brand	OnePlus
Model	5T (A5010)
OS (Version)	OxygenOS (9.0.8)
Android version	9.0
Bluetooth version	5.0
Battery capacity	3300 mAh

In order to evaluate the energy consumption in the different test cases, two indicators were used. The first is the battery percentage drop given by the device's operating system and the other is determining the energy consumed by the device. This latter measurement was performed by an external current meter. In order to only account for the energy consumption of the demonstration visits, the following procedure was applied:

1. Charge the device to full capacity;
2. Perform the demo visit;
3. Take note of the battery percentage and shutdown the device;
4. Connect the current meter between the device and the charger;
5. Disconnect the charger (and meter) after reaching full capacity;
6. Take note of the energy consumed value (in mAh) given by the meter.

Both percentage drops and estimated for energy consumption after each test case are presented on table 6.3.

Analyzing the obtained energy consumption measures on table 6.3, the ”Idle” scenario obtained the lowest energy consumption, as expected. On the other hand, the highest energy consumption was obtained on the scenario where all the location services were kept active during the visits. Comparing this

Table 6.3: Measured energy consumption during the demonstration visits.

	Demo 1 (75 min)		Demo 2 (62 min)	
	Estimated consumption (mAh)	Percentage drop	Estimated consumption (mAh)	Percentage drop
Idle	102	5%	98	6%
Optimized w/o QR code	194	11%	162	10%
Optimized with QR code	252	12%	170	10%
All Services On	425	18%	302	13%

scenario with the scenario where the middleware performed an active service management (”Optimized with QR code”), the energy savings were about 40%. Given the majority of the simulated visits duration is spent inside rooms identified by BLE beacons, having the GPS service to be turned off in these areas achieved a significantly lower energy consumption.

Considering the two scenarios when the active service management is performed by the middleware but differ on the scanning of the available QR codes, one can observe the scanning process resulted on a measurable impact on the energy consumption. The increased consumption is justified not only by repeatedly accessing the camera but also by the several network requests to obtain the items’ details.

6.2.2 Application size evaluation

In order to evaluate the impact of the middleware on the size of an application that incorporates it, a comparison was made between the demonstration application (developed on section 5.2) and the same application without importing the middleware’s classes. The Android Studio APK analyzer was used to compare the final sizes of both applications. Table 6.4 contains the obtained size differences.

Table 6.4: Size comparisons between an application with or without including the middleware.

	Without Middleware	With Middleware	Difference
APK total size (compressed)	2.1 MB	2.5 MB	335.4 KB
Application classes size (uncompressed)	1.3 MB	2.0 MB	753.8 KB

Regarding the total size of the APK, where the entirety of the application is compressed on a single file, one can asses that the inclusion of the middleware increased the file size by less than 350 KB. Considering the size of the application’s classes, when including the middleware into the application, the uncompressed size of this classes increased 753.8 KB. Therefore, the inclusion of the middleware did not have a significant impact on both the classes’ and the APK’s sizes, which suggests the majority of the application classes correspond to core libraries of the Android environment.

7

Conclusion

Contents

This final chapter summarizes the main accomplishments of this work and proposes some future improvements for the solution.

7.1 Accomplishments

This work starts by consider a wider definition for the POIs and how hierarchical relationships can be established between them. In other words, not only relevant areas can be treated as POIs but also smaller areas and relevant items contained inside them as well. In order to properly describe the arrangement of a wide diversity of POIs, a JSON schema was developed. This schema enforces a set of guidelines for defining these structures while supporting several location description techniques for each element.

The defined POI structures were defined to be external to any exhibition navigation application. In order to access an exhibition layout, this type of applications simply need to load the corresponding POI structures at run-time. Thus, if future modifications are made on the physical layout of the exhibition, the POI can be updated independently of the application and, consequently, it is not necessary to recompile it.

The other main contribution of this work is a middleware that simplifies the development process of exhibition navigation applications. By receiving a compliant structure to the JSON schema defined above and aggregating the location callback of several location services, the middleware is able to determine which is the most relevant POI at any given time for an exhibition visitor.

One of the main abstractions provided by the middleware is the integration of several location services. Therefore, during the development process of applications that utilize this middleware, the developer does not need to integrate and configure any location service. Furthermore, the middleware also is able to perform an automatic management of the running services. By analysing the visitor's location relative to an exhibition's layout (using its POI representation), the middleware only activates the necessary location services for that location. This management not only further decouples the application developer from the services by also allows for significant energy savings while using exhibition navigation applications. Considering the scenario of an exhibition application that does not have any kind of active service management, the inclusion of this middleware can obtain energy savings of about 40%, without requiring any extra effort to the application developer.

Lastly, the middleware also includes UI components that enable user interaction dependent location services, such as NFC tags or QR codes scanning. Once again, by defining the necessary components directly on the middleware, the entire services' operation is hidden from the application developer. In order to integrate this type of services, the application developer simply has to import the defined UI components into the application's UI.

7.2 Future Work

Despite the accomplishment of the identified requirements for this work, some future improvements are proposed below:

1. **Support more complex relationships between POIs** - Despite providing a good starting point for defining relationships between different POIs, the tree-like POI structures only allow direct hierarchical relationships, *i.e.* "this POI is contained inside another". However, a very common layout for indoor exhibitions is having a set of interconnected rooms which, on the proposed structures, would solely correspond to a POI having a set of child POIs (not defining direct relationships between them). Therefore, the support of graph or mesh POI structures could better describe some physical exhibition layouts at the same it enables better search algorithms: recalling the multiple BLE beacon identified rooms, instead of directing the search of neighbouring nodes to their parent, the search algorithm could prioritize the evaluation of the connected rooms on the POI representation;

2. **Variable service parameters** - The external service classes defined on the middleware provided the fundamental location callbacks to determine the visitor's location and which is the most relevant POI according to it. However, both the Fused Location Provider and the BLE services were configured using static parameters, empirically determined and adjusted. Given their ability to be dynamically configured each time they are executed (or even while running if the "binding service" approach is taken), the services' operation could be further optimized according the visitor's surroundings;

3. **Integrate new location services** - The proposed middleware laid the foundations for easier integration of new location services. Resorting to the implemented components of the middle and following the same implementation process as the integration of the QR code service, a NFC identifier based location system could be easily integrated into the solution (both identify items, have similar operational ranges and similar user interactions to operate properly).

Bibliography

[1] S. Medic and N. Pavlovic, "Mobile technologies in museum exhibitions," *Turizam*, vol. 18, no. 4, pp. 166–174, 2014.

[2] K. Best, "Making museum tours better: Understanding what a guided tour really is and what a tour guide really does," *Museum Management and Curatorship*, vol. 27, no. 1, pp. 35–52, feb 2012.

[3] D. Munjeri, "Tangible and intangible heritage: From difference to convergence," in *Museum International*, vol. 56, no. 1-2, 2004, pp. 12–20.

[4] B. Lord and M. Piacente, *Manual of Museum exhibitions*. Rowman & Littlefield Publishers, 2014.

[5] Fundação Calouste Gulbenkian, "Fundação Calouste Gulbenkian," 2020. [Online]. Available: https://gulbenkian.pt/ Accessed on 2020-08-07.

[6] Fundação de Serralves, "Fundação de Serralves," 2020. [Online]. Available: https://www.serralves.pt/ Accessed on 2020-08-07.

[7] Parques de Sintra – Monte da Lua, "Parques de Sintra," 2020. [Online]. Available: https://www.parquesdesintra.pt/

[8] UNESCO, "List of UNESCO Global Geoparks (UGGp)," 2020. [Online]. Available: http://www.unesco.org/new/en/natural-sciences/environment/earth-sciences/unesco-global-geoparks/list-of-unesco-global-geoparks/?fbclid=IwAR3Ch0SxN9Irj_vYb2-g5cRn0ocl0xDWf2E6a90mdTZ3jy2sV3P7PlT5O-4 Accessed on 2020-08-07.

[9] X. Wei and Z. Jianping, "Mobile Application Used in Museum Learning and Its Case Study," in *Proceedings - 2015 International Conference of Educational Innovation Through Technology, EITT 2015*. Institute of Electrical and Electronics Engineers Inc., apr 2016, pp. 90–93.

[10] H. Tsai and K. Sung, "Mobile applications and museum visitation," *Computer*, vol. 45, no. 4, pp. 95–98, apr 2012. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6178133

[11] R. Wojciechowski, K. Walczak, M. White, and W. Cellary, “Building Virtual and Augmented Reality Museum Exhibitions,” in *Proceedings of the ninth international conference on 3D Web technology - Web3D ’04*. New York, New York, USA: ACM Press, 2004.

[12] M. A. Nassar and F. Meawad, “An augmented reality exhibition guide for the iphone,” in *Proceedings - 2010 International Conference on User Science and Engineering, i-USEr 2010*, 2010, pp. 157–162. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5716742

[13] N. Desai, “Recreation of history using augmented reality,” *ACCENTS Transactions on Image Processing and Computer Vision*, vol. 4, no. 10, pp. 1–5, 2018.

[14] Smithsonian National Museum of Natural History DC, “Bone Hall,” 2015. [Online]. Available: https://naturalhistory.si.edu/exhibits/bone-hall Accessed on 2020-08-09.

[15] V. Cesário, “Guidelines for combining storytelling and gamification: Which features would teenagers desire to have a more enjoyable museum experience?” in *Conference on Human Factors in Computing Systems - Proceedings*, 2019.

[16] F. Borrero, P. C. Sanjuán Muñoz, and G. Ramírez González, “Técnicas de gamificación en el turismo, prueba de aplicación, Casa Museo Mosquera,” *Sistemas y Telemática*, vol. 13, no. 33, pp. 63–76, 2015.

[17] A. Küpper, *Location-Based Services: Fundamentals and Operation*. John Wiley and Sons, dec 2005.

[18] National Coordination Office for Space-Based Positioning Navigation and Timing, “GPS.gov: Space Segment,” 2020. [Online]. Available: https://www.gps.gov/systems/gps/space/ Accessed on 2020-07-31.

[19] IAC, “Glonass history,” 2018. [Online]. Available: https://glonass-iac.ru/en/guide/index.phphttps://www.glonass-iac.ru/en/guide/index.php Accessed on 2020-07-31.

[20] ESA, “ESA - What is Galileo?” 2019. [Online]. Available: https://www.esa.int/Applications/Navigation/Galileo/What_is_Galileo

[21] European GNSS Agency, “System — European GNSS Service Centre,” 2020. [Online]. Available: https://www.gsc-europa.eu/galileo/system Accessed on 2020-07-31.

[22] Reuters, “China puts final satellite into orbit to try to rival GPS network - Reuters,” 2020. [Online]. Available: https://uk.reuters.com/article/uk-space-exploration-china-satellite/china-puts-final-satellite-into-orbit-to-try-to-rival-gps-network-idUKKBN23U0BP Accessed on 2020-07-31.

[23] X. Li, X. Zhang, X. Ren, M. Fritsche, J. Wickert, and H. Schuh, “Precise positioning with current multi-constellation Global Navigation Satellite Systems: GPS, GLONASS, Galileo and BeiDou,” *Scientific Reports*, vol. 5, no. 1, p. 8328, feb 2015.

[24] H. Laitinen, S. Ahonen, S. Kyriazakos, J. Lähteenmäki, R. Menolascino, and S. Parkkila, “Project Number: IST-2000-25382-CELLO Project Title: Cellular network optimisation based on mobile location Cellular Location Technology,” 2001.

[25] I. K. Adusei, K. Kyamakya, and K. Jobmann, “Mobile positioning technologies in cellular networks: An evaluation of their performance metrics,” in *Proceedings - IEEE Military Communications Conference MILCOM*, vol. 2, 2002, pp. 1239–1244.

[26] R. S. Campos, “Evolution of positioning techniques in cellular networks, from 2G to 4G,” 2017.

[27] Wigle.net, “WiGLE: Wireless Network Mapping,” 2020. [Online]. Available: https://www.wigle.net/ https://wigle.net/ Accessed on 2020-08-07.

[28] A. Khalajmehrabadi, N. Gatsis, and D. Akopian, “Modern WLAN Fingerprinting Indoor Positioning Methods and Deployment Challenges,” pp. 1974–2002, jul 2017.

[29] V. Honkavirta, T. Perälä, S. Ali-Löytty, and R. Piché, “A comparative survey of WLAN location fingerprinting methods,” in *Proceedings - 6th Workshop on Positioning, Navigation and Communication, WPNC 2009*, 2009, pp. 243–251.

[30] Bluetooth SIG, “Bluetooth Core Specification versiom 5.2,” 2019. [Online]. Available: www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc{_}id=478726 Accessed on 2020-08-09.

[31] R. Faragher and R. Harle, “Location fingerprinting with bluetooth low energy beacons,” *IEEE Journal on Selected Areas in Communications*, vol. 33, no. 11, pp. 2418–2428, nov 2015.

[32] Y. C. Pu and P. C. You, “Indoor positioning system based on BLE location fingerprinting with classification approach,” *Applied Mathematical Modelling*, vol. 62, pp. 654–663, oct 2018.

[33] Near Field Communication Forum, “About the Technology — NFC Forum,” pp. 1–7, 2019. [Online]. Available: https://nfc-forum.org/what-is-nfc/about-the-technology/ Accessed on 2020-07-30.

[34] B. Ozdenizci, V. Coskun, and K. Ok, “NFC internal: An indoor navigation system,” *Sensors (Switzerland)*, vol. 15, no. 4, pp. 7571–7595, mar 2015.

[35] Denso Wave Incorporated, “History of QR Code,” 2020. [Online]. Available: https://www.qrcode.com/en/history/ Accessed on 2020-07-26.

[36] W3C, “Points of Interest,” 2012. [Online]. Available: https://www.w3.org/2010/POI/wiki/Main{_}Page Accessed on 2020-08-11.

[37] ——, “Points of Interest Core,” 2012. [Online]. Available: https://www.w3.org/2010/POI/documents/Core/core-20111216.html{#}location Accessed on 2020-08-12.

[38] Fiware, “PointOfInterest - Fiware-DataModels,” 2020. [Online]. Available: https://fiware-datamodels.readthedocs.io/en/latest/PointOfInterest/PointOfInterest/doc/spec/index.html Accessed on 2020-08-12.

[39] I. E. T. F. ITEF, “RFC 7946 - The GeoJSON Format,” p. 28, 2016. [Online]. Available: https://tools.ietf.org/html/rfc7946http://www.rfc-editor.org/info/rfc7946 Accessed on 2020-08-12.

[40] R. L. Pereira, P. C. Sousa, R. Barata, A. Oliveira, and G. Monsieur, “CitySDK Tourism API - building value around open data,” *Journal of Internet Services and Applications*, vol. 6, no. 1, pp. 1–13, 2015.

[41] S. H. Hashemi and J. Kamps, “Exploiting behavioral user models for point of interest recommendation in smart museums,” *New Review of Hypermedia and Multimedia*, vol. 24, no. 3, pp. 228–261, jul 2018.

[42] M. Nitti, V. Pilloni, D. Giusto, and V. Popescu, “IoT Architecture for a sustainable tourism application in a smart city environment,” *Mobile Information Systems*, vol. 2017, 2017.

[43] International Data Corporation, “IDC - Smartphone Market Share - OS,” p. 1, 2019. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os Accessed on 2020-08-19.

[44] Android Developers, “android.location,” 2020. [Online]. Available: https://developer.android.com/reference/android/location/package-summary Accessed on 2020-08-18.

[45] Google, “Google Location Services API,” 2020. [Online]. Available: https://developers.google.com/android/reference/com/google/android/gms/location/package-summary Accessed on 2020-08-18.

[46] Android Developers, “android.bluetooth,” 2019. [Online]. Available: https://developer.android.com/reference/android/bluetooth/package-summary Accessed on 2020-08-16.

[47] Radius Network, “Android Beacon Library,” 2016. [Online]. Available: https://altbeacon.github.io/android-beacon-library/ Accessed on 2020-08-16.

[48] Meridian Apps, “Meridian — Indoor Navigation SDK, Push Notification SDK,” 2019. [Online]. Available: https://meridianapps.com/sdks/?{_}ga=2.253281353.1565558491.1547487027-457411055.1527783307 Accessed on 2020-09-129.

[49] Kontakt.io, “Kontakt.io Developer Center,” 2019. [Online]. Available: https://developer.kontakt.io/ Accessed on 2020-09-12.

[50] N. M. R. Sousa, “UBILocus - Framework for location aware application development,” Ph.D. dissertation, Instituto Superior Ténico, 2017.

[51] Zxing, “GitHub - zxing/zxing: ZXing (”Zebra Crossing”) barcode scanning library for Java, Android,” 2020. [Online]. Available: https://github.com/zxing/zxing/https://github.com/zxing/zxing Accessed on 2020-08-18.

[52] Google Developers, “Barcode API Overview — Mobile Vision,” 2020. [Online]. Available: https://developers.google.com/vision/android/barcodes-overview Accessed on 2020-08-18.

[53] SQLite.org, “SQLite Home Page,” p. One, 2014. [Online]. Available: https://sqlite.org/index.html Accessed on 2020-08-19.

[54] Android Developers, “Save data in a local database using Room,” 2018. [Online]. Available: https://developer.android.com/training/data-storage/room Accessed on 2020-08-19.

[55] The Mad Pixel Factory, “Second Canvas — Home,” 2019. [Online]. Available: https://www.secondcanvas.net Accessed on 2020-09-12.

[56] Museu Nacional de Arte Antiga, “Museu Nacional de Arte Antiga — Página inicial,” 2020. [Online]. Available: http://museudearteantiga.pt/ Accessed on 2020-09-12.

[57] J. schema org, “JSON Schema — The home of JSON Schema,” 2020. [Online]. Available: https://json-schema.org/http://json-schema.org/

[58] Newtonsoft, “JSON Schema Validator,” 2020. [Online]. Available: https://www.jsonschemavalidator.net Accessed on 2020-08-25.

[59] Google, “Design - Material Design,” 2020. [Online]. Available: https://material.io/design Accessed on 2020-09-06.

[60] ——, “Overview — Maps SDK for Android — Google Developers,” 2019. [Online]. Available: https://developers.google.com/maps/documentation/android-sdk/overview Accessed on 2020-09-06.

[61] Pycom, “Pycom - Next Generation Internet of Things Platform,” 2019. [Online]. Available: https://pycom.io/ Accessed on 2020-09-11.

A

JSON Schema

```
{
  "$id":"https://example.com/poi-tree.schema.json",
  "$schema":"http://json-schema.org/draft-07/schema#",
  "title":"POI Tree Schema",
  "description":"Validation schema for a tree structure of POIs (nodes)",
  "type":"object",
  "properties":{
    "nodes":{
      "type":"array",
      "items":{
        "$ref":"#/definitions/node"
      },
      "uniqueItems":true
    }
  },
  "required":[
    "nodes"
  ],
  "definitions":{
    "node":{
      "type":"object",
      "properties":{
        "id":{
          "type":"number"
        },
        "name":{
          "type":"string"
        },
        "location":{
          "$ref":"#/definitions/location"
```

```
        },
        "children":{
          "type":"array",
          "items":{
            "$ref":"#/definitions/node"
          }
        }
      },
      "required":[
        "id",
        "location"
      ],
      "additionalProperties":false
    },
    "location":{
      "type":"object",
      "properties":{
        "location_type":{
          "enum":[
            "geographical",
            "relative",
            "symbolic"
          ]
        },
        "location_properties":{

        }
      },
      "allOf":[
        {
          "if":{
            "properties":{
              "location_type":{
                "const":"geographical"
              }
            }
          },
          "then":{
            "properties":{
              "location_properties":{
                "$ref":"#/definitions/geographical_location"
              }
            }
          }
        },
        {
          "if":{
            "properties":{
              "location_type":{
                "const":"relative"
              }
            }
          },
          "then":{
            "properties":{
              "location_properties":{
                "$ref":"#/definitions/relative_location"
              }
            }
          }
        },
        {
          "if":{
            "properties":{
              "location_type":{
                "const":"symbolic"
              }
            }
          },
          "then":{
            "properties":{
```

```
                "location_properties":{
                  "$ref":"#/definitions/symbolic_location"
                }
              }
            }
          }
        ],
        "required":[
          "location_type",
          "location_properties"
        ],
        "additionalProperties":false
      },
      "geographical_location":{
        "type":"object",
        "properties":{
          "geographical_location_type":{
            "enum":[
              "circle",
              "polygon"
            ]
          },
          "geographical_location_properties":{

          }
        },
        "allOf":[
          {
            "if":{
              "properties":{
                "geographical_location_type":{
                  "const":"circle"
                }
              }
            },
            "then":{
              "properties":{
                "geographical_location_properties":{
                  "$ref":"#/definitions/geographical_circle"
                }
              }
            }
          },
          {
            "if":{
              "properties":{
                "geographical_location_type":{
                  "const":"polygon"
                }
              }
            },
            "then":{
              "properties":{
                "geographical_location_properties":{
                  "$ref":"#/definitions/geographical_polygon"
                }
              }
            }
          }
        ],
        "required":[
          "geographical_location_type",
          "geographical_location_properties"
        ],
        "additionalProperties":false
      },
      "geographical_circle":{
        "type":"object",
        "properties":{
          "center":{
            "$ref":"#/definitions/geographical_coordinates"
```

```
        },
        "radius":{
          "type":"number"
        }
      },
      "required":[
        "center"
      ],
      "additionalProperties":false
    },
    "geographical_polygon":{
      "type":"object",
      "properties":{
        "points":{
          "type":"array",
          "items":{
            "$ref":"#/definitions/geographical_coordinates"
          },
          "minItems":3,
          "uniqueItems":true
        }
      },
      "required":[
        "points"
      ],
      "additionalProperties":false
    },
    "relative_location":{
      "type":"object",
      "properties":{
        "relative_location_type":{
          "enum":[
            "circle",
            "polygon"
          ]
        },
        "relative_location_properties":{

        },
        "map_id":{
          "type":"number"
        }
      },
      "allOf":[
        {
          "if":{
            "properties":{
              "relative_location_type":{
                "const":"circle"
              }
            }
          },
          "then":{
            "properties":{
              "relative_location_properties":{
                "$ref":"#/definitions/relative_circle"
              }
            }
          }
        },
        {
          "if":{
            "properties":{
              "relative_location_type":{
                "const":"polygon"
              }
            }
          },
          "then":{
            "properties":{
              "relative_location_properties":{
```

```
                "$ref":"#/definitions/relative_polygon"
              }
            }
          }
        }
      ],
      "required":[
        "relative_location_type",
        "relative_location_properties",
        "map_id"
      ],
      "additionalProperties":false
    },
    "relative_circle":{
      "type":"object",
      "properties":{
        "center":{
          "$ref":"#/definitions/cartesian_coordinates"
        },
        "radius":{
          "type":"number"
        }
      },
      "required":[
        "center"
      ],
      "additionalProperties":false
    },
    "relative_polygon":{
      "type":"object",
      "properties":{
        "points":{
          "type":"array",
          "items":{
            "$ref":"#/definitions/cartesian_coordinates"
          },
          "minItems":3,
          "uniqueItems":true
        }
      },
      "required":[
        "points"
      ],
      "additionalProperties":false
    },
    "symbolic_location":{
      "type":"object",
      "properties":{
        "id":{
          "type":"string"
        },
        "technology":{
          "enum":[
            "beacon",
            "qrcode",
            "rfid"
          ]
        },
        "threshold":{
          "type":"number"
        }
      },
      "required":[
        "id"
      ],
      "additionalProperties":true
    },
    "geographical_coordinates":{
      "type":"object",
      "properties":{
        "latitude":{
```

```
            "$ref":"#/definitions/geographical_coordinate"
          },
          "longitude":{
            "$ref":"#/definitions/geographical_coordinate"
          }
        },
        "required": [
          "latitude",
          "longitude"
        ]
      },
      "geographical_coordinate":{
        "type":"number",
        "minimum":-180,
        "maximum":180
      },
      "cartesian_coordinates":{
        "type":"object",
        "properties":{
          "x":{
            "type":"number"
          },
          "y":{
            "type":"number"
          }
        },
        "required": [
          "x",
          "y"
        ],
        "additionalProperties": false
      }
    }
}
```