D5.2 Symbiosis of smart objects across IoT environments

688156 - symbIoTe - H2020-ICT-2015

Report on System Integration and Application Implementation

The symbIoTe Consortium

Intracom SA Telecom Solutions, ICOM, Greece
Sveučiliste u Zagrebu Fakultet elektrotehnike i računarstva, UNIZG-FER, Croatia
AIT Austrian Institute of Technology GmbH, AIT, Austria
Nextworks Srl, NXW, Italy
Consorzio Nazionale Interuniversitario per le Telecomunicazioni, CNIT, Italy
ATOS Spain SA, ATOS, Spain
University of Vienna, Faculty of Computer Science, UNIVIE, Austria
Unidata S.p.A., UNIDATA, Italy
Sensing & Control System S.L., S&C, Spain
Fraunhofer IOSB, IOSB, Germany
Ubiwhere, Lda, UW, Portugal
VIPnet, d.o.o, VIP, Croatia
Instytut Chemii Bioorganicznej Polskiej Akademii Nauk, PSNC, Poland
NA.VI.GO. SCARL, NAVIGO, Italy

For more information on this document or the symbIoTe project, please contact:
Sergios Soursos, INTRACOM TELECOM, souse@intracom-telecom.com

Document Control

Number: D5.2

Title: Report on System Integration and Application Implementation

Type: Public

Editor(s): Gerhard Dünnebeil, Refiz Duro, AIT
E-mail: gerhard.duennebeil@ait.ac.at, refiz.duro@ait.ac.at

Author(s): Mario Drobics, Karl Kreiner, Ivan Gojmerac, AIT; Joao Garcia, UW, Raquel Ventura Miravet, S&C; Matteo Pardi, NXW; Reinhard Herzog, IOSB; Vasilis Glykantzis, ICOM; Juan Belmonte Rodriguez, Antonio Paradell Bondia, WLI; Luca De Santis, NAVIGO; Szymon Mueller, PSNC. Aleksandar Antonić, UNIZG-FER.

Doc ID: D5.2-v20

Amendment History

Version	Date	Author	Description/Comments
V0.1	March 23rd , 2017	R. Duro	Initial ToC
V0.2	May 2nd, 2017	G.Dünnebeil	Structure for the use case descriptions
V0.3	June 7th, 2017	Matteo Pardi, Raquel Ventura Miravet, Reinhard Herzog, G. Dünnebeil	Merged first round of contributions
V0.4	June 7th, 2017	Reinhard Herzog	Fixed the mess left over from merging chaos
V0.5	June 13th, 2017	Refiz Duro	Updates to Chapter 2 (Introduction), addressed formatting issues
V0.6	June 20th, 2017	Vasilis Glykantzis, Juan Belmonte, Antonio Paradell Bondia, Refiz Duro	Integration of symbIoTe core descriptions (Chapter 3), and Smart Stadium application descriptions
V0.7	June 20th, 2017	Refiz Duro	Edits, structural updates
V0.8	June 22nd, 2017	Matteo Pardi	Input from NXW integrated – Chapter 4 (Applications: Smart Area Control and Home Comfort)
V0.9	June 29th, 2017	Karl Kreiner, Vasilis Glykantzis, Reinhard Herzog, Luca De Santis	Added input from AIT on Smart Home Health use case (only added structure) Updated description of development process Description of EduCampus Description of Use Case Smart Yachting
V0.10	July, 11th, 2017	Gerhard Dünnebeil, Juan Belmonte Rodriguez, Szymon Mueller	Editorial work, fixed the sequence diagrams for Smart Stadium UC, Added the chapter about stress testing
V0.10	July, 13th, 2017	Joao Garcia	Description of SMEUR use case
V0.10	July, 14th, 2017	Aleksandar Antonic	Code examples for cloud services
V0.11	July, 18th, 2017	Luca De Santis, Refiz Duro	Update to Use Case Scenario “Smart Yachting” (Section 7.5). Edits before the internal interview (including moving text to Appendix, writing Executive Summary, Conclusion).
V0.12	July 24th-July 27th, 2017	Corinna Schmitt, Refiz Duro	Internal Review by Corinna, edits by Refiz
V0.13	July 29th - August 03, 2017	Marcin Plociennik, Zvonimir Zelenika, Refiz Duro, Mario Drobics	Internal review by Marcin, Zvonimir. Followed by significant edits by Refiz, Mario (doc harmonization, restructuring, text updates, references, new figures, etc.)
V0.14	July 09th, 2017	Joao Garcia, Raquel Ventura Miravet, Matteo Pardi, Luca de Santis, Reinhard	Review-based input on Smart Mobility, Smart Home, Smart Yachting and EduCampus integrated.

		Herzog, Karl Kreiner	
V0.15	July 11th, 2017	Refiz Duro	Edits to Chapter 5, tables, harmonization of document, and identification of the still-missing pieces. Update of Executive Summary.
V0.16	July14th, 2017	Reinhard Herzog, Luca de Santis, Matteo Pardi, Gerhard Dünnebeil, Marcin Pliciennik, Raquel Ventura Miravet	Integration of input due to missing parts.
V0.17	Aug.22nd	Tilemachos Pechlivanoglou, Karl Kreiner, Joao Garcia	Last minute integration of inputs, preparing second internal review
V0.18	Aug 24th	Luca De Santis, Alexandar Antonić, Juan Belmonte	More last minute input. Brushed up to be the final version for the second internal review
V0.19	Aug. 28th	Gerhard Dünnebeil	Latest change request from the second review.
V0.20	Aug. 31st	Sergios Soursos	Editorial changes, submission-ready version

(This page is left blank intentionally.)

Table of Contents

1 Executive Summary

With the purpose of fostering the development of an open IoT ecosystem and market, the H2020 project symbIoTe will provide an abstraction layer for various existing IoT platforms as its primary objective towards IoT platform interoperability. Furthermore, symbIoTe will also pursue the challenging tasks of implementing IoT platform federation, enabling the platforms to interoperate, collaborate, share resources for the mutual benefit, and to support the migration of smart objects between various IoT domains and platforms.

Among the main activities in Work Package 5 (WP5), entitled "Use-case based Trials and Deployments", are the activities directly related to design and implementation of applications based on the selected symbIoTe use cases that utilize the domain-specific symbIoTe APIs specified in Work Package 1. In this deliverable, D5.2, the focus is in particular set on the system integration (symbIoTe ecosystem), as well as on the involved applications aimed at the end users that need to be integrated in the symbIoTe ecosystem.

Since providing IoT services is an extremely distributed and performance oriented procedure, we design a system matching these needs. We decided to follow the microservices architecture, which aims to provide better scalability, performance and code maintenance. Information regarding the system integration, such as the project structure, the description of the source tree, dependencies, code coverage, as well as feature planning information is provided. The design is flexible enough to allow for language independency when, e.g., developing or using platform-specific plugins for an IoT platform integration.

Further, a list of performance requirements for the symbIoTe system was defined in Deliverable D1.2. To ensure that provided services/applications fulfill those requirements, a stress testing testbed together with the test scenarios is designed and created. Testing including varied number of users and concurrent requests to the system showed that symbIoTe release R2 is relatively stable and can handle pretty large amount of request for medium sized datasets. Testing also showed that we require introducing specific approaches to handle larger datasets and more concurrent users using the system.

The second part considers applications to run on top of the symbIoTe prototype. The main purpose of such applications is to simplify and optimize end users' daily activities in various situations and environments. Five use cases of symbIoTe cover different domains, and make use of different applications:

1. Smart Residence: *Smart Healthy Indoor Air, Smart Area Controller, Home Comfort and Smart Health Mirror applications.*
2. Smart Mobility and Ecological Routing: *Mobile and Web applications offering optimized routing alternatives and point of interest search.*
3. Edu Campus: *Searching for a Room application*.
4. Smart Stadium: *Visitor, Retailer and Promowall applications*.
5. Smart Yachting. *Portnet and Centrale Acquisti applications*.

The listed applications allow for optimal collaboration and cooperation on top of the available resources, e.g., sensors, actuators, mobile devices, processing resources, etc. Development of the non-existing, or adjustment of already existing applications in the

given list to provide services through the symbIoTe ecosystem serve as excellent examples and/or tests of the working interoperable framework across existing platforms.

In order to make this possible, applications as well as the IoT platforms, need to be compliant. In particular, symbIoTe has four different interoperability modes, which are referred to as “Compliance Levels” (L1 to L4) that aim to enable an incremental deployment of functionalities across the architectural domains: 1) Application Domain, 2) Cloud Domain, 3) Smart Space Domain, and 4) Device Domain. This allows for flexibility in choosing the collaboration level with other platforms within the symbIoTe-enabled ecosystem. Different components are accordingly required for different domains and desirable compliance level(s).

Most of the symbIoTe use cases target L1 and L2 compliance levels, with extensions to L3 and L4 in Smart Residence and Smart Yachting. Hence, the IoT platforms need to integrate the required symbIoTe components according to these compliance levels, while the aforementioned applications must be designed (or adjusted/extended) and developed accordingly. We provide an overview of the current status of the platform integration, their levels of compliance, as well as on the development and integration status of the end user applications.

This document primarily reports on the work done in Task T5.1 and on the work done and planned in Task 5.2 which will end in Month 26. The work on the integration of the symbIoTe prototype and the developed and implemented applications running on top of it will continue until month 30 in Task T5.4, with the final results presented in Deliverable D5.4.

2 Introduction

This section includes a brief introduction to symbIoTe, identifies the purpose of the document, and maps the included results to the objectives mentioned in the description of work (DoW). Finally, the document structure is explained.

2.1 symbIoTe

In a world of smart networked devices and wearables as well as sensors and actuators, which blend with the surrounding environment to provide daily life services, transparent and secure access to and usage of the available resources across various IoT domains is crucial to satisfy the needs of an increasingly connected society. Users are in demand of novel applications that simplify their daily activities in various situations and environments. Some examples of such situations and environments occur at home or at the office, when commuting, or at airports/train stations, and during leisure activities such as visiting stadiums or shopping malls. Following such demands, new requirements have emerged due to the growing number of IoT users worldwide, lower entry barriers for non-technical users to become content and service providers, and the available IoT platforms and services on the market.

The current situation, however, is that of fragmented IoT ecosystems. This is best depicted by a series of vertical solutions, which on the one hand integrate connected objects within local environments (e.g., home, office, etc.) that we call smart spaces, and on the other hand connect smart spaces with back-end cloud hosting software components, which are often proprietary. The vertical solution implies restrictions to the ecosystem that is developed around a single platform, thereby limiting access to all other IoT ecosystems. Interoperability and IoT federations are thus needed to achieve collaboration and access to services and resources provided by the different IoT platforms.

Figure 1 shows an example of IoT ecosystems powered by three different platforms: Platform 1 focuses on integrating Smart Home environments; Platform 2 is tailored to the needs of office and Smart Campus environments, while Platform 3 focuses on providing solutions for public spaces. There are numerous commercial offerings in the form of services and applications in these domains on the market. Infrastructure providers are at the beginning of the value chain by setting up devices and gateways in smart spaces, IoT platform developers maintain and sell the platforms, cloud and IoT service providers host the platforms, while application developers/providers build innovative web and mobile applications on top the platforms and infrastructure. End users interact either directly with infrastructure providers and use the provided applications for their infrastructure, or with IoT service providers who offer bundled service. Telecom operators are in the pole position to expand their service portfolio with IoT services and to act as infrastructure providers by expanding their existing infrastructure with IoT resources. It is clear that application developers and providers are locked in with a platform and need to adjust their solutions for each new platform and underlying infrastructure, while infrastructure providers cannot offer their resources to multiple IoT service providers.

symbIoTe comes to remedy this fragmented environment by providing an abstraction layer for a “unified view” on various platforms and their resources in a way that platform resources are transparent to application designers and developers. In addition, symbIoTe also implements IoT platform federations so that they can securely interoperate, collaborate and share resources for the mutual benefit, and what is more, support the migration of smart objects between various IoT domains and platforms (Figure 1).

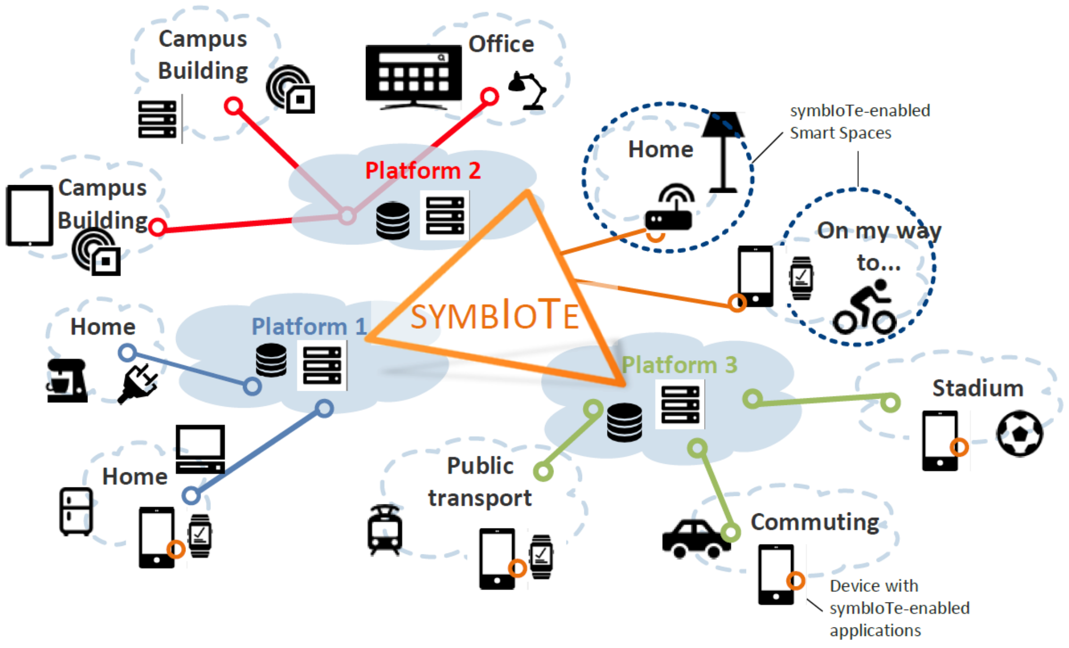


Figure 1: symbIoTe integrates different IoT platforms and ecosystems

2.2 Purpose of the Document and Scope

The purpose of the Deliverable D5.2 entitled “Report on System Integration and Application” is two-fold. On the one hand, it reports on the work done in Task T5.1. On the other hand, it provides detailed planning and a status update of the user application development (Task 5.2) upon the existing core functionality.

This document presents a snapshot of the work done and planned. It reflects the state of the work as it is mid of August 2017.

2.3 Task T5.1 Objectives

Task T5.1 is concerned with the definition of an implementation framework for the symbIoTe architecture. This activity serves as guide for all the implementation tasks in the technical WPs (T2.2, T2.3, T3.3, T4.1, T4.2 and T4.3 respectively), to set common methodologies, tools and workflows for the development of the symbIoTe components. A second set of activities concerns prototype integration of software components developed in previous WPs.

2.4 Task T5.2 Objectives

Task T5.2 implements the symbIoTe use case-related applications, including tools and application workflows, followed by initial functional test that will validate their features. The use cases involved are designed and defined in Tasks T1.1 and T1.3 (and in their respective deliverable documents[1], [2]). In addition to the implementation of the use case scenarios, T5.2 also interacts with the efforts of symbIoTe Open Call partners (WP6), who base their application on the defined domain-specific symbIoTe APIs. Finally, Task T5.2 provides an empirical set of guidelines for the implementation of symbIoTe applications. As a part of WP 5, Task 5.2 ends in month 26.

2.5 Document Structure

The rest of the Deliverable D5.2 is structured as follows. Chapter 3 gives an overview on existing components that belong to the core software. This chapter also describes the installation process of a cloud instance. An overview of the API's used by the core as well by some user code are documented. Further individual components and details about their integration process are presented. This is followed in Chapter 4 by stress tests on performance requirements that were defined in previous deliverable D1.3 [3].
Chapter 5 provides an overview on the design of the user applications which have been briefly introduced in Deliverable D1.3 [2], as well as their development and integration status, followed by a brief platform integration status. Chapter 6 concludes this Deliverable D5.2. Chapter 9 includes the Appendix with information on how to build and install the symbIoTe Core and Cloud.

3 System integration

This chapter provides information regarding the system integration, such as the project structure, the description of the source tree, dependencies, code coverage, as well as feature planning information. Moreover, a brief look at the architecture is also given.

3.1 symbIoTe architecture

As it was already stated in detail in the symbIoTe's report on "Initial and final system architecture and requirements" (see D1.2 [24] and D1.4 [3] respectively), the symbIoTe approach comprises four layered domains, as depicted in Figure 2:

1) **Application domain (APP)** offers a high-level API for managing symbIoTe's virtual IoT environments for supporting cross-platform discovery and management of resources.
2) **Cloud domain (CLD)** hosts the cloud-adjusted building blocks of specific platforms.
3) **Smart Space domain (SSP)** comprises smart objects, IoT gateways as well as local computing and storage resources, enabling dynamic sensor discovery and configuration in smart spaces available within home environment.
4) **Device domain (SD)** spans over heterogeneous devices which are capable to dynamically blend with the surrounding environment and are discovered by the symbIoTe middleware which performs the initial "introduction" of devices within a smart space.

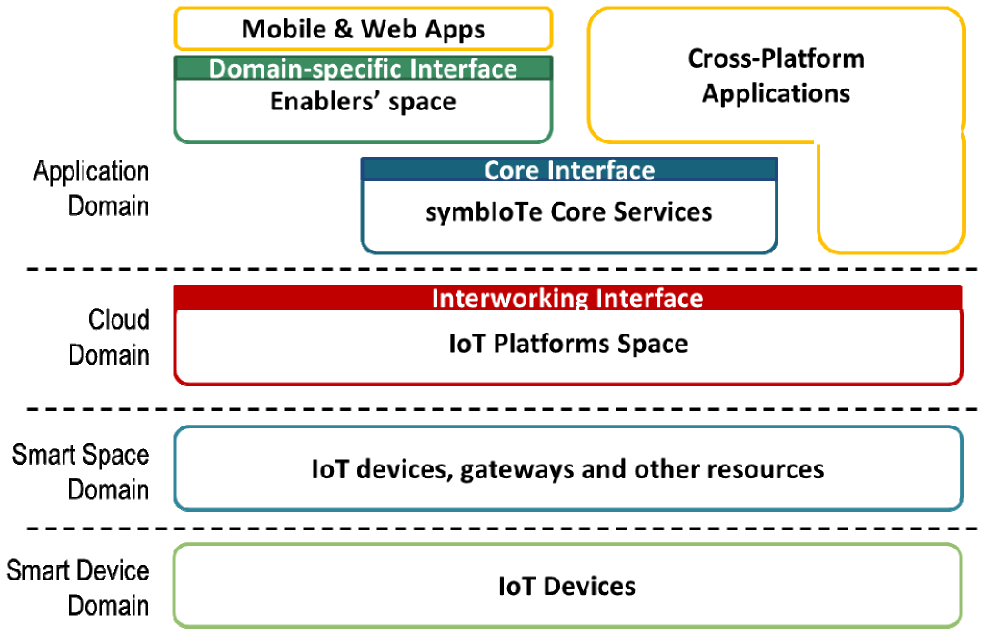


Figure 2: The symbIoTe high-level architecture

symbIoTe has also introduced four different interoperability modes, which are referred to as Compliance Levels (Figure 3), and that aim to enable an incremental deployment of functionalities across the architectural domains (APP, CLD, SSP, and SD). Platform owners are free to choose the compliance level they desire and consequently, the level of the collaboration with other platforms within the symbIoTe-enabled ecosystem. Naturally, different components are required to be installed in different domains according to the desirable compliance level.

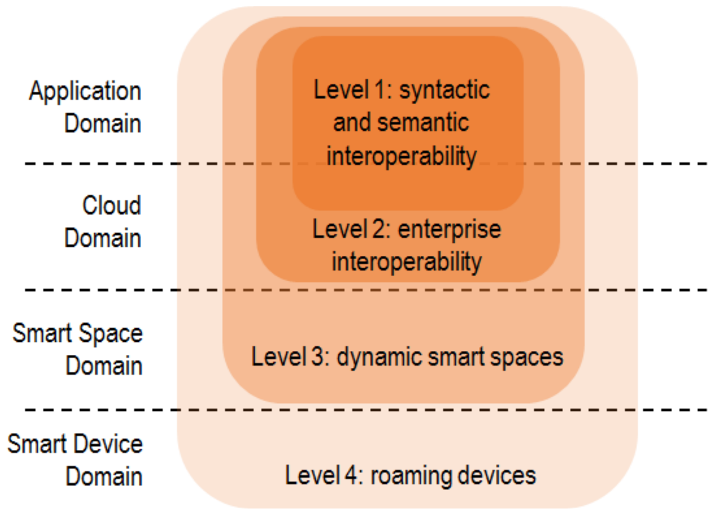

Application Domain
Cloud Domain
Smart Space Domain
Smart Device Domain
Level 1: syntactic and semantic interoperability
Level 2: enterprise interoperability
Level 3: dynamic smart spaces
Level 4: roaming devices

Figure 3: symbIoTe Compliance Levels

The current release implementation, R2, aims to provide components that support Level 1 (L1) compliance modes. This will allow for a unified view on various platforms and their resources. The release currently under development, R3, will provide functionality for a platform federation, where accessing each platform's services (and resources) from within another platform is made possible. Both compliance levels require components placed in two domains - APP and CLD. IoT platforms, which want to become part of the symbIoTe ecosystem, need to integrate the required symbIoTe components in the CLD according to the compliance level they choose. This will enable semantic interoperability and open access to IoT services that a platform chooses to register and make discoverable via the symbIoTe Core Services or inside a federation. Furthermore, symbIoTe supports authenticated and authorized access to IoT services, which will allow the platform to retain access control on their devices.

The symbIoTe components are available in GitHub1. The components are bundled in several super-repositories, which make use of the git submodules, according to the domain, which need to be instantiated. These are the symbIoTe Core2 and Cloud3 repositories. symbIoTe Core Contains all the components instantiated in the core, while symbIoTe Cloud contains all the components that should be instantiated in the respective platform side.

Providing IoT services is an extremely distributed and performance oriented procedure. Therefore, we designed a system that matches these needs. Consequently, we decided to follow the microservices architecture, which aims to provide better scalability, performance and code maintenance. We adopted this approach both for the symbIoTe core and cloud as well as for the enablers. Thus, the functionalities of the symbIoTe Core and the

[1] https://github.com/symbiote-h2020
[2] https://github.com/symbiote-h2020/SymbioteCore
[3] https://github.com/symbiote-h2020/SymbioteCloud

symbIoTe Cloud are composed by using several components that are glued together forming a “microservice architecture”. The repositories refer to all microservice components that are relevant for the respective functionality. All the code developed under symbIoTe is available in the symbIoTe page[4].

Figure 4 shows the relevant microservices and their relation. The term "relevant” in this context means, that there are more services that support the shown services by providing infrastructure like logging, configuration, persistent storage and inter-component communication. These auxiliary services do not add to the logical functionality and are thus omitted here.

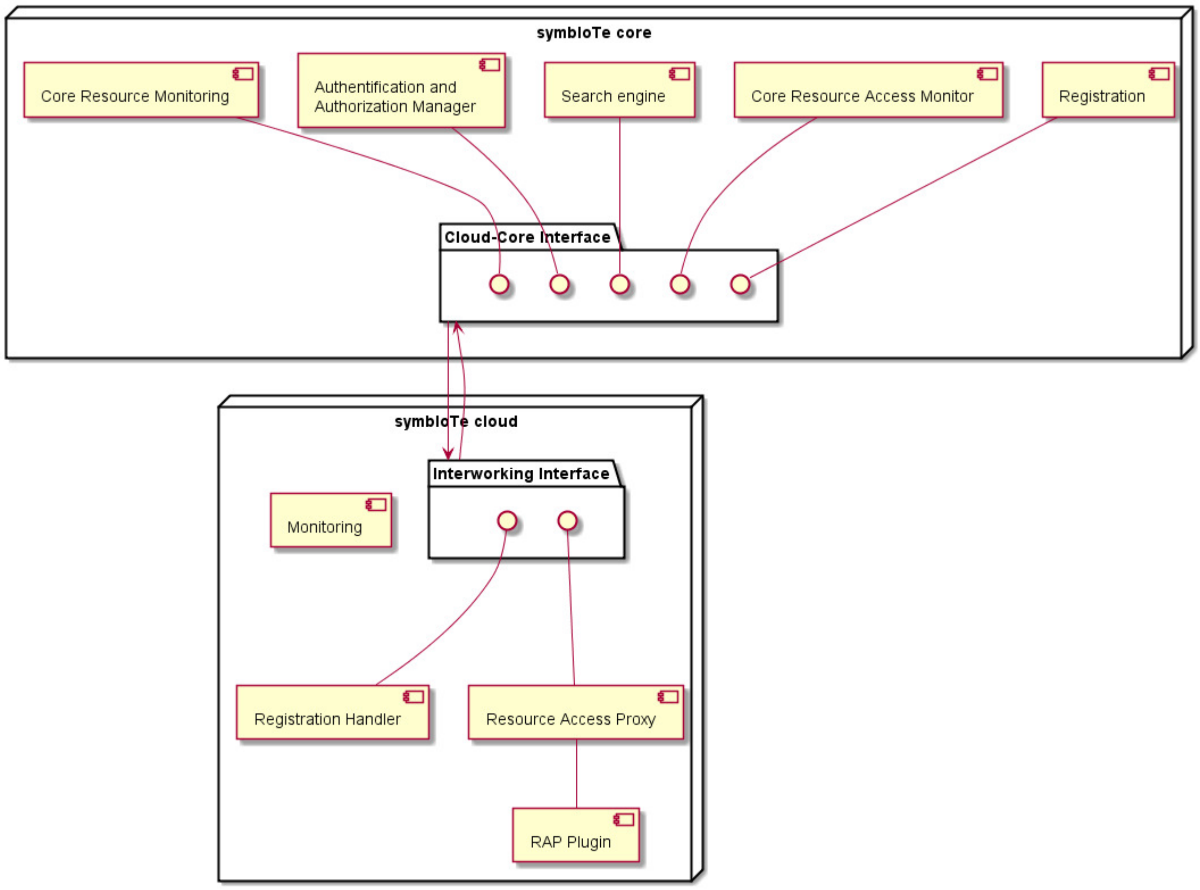


Figure 4: Existing microservices

3.2 Project Build & Deployment

All source codes are available in github.

A good starting point is

- https://github.com/symbiote-h2020/SymbioteCore for the core components and

[4] https://github.com/symbiote-h2020

- https://github.com/symbiote-h2020/SymbioteCloud for the platform-side components.

Both repositories contain instructions of how to install and build all components.

From time to time comments are resolved and changes settled. The resulting version is then published under the name “README” as part of the software and is as such available under the above mentioned locations.

3.3 Common integration information across components

This section presents integration information common for all symbIoTe core and cloud components.

3.3.1 Programming Language

For developing the symbIoTe framework, we chose the Java programming language, since it is a very popular language, easy to write, compile and debug, and offers numerous frameworks (e.g., Spring5 framework), which make it extremely easy to develop simple, portable, fast and flexible Java virtual machine (JVM)-based systems and applications. Furthermore, the choice of java is highly compatible with the microservices approach, since there are many java frameworks facilitating the development of microservices (e.g., Spring Boot6, Spring Cloud7).

Even though symbIoTe is entirely developed in Java, we provide standard communication mechanisms to all the symbIoTe components (e.g., message queues). Therefore, the platform-specific plugins that are required to enable the integration of an IoT platform to the symbIoTe framework are language independent and must not necessarily be developed in Java.

3.3.2 Generic Source Tree Information

In symbIoTe, we follow the Standard Directory Layout for java. Meaning all the symbIoTe components include the following subfolders:

- `src/main/java/eu/h2020/symbiote`: Application/Library sources
- `src/main/resources`: Application/Library resources (e.g., bootstrap.properties configuration file of Spring Boot)
- `src/test/java/eu/h2020/symbiote`: Test sources
- `src/main/resources`: Test resources

[5] https://spring.io/
[6] https://projects.spring.io/spring-boot/
[7] http://projects.spring.io/spring-cloud/

3.3.3 Building tool

Gradle[8] was our final choice for a building tool, due to its simplicity of creating and maintaining building scripts, its excellent documentation and performance. In order to facilitate the simplicity and quality of the development procedure, we used the following Gradle plugins listed in Table 1.

Plugin	Version	Description
`java`	`default`	Plugin necessary for java
`org.springframework.boot`	`1.5.4.RELEASE`	Plugin necessary for Spring Boot
`io.spring.dependency-management`	`1.0.0.RELEASE`	A Gradle plugin that provides Maven-like dependency management functionality
`jacoco`	`default`	Gradle plugin that generates Jacoco reports from a Gradle Project.
`org.owasp.dependencycheck`	`1.4.5.1`	A software composition analysis plugin that identifies known vulnerable dependencies used by the project.
`eclipse`	`default`	Plugin for Eclipse
`idea`	`default`	Plugin for Intellij IDEA
`com.cinnober.gradle.semver-git`	`2.2.1`	Gradle plugin that combines git tags and semantic versioning, and sets the gradle version property accordingly.

Table 1: Gradle Plugins

3.3.4 Common Java Dependencies across components

Towards simplifying and accelerating the implementation procedure, we also made use of Spring framework and specifically of the Spring Boot project (1.5.3.RELEASE) and Spring Cloud project (Dalston.RELEASE). In Table 2 all the common Java dependencies across projects are listed.

Table 2: Java Dependencies

Group Id	Artifact Id	Version	Type
`org.springframework.cloud`	`spring-cloud-starter-config`	`Dalston.RELEASE`	`compile`
`org.springframework.cloud`	`spring-cloud-starter-eureka`	`Dalston.RELEASE`	`compile`
`org.springframework.cloud`	`spring-cloud-starter-zipkin`	`Dalston.RELEASE`	`compile`
`org.springframework.boot`	`spring-boot-starter-amqp`	`1.5.4.RELEASE`	`compile`
`com.github.symbiote-h2020`	`SymbIoTeLibraries`	`0.2.0`	`compile`
`junit`	`junit`	`4.+`	`testcompile`

[8] https://gradle.org/

org.springframework.boot	spring-boot-starter-test	1.5.4.RELEASE	testcompile

All spring libraries are published under an Apache 2.0 license. Junit is only used during the build process but comes with an Eclipse Public License - v 1.0. Both allow to run symbiote applications without the need to publish source code (like in the GPL).

3.3.5 External Tools

symbIoTe has further dependencies on external tools, so that certain aspect of the projects can be easily realized:

- *RabbitMQ*[9] (version 3.6.9+): message queue server for internal messaging between same domain components. RabbitMQ's use is granted under a "Mozilla Public License"
- *MongoDB*[10] (version 3.2.13+): database used by symbIoTe components. The related license is the "GNU AFFERO GENERAL PUBLIC LICENSE".
- *Icinga 2*[11]: for monitoring registered resources. Icinga is licensed under the terms of the GNU General Public License Version 2.
- *Nginx*[12] (version 1.12.0): for enabling access of platform components with the external world (i.e., applications, enablers, symbIoTe core). Nginx is released under the terms of a BSD-like *license.*

3.3.6 Continuous Integration

During implementation, we used the branching model as described in deliverable D5.1 [4] along with the continuous integration server *Travis*[13]. The test reports are also automatically pushed to *codecov*[14]. Specific testing information per component is provided in the following Section 3.4.

3.4 Software Releases

The project has already performed two major releases of the symbIote software, while a third one is under preparation. The following table summarizes the symbIoTe software releases, providing also the respective GitHub links.

Release	Date	GitHub Link
0.1.0 (R1)	21/02/2017	Core: https://github.com/symbiote-h2020/SymbioteCore/releases/tag/0.1.0 Cloud: https://github.com/symbiote-h2020/SymbioteCloud/releases/tag/0.1.0
0.2.0 (R2)	22/05/2017	Cloud: https://github.com/symbiote-h2020/SymbioteCore/releases/tag/0.2.0

9 https://www.rabbitmq.com/
10 https://www.mongodb.com/
11 https://www.icinga.com/products/icinga-2/
12 https://nginx.org/en/
13 https://travis-ci.org/
14 https://codecov.io/github/symbiote-h2020

		Core: https://github.com/symbiote-h2020/SymbioteCloud/releases/tag/0.2.0
0.2.1	20/07/2017	Core: https://github.com/symbiote-h2020/SymbioteCore/releases/tag/0.2.1 Cloud: https://github.com/symbiote-h2020/SymbioteCloud/releases/tag/0.2.1
0.3.0 (R3)	expected	To be provided.

3.5 Component-specific integration information

This section provides specific integration information per component. Although not finalized and released at the cut-off date for this deliverable, the implementation of R3 is work in progress and has advanced a lot. Thus the following chapters also refer to the current status of R3. Features mentioned as “implemented” there are certain to be incorporated in that release.

3.5.1 Component: Administration

3.5.1.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/Administration`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/Administration
Code Coverage Snapshot	71%

3.5.1.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Interface definition • User registration • Platform registration, modification, removal	
0.2.0	• Interface definition (enhancements) • Visual improvements • App registration • Security credential passing to users	• Admin access to logs (pushed to 0.3.0)
0.3.0	• Interface definition (enhancements) • List registered resources • Admin access to logs • Admin user actions	

3.5.2 Component: Authentication and Authorization Manager

3.5.2.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/AuthenticationAuthorizationManager`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/AuthenticationAuthorizationManager
Code Coverage Snapshot	80%

3.5.2.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0		
0.2.0	• Interface definition • App registration • Platform registration • Authentication • Core tokens issuing • AAMs repository	• Token validation (done partially, pushed to 0.3.0)
0.3.0	• Interface definition (enhancements) • Revoking tokens • Home to Core tokens translation • Illegal access detection • Password change / reset	

3.5.3 Component: Cloud-Core Interface

3.5.3.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/CloudCoreInterface`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/CloudCoreInterface
Code Coverage Snapshot	52%

3.5.3.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Endpoint for Registration Handler requests • Async. core communication (RabbitMQ)	
0.2.0	• Endpoint for Registration Handler requests (enhancements) • RDF endpoint for Registration Handler requests	
0.3.0	No new features planned	

3.5.4 Component: Core Interface

3.5.4.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/CoreInterface`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/CoreInterface

Code Coverage Snapshot	30%

3.5.4.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Endpoint for searching • Parameterized query translation • Endpoint for access to resources • Async. core communication (RabbitMQ)	
0.2.0	• Endpoint for searching (enhancements) • Parameterized query translation (enhancements) • Endpoint for access to resources (enhancements)	
0.3.0	• Endpoint for updating resources' status	

3.5.5 Component: Core Resource Access Monitor

3.5.5.1 Generic Information

URL of git repository	https://github.com/symbiote-h2020/CoreResourceAccessMonitor
URL of code coverage reports	https://codecov.io/gh/symbiote-h2020/CoreResourceAccessMonitor
Code Coverage Snapshot	67%

3.5.5.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Interface Definition • Redirection to actual resources	
0.2.0	• Interface Definition (Enhancements) • Redirection to actual resources (Enhancements) • Integration of the Security Handler	• Resource Popularity Tracking (Pushed to 0.3.0)
0.3.0	• Collect access information from RAP • Resource Popularity Tracking • Integration of the Security Handler newest version	

3.5.6 Component: Core Resource Monitoring

3.5.6.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/CoreResourceMonitor`

URL of code coverage reports	https://codecov.io/github/symbiote-h2020/CoreResourceMonitor
Code Coverage Snapshot	30%

3.5.6.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Interface definition	
0.2.0	• Status check polling • Status notification support	
0.3.0	• Extended monitoring	

3.5.7 Component: Monitoring

3.5.7.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/Monitoring`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/Monitoring
Code Coverage Snapshot	5%

3.5.7.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Interface definition	
0.2.0	• Monitor information • Handling of devices	• Resource status pull (pushed to 0.3.0)
0.3.0	• Handling of devices (enhancements) • Resource status pull	

3.5.8 Component: Registration Handler

3.5.8.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/RegistrationHandler`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/RegistrationHandler
Code Coverage Snapshot	60%

3.5.8.2 Feature History

Release	Planned Features Done	Planned Feature Not Done

0.1.0	• Interface definition • Integration with external components	
0.2.0	• Information model • Integration with external components (enhancements) • Integration with security	
0.3.0	None, all planned features implemented in R0.2.0	

3.5.9 Component: Registry

3.5.9.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/Registry`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/Registry
Code Coverage Snapshot	46%

3.5.9.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Interface definition • IoT Services (resources) registration • Platforms registration • Resource metadata storage • Updates of resources • Payload for interfaces	
0.2.0	• Registration validation • Updates of resources • (enhancements)	
0.3.0	• Registration validation (enhancements) • Resource metadata storage (enhancements)	

3.5.10 Component: Resource Access Proxy

3.5.10.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/ResourceAccessProxy`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/ResourceAccessProxy
Code Coverage Snapshot	10%

3.5.10.2 Feature History

Release	Planned Features Done	Planned Feature Not Done

0.1.0	• Interface definition • Northbound API interface	•
0.2.0	• Interface definition (enhancements) • Northbound API interface (enhancements) • Subscription support • Plugin interface	•
0.3.0	• OData support	•

3.5.11 Component: Search

3.5.11.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/Search`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/Search
Code Coverage Snapshot	51%

3.5.11.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0	• Interface definition • Endpoint for handling search requests • Searching resources	•
0.2.0	• Endpoint for handling search requests (enhancements) • Searching resources • (enhancements)	•
0.3.0	• Ranking • Filtering	•

3.5.12 Component: Semantic Manager

3.5.12.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/SemanticManager`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/SemanticManager
Code Coverage Snapshot	47%

3.5.12.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0		

0.2.0	• Storage of models • Storage of instance of data • Translate JSON -> RDF • Translate RDF -> JSON • Construct SPARQL from JSON • SPARQL endpoint for query	• Validate instance of data (done partially, pushed to 0.3.0)
0.3.0	• Storage of models (enhancements) • Validate PIM	

3.5.13 Library: Security Handler

3.5.13.1 Generic Information

URL of git repository	`https://github.com/symbiote-h2020/SymbIoTeSecurity`
URL of code coverage reports	https://codecov.io/github/symbiote-h2020/SymbIoTeSecurity
Code Coverage Snapshot	29%

3.5.13.2 Feature History

Release	Planned Features Done	Planned Feature Not Done
0.1.0		
0.2.0	• Request core token • Request foreign token • Certificate validation • Token validation	• Challenge-response procedure (pushed to 0.3.0) • Revocation procedure (pushed to 0.3.0)
0.3.0	• Challenge-response procedure • Revocation procedure • Request foreign token (enhancements) • Access policy validation	

4 Stress testing of symbIoTe components

Deliverable D1.2 [24] contains a list of requirements of the symbIoTe system. Among them there were *SHOULD* requirements concerning response times (requirement #25 and #26), latency (#27) and expected volume of the load on the search service side (#31). To ensure that provided services fulfil those performance requirements we decided to create a stress testing testbed and testing scenarios and perform a set of tests on varied number of users and concurrent requests to the system based on R2 release of the symbIoTe. This chapter contains description of the tools used, scenarios as well as results and conclusions based on them.

4.1 Testing scenario

For the testing purpose we used Gatling[15] framework. It is an open source load and performance testing tool for web applications written in Scala[16]. It allows defining:

- Request types
- Frequency of requests
- Users
- Test scenarios

Moreover, because symbIoTe components are using Gradle for building, it is important that Gatling is able to integrate with it by using a plugin[17].

Tests were performed on two setups:

1. *local* - Laptop, i5-3360M CPU @ 2.80 GHz, 2 cores (HT), 16 GB RAM, Windows 10. Components running on machine: Search, CoreInterface, CoreConfigService.
2. *remote* – symbIoTe development server running on OpenStack: Intel(R) Core(TM) i7-4910MQ CPU @ 2.90GHz, 4 cores (HT), 16GB RAM
   Running whole Core stack: CloudCoreInterface, CoreInterface, CoreConfigService, Administration, Registry, Core Resource Access Monitor, SemanticManager, Search

In order to simulate real usage of symbIoTe components a testbed generator has been created, which allows generating X number of platforms with Y number of resources each (X and Y can be any positive integer). Generator creates Mobile and Stationary Sensors, which observe random number of properties from a set of predefined properties and are located in a random location from a set of predefined locations. This allowed us to create three different datasets for Search component:

1. *small* – represents actual symbIoTe resources registered at the time of tests. It consisted of roughly 5 platforms with 10 sensors each.

[15] http://gatling.io/
[16] https://www.scala-lang.org/
[17] https://github.com/lkishalmi/gradle-gatling-plugin

2. *large* – generated for 30 platforms with 50 resources each (1500 total resources)
3. *huge* – generated for 100 platforms with 100 resources each (10.000 total resources)

Gatling tool allows simulating real usage of web application by specifying amount of concurrent users accessing the service over some period of time. This technique was used to define a scenario we used for testing. Each users performs two actions:

1. *searchAll* – execution of parameterised search query without specifying any filter options, which returns all resources stored in the Core
2. *searchProperties* – execution of parameterised search query with specified observed property: temperature which returns all resources that are sensing temperature

Tests, for which results are presented in next chapters, were run by ramping 10, 20, 30, 40, and 50 users over period of 10 seconds. For each user between execution of *searchAll* and *searchProperties* queries there was a 3 seconds pause, simulating real clients interacting with the platform.

4.2 Achieved Results

Parameters that were the most important for us during the tests were following:

- success rate – how many requests finished successfully, how many failed
- average response time
- percentiles: 75th, 95th and 99th

Because of the microservice architecture and collaboration of multiple services in providing different functionalities chances of high percentiles occurring at least once are pretty high.

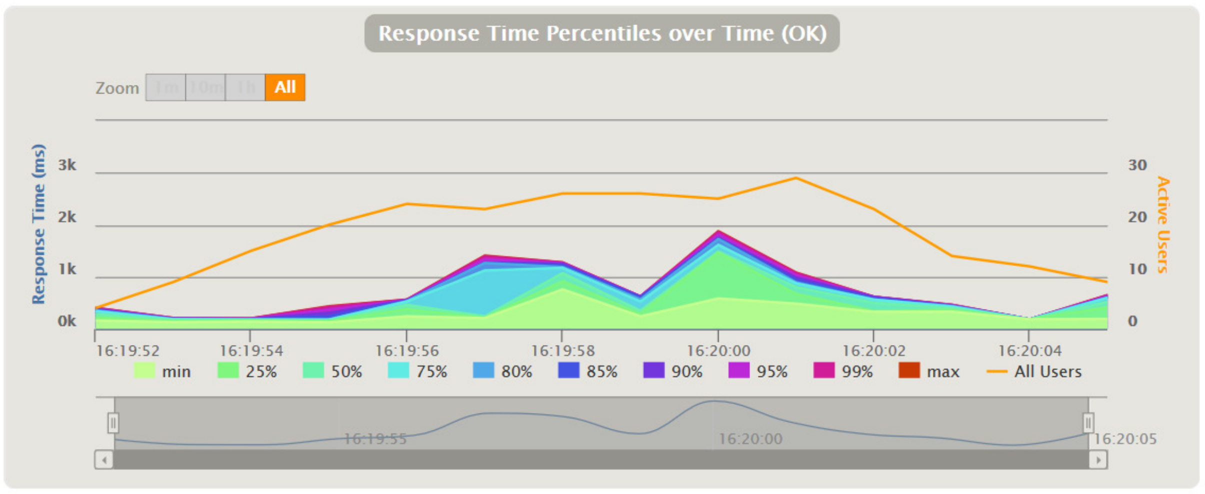
4.2.1 Local Testing

Local testing setup handled *small* dataset without any failed requests for any number of users in the test scenario. The results are presented in Table 3.

Table 3 Local setup, small dataset, and example executions

	10 users	20 users	30 users	40 users	50 users
Mean time	185 ms	199 ms	244 ms	209 ms	241 ms
75th	198 ms	216 ms	269 ms	232 ms	289 ms
95th	329 ms	343 ms	447 ms	369 ms	438 ms
99th	417 ms	397 ms	537 ms	431 ms	525 ms

Example execution for 50 users and response time percentiles are shown in Figure 5.



raph

For a *large* dataset, system handled 10 concurrent users without any fail rate. For more users we started to notice increasing number of failed response rates, going as high as 70-80% failure rate. Detailed data is shown in Table 4.

Table 4: Local setup, large dataset, and average results over multiple tests.

	10 users	20 users	30 users	40 users	50 users
Fail rate (KO)	-	~20%	~50%	~70%	~75%
Mean time	~6.0 s	~13.0 s	~15.5 s	~17.5 s	~17.5 s
75th	~8.5 s	~19.0 s	~20.0 s	~20.0 s	~20.0 s
95th	~9.0 s	~20.0 s	~20.0 s	~20.0 s	~20.0 s
99th	~9.5 s	~20.0 s	~20.0 s	~20.0 s	~20.0 s

Due to already high failure rate of *large* dataset, *huge* dataset has not been tested on a local setup.

4.2.2 Remote Testing

Similar to *local* setup *small* dataset has been handled without any problems, with better time scores – even with 50 users (shown in Figure 6) 99th percentile was around 250 ms.

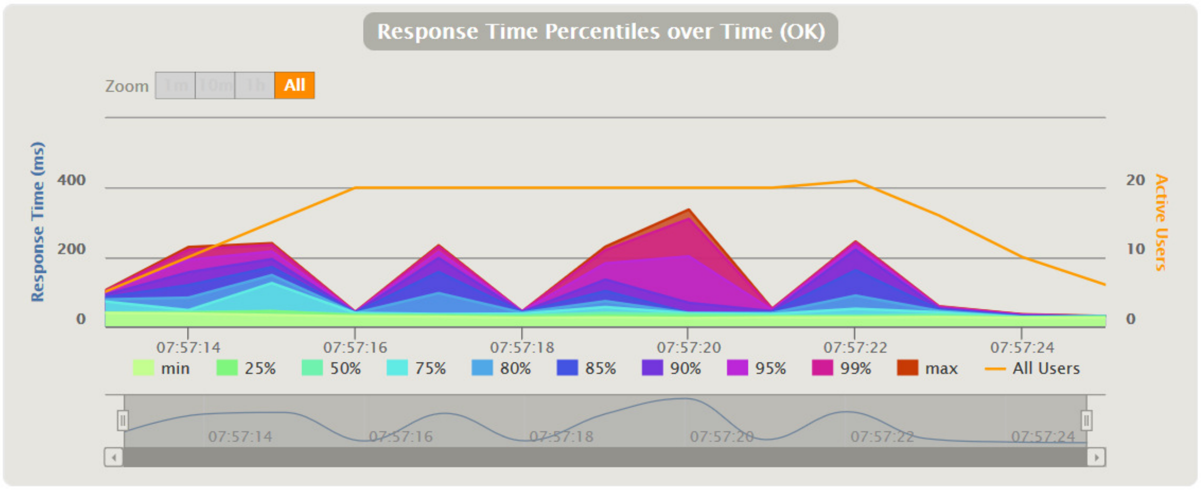


Figure 6: Remote setup, small dataset, 50 users, and percentiles graph

For *large* dataset we noticed increased response times, but behaviour was better than with *local* configuration: failed requests started with 40 concurrent users and occurred 30% of time. Table 5 shows average times by performing repeated tests.

Table 5: Remote setup, large dataset, average results over multiple tests.

	10 users	20 users	30 users	40 users	50 users
Fail rate (KO)	-	-	-	~30%	~50%
Mean time	~1.8 s	~5.0 s	~9.2 s	~13.0 s	~15.0 s
75th	~2.3 s	~7.5 s	~14.0 s	~20.0 s	~20.0 s
95th	~2.6 s	~9.0 s	~15.2 s	~20.1 s	~20.1 s
99th	~3.0 s	~9.5 s	~16.3 s	~20.2 s	~20.2 s

Example executions for 30 users and 50 users are presented in Figure 7 and Figure 8.

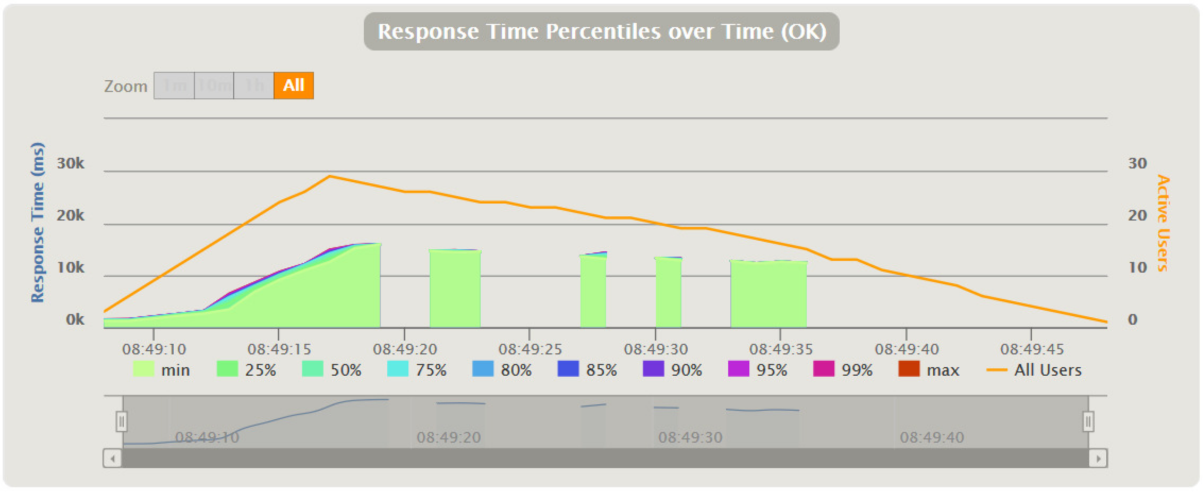


Figure 7 Remote setup, large dataset, 30 users, and percentiles graph

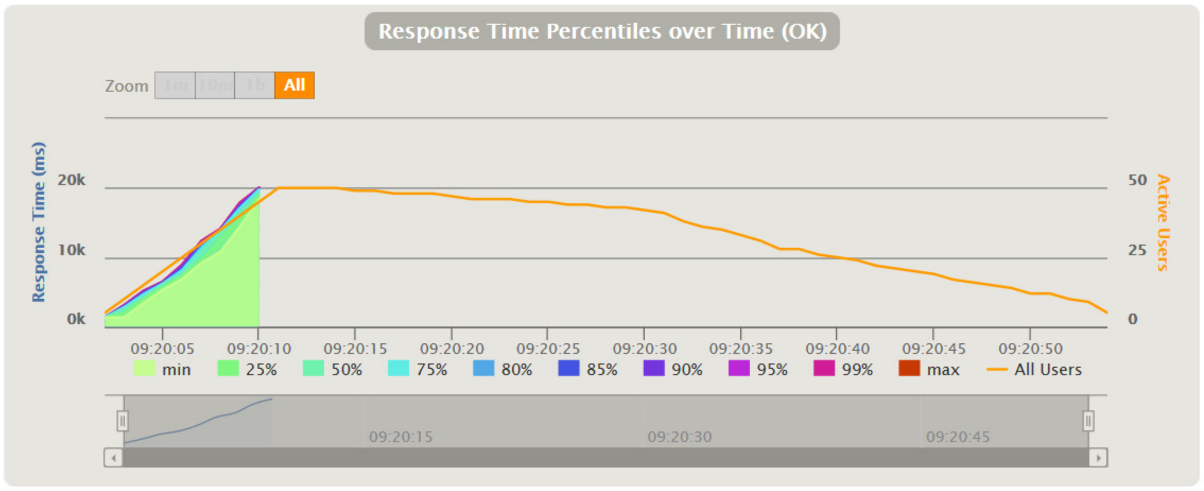
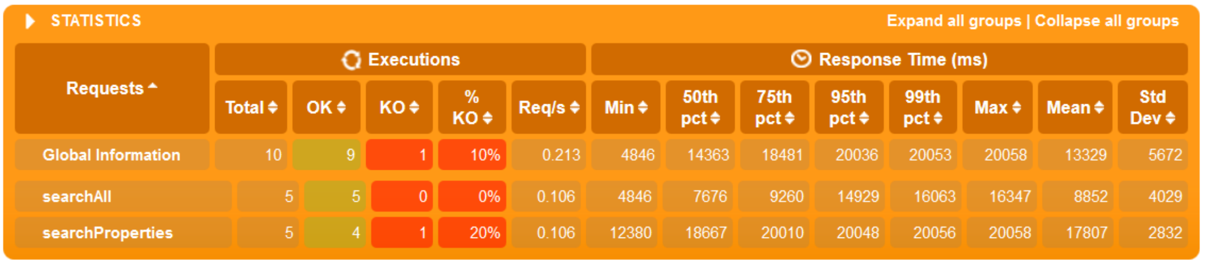


Figure 8 Remote setup, large dataset, 50 users, and percentiles graph

For the *Remote* setup we also performed tests on *huge* dataset, consisting of 10.000 resources (Figure 9). These tests already with five concurrent users achieved a knock-out (KO) rate of around 15% with similar times as with *large* dataset for 40-50 users. We also performed tests for 10 users which reached a KO rate of 60%.


STATISTICS
Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	10	9	1	10%	0.213	4846	14363	18481	20036	20053	20058	13329	5672
searchAll	5	5	0	0%	0.106	4846	7676	9260	14929	16063	16347	8852	4029
searchProperties	5	4	1	20%	0.106	12380	18667	20010	20048	20056	20058	17807	2832

Figure 9 Remote setup, huge dataset, 5 users, and statistics

The amount of data that was generated and transferred for *huge* dataset is really big. Query for all resources (*searchAll)* returns a 6 MB JSON with 230.000 lines, and resources observing temperature (*searchProperties)* response is smaller, with 2.7 MB with 88.000 lines of codes. Executing even single request (using POSTMAN or similar tool for executing HTTP POST requests) requires around 6 seconds to load.

4.3 Findings

Tests have shown that symbIoTe release 0.2.0 is relatively stable and can handle pretty large amount of requests for medium sized datasets. It also shows that we require introducing specific approaches to handle larger datasets and more concurrent users using the system. This is especially needed because the next release of L1 will introduce more communication between the components, which will increase the complexity of provided functionalities.

One of the solutions that will improve the stability of the system is introduction of *circuit breaker*. Circuit breaker in microservice architecture is responsible for cutting off further requests to specific services in case of heavy load and increased amount of failed requests. This allows for better and faster feedback to the client that the system is under stress and to try allocating and sending requests at later time. This can be handled by providing, e.g., *HTTP 429* response *“Too many requests”* instead of standard *HTTP 500* response if we did not use circuit breaker approach.

Another solution is to provide *load balancing* of the requests between replicated services. For future testing of the system we will perform similar tests for a number of Search instances running in parallel and handling incoming requests in round-robin (or other suitable strategy). This will improve both response times and stability of the system, when expensive operations like SPARQL[18] queries and generating objects for large datasets can be handled individually by replicated services.

Other approaches and improvements are also expected in the REST[19] endpoints and in the internal communication between the components, like for example pagination of the large responses.

[18] https://www.w3.org/TR/rdf-sparql-query/
[19] https://en.wikipedia.org/wiki/Representational_state_transfer

5 symbIoTe Use Case Related Application Implementation

There are currently 12 applications planned for symbIoTe, which are in a state of development or in a state of being implemented within the symbIoTe environment. Details on each of them are provided in this chapter. The final report on the implementation will be provided in a subsequent deliverable (D5.4, title "Integrated Prototype and Developed Applications"). As always, the information is considered in the light of the use cases Smart Residence, Smart Mobility and Ecological Routing, EduCampus, Smart Stadium and Smart Yachting. Table 6 provides a concise overview of applications, platforms and their status.

Table 6: Application details and status

Application name	Related IoT platforms	Level Of Compliance	Application development status	Integration status
Smart Residence				
Smart Healthy Indoor Air	openUwedat nAssist	L3	In development	symbIoTe compliance under development[20]
Smart Area Controller	Symphony NN platform	L2	In development	Symphony is L1 compliant
Home Comfort	Symphony NN platform	L3	In development	Symphony is L1 compliant
Smart Health Mirror SMILA	KIOLA	L1	In development	symbIoTe compliance under development
Smart Mobility and Ecological Routing				
Mobile App	OpenIoT openUwedat MoBaaS	L1 / L2	In development	OpenIoT and openUwedat are fully integrated at compliance Level 1 (for Core Release 2) MoBaaS compliance is under development
Web App	OpenIoT openUwedat	L1 / L2	Web App and routing engine do exist. Extending this application to use air quality as an additional metric is in	OpenIoT and openUwedat are fully integrated at compliance Level 1 (for Core Release 2)

[20] See Section 5.1.1.5

			development	
Edu Campus				
Room Searching App	Building Management System KIT Smart Campus	L2	In development	In development
Smart Stadium				
NN Visitor Application	User Platform Beacon Platform Promotion and Information Platform Remote Ordering Platform	L1	In development, GUIs already available	L1 under development over R2
NN Retailer Application	Remote Ordering Platform User Platform Beacon Platform Promotion and Information Platform	L1	In development, GUIs already available	L1 under development over R2
Promowall (Mobile app, touch screen Smart TVs)	Promotion and Information Platform	L1	Fully developed, symbIoTe integration under development.	L1 under development over R2
Smart Yachting				
Portnet	Navigo Digitale Symphony	L1 / L3 / L4	Enabler under development. Yacht app (integrated in Symphony) under development.	symbIoTe compliance under development for Navigo Digitale. Symphony is already L1 compliant
Centrale Acquisti	Navigo Digitale Symphony	L1	Enabler under development. Yacht app (integrated	symbIoTe compliance under development for Navigo Digitale.

			in Symphony) under development.	Symphony is already L1 compliant.

5.1 Use Case Scenario "Smart Residence"

The Smart Residence use case will implement four different applications that will offer comfort, automation, security, energy efficiency and healthcare services. All of them will use advanced and ubiquitous technologies including sensors and other devices that are integrated in the residential infrastructure. The applications are:

- Smart Healthy Indoor Air: This application is based on the indoor/outdoor air quality monitoring and pursues to improve indoor air quality by giving recommendations and alerts.
- Smart Area Controller: this application is related to the Dynamic Interface Adaptation scenario, where the user's control interface automatically reconfigures itself, according to the controllable CPS in range.
- Home Comfort: this application demonstrates the Energy Saving scenario, which shows how to automatically control home devices, in order to keep environmental parameters (e.g. light, temperature, humidity, etc.) to some predefined comfort values.
- Smart Health Mirror: It is an Ambient Assisted Living (AAL) application to help people, in particular elderly people, to live independently for longer.

5.1.1 Application: Smart Healthy Indoor Air

This application is based on the indoor/outdoor air quality monitoring and pursues to improve indoor air quality. Indoor air quality (IAQ) refers to the quality of the air inside buildings as represented by concentrations of pollutants and thermal (temperature and relative humidity) conditions that affect the health, comfort and performance of occupants. It is important to ensure that the air inside of the building we inhabit on a daily basis is of a good quality. Outdoor generated air pollution is relevant for indoor air quality and health. Exposure to indoor air pollution has been linked to the development of everything from infections to asthma or to poor sleep. It can also cause less serious side effects such as headaches, dry eyes and nasal congestion.

Sensing and Control Systems SL partner's (S&C) roadmap aims to create a smart home/office connected with the city. The current S&C's platform, nAssist[21], monitors and controls a number of direct parameters related to indoor air quality, such as CO_2 levels, humidity and temperature. In addition, this platform monitors and controls other factors that are important for indoor environmental quality considerations such as light and noise since they also affect occupants.

[21] http://www.sensingcontrol.com/solutions/customizable-iot-platform.html

The idea is to increase this framework to understand how indoor and outdoor sources of pollution, heat and humidity, together with the ventilation and air conditioning systems, affect the indoor air quality in buildings. It also begins to address methods of controlling those factors in order to improve quality of the indoor air for occupants' health, comfort and performance. To achieve this goal, the smart home will communicate with outdoor air quality data received from other federated platforms. Such data can include air and noise pollution levels. The smart home will react to changes in temperature, humidity, CO_2 levels and noise and maintaining a healthy and safe indoor environment by recommending actions to the user such as using air purifiers, ventilation systems and opening/closing the windows to eliminate unpleasant impacts. S&C aims to provide more robust solutions with focus on clean environment and optimised energy use.

In particular, we will implement a smartphone app capable of monitoring real-time indoor and outdoor air quality information. Without this information, usually we ventilate late and too long. This application (Smart Healthy Indoor Air) will indicate when, how and how long a room should be ventilated taking into consideration that windows are the easiest ventilation option but not the healthiest one depending on the outdoor air quality. There are other ways to provide healthy flow of air throughout the room, such as turning on the air conditioner, or individual air purifiers. The automatic control of some devices, as are air purifiers, ventilation and air conditioning systems are out of the scope of this application.

A more robust application could fit within Smart Energy City concepts by taking into consideration energy efficiency to adjust levels of air purifiers, ventilation and air conditioning systems. This will be pursued when suitable partners apply and join symbIoTe. A goal of S&C is to engage and attract such actors.

5.1.1.1 Design

Outdoor air quality data can be provided by other federated platforms dedicated to offering Smart Cities' services. Also, given the limited number of monitoring stations available and placed at representative spots to record the outdoor air quality, an accurate assessment of spatial variation is highly required. Spatial interpolation techniques applied to the available monitoring data to provide air quality information closest to the location of the smart home will be used. This functionality will be provided by the component named as Interpolator, which is also used in the smart mobility use-case. This showcases how symbIoTe enables the realization of high level services, involving different IoT platforms, and offering their results to different (end-user) applications. Our application will send the GPS location of the smart home and will get the estimated value about the air quality for this specific location from symbIoTe. This is the main benefit of using symbIoTe for our application. We can offer a more robust and precise application without developing new functionalities (Figure 10).

On the other hand, the S&C's platform, nAssist, will make available to symbIoTe data related to the indoor air quality at home: temperature, humidity, CO levels, noise and luminosity. As already explained, exposure to indoor air pollution has been linked to the development of everything from infections to asthma to poor sleep. It can also cause less serious side effects such as headaches, dry eyes and nasal congestion. This information can be helpful for remote healthcare applications.

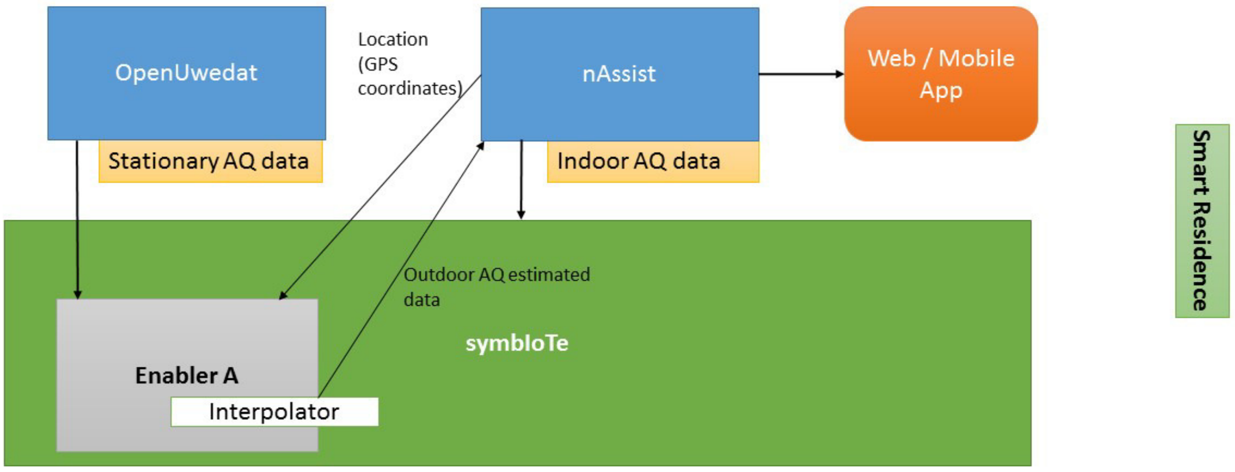


Figure 10: High-level architecture showing the involved platforms, applications and involved symbIoTe components (e.g., enabler) for Smart Healthy Indoor Air application

5.1.1.2 Compliance Level

The S&C's platform nAssist (symbIoTe L2-compliant) will acquire, store and process all data that measure the indoor air quality. Temperature, humidity, CO, and luminosity levels data will be published within symbIoTe.

5.1.1.3 Platform

nAssist is a software platform which has been designed to be easily adapted to different areas of application that require data collection and data processing from logical and physical devices (sensors and actuators). It is composed of:

- Drivers
- Embedded systems
- SDK
- Databases (NoSQL, MS Sql, cloud storage)
- Web applications
- Mobile applications
- Other SW components (scheduler, complex event processing unit and event manager)

nAssist has been constructed in a modular fashion that facilitates a rapid expansion of the system and features addition as well as for installation of different modules across different servers. All the components of the architecture are developed on top of two technologies: Microsoft .NET Framework 4.5 & Microsoft C# and Microsoft SQL Server Azure / SQL Server 2012. nAssist uses Azure for cloud deployments.

The nAssist logic view is composed of four main modules:

- **Hardware Interface:** This module is responsible of the communication with field devices. The communication can be bidirectional depending on the type of devices. The information transfered between devices is parsed, cleaned, formatted and finally sent to processing modules.

- **Processing modules:** The central module handles all the processing/intelligence of the software. It is composed of an event manager, automation routines, alarm/events, and auto generation based on rules.
- **User interfaces:** This module uses standar Web clients.
- **Database:** It provides a data warehouse and business intelligence components to analyse, manage data from devices.

The nAssist platform follows a Services Oriented Architecture (SOA) paradigm. The logic architecture is based on an N-layered architecture to provide solid, flexible, scalable and reliable back-end. The main layers are:

- **Presentation layer:** It is responsible of the interaction between the user and the application.
- **Distributed services layer (web services):** It was built using standard protocols inluding SOAP and WSDL.
- **Application layer:** it contains the tasks of the application and coordinates domain and infrastructure objects.
- **Domain layer:** It represents business/domain concepts, information on the status of business process and implementation of domain rules.
- **Data persistence infrastructure layer:** It captures data from the own system or data from external sources.Cross-cutting infrastructure layer: It is responsible of aspects related to the techology/infrastructure, such as data persistence (repositories), security, logging, operations, etc.

5.1.1.4 *User Interaction*

The GUI design for the app will be based on the current S&C's product, enControl[22]. The product has GUI based on web and smartphone iOS and Android (Figure 11 - Figure 14).

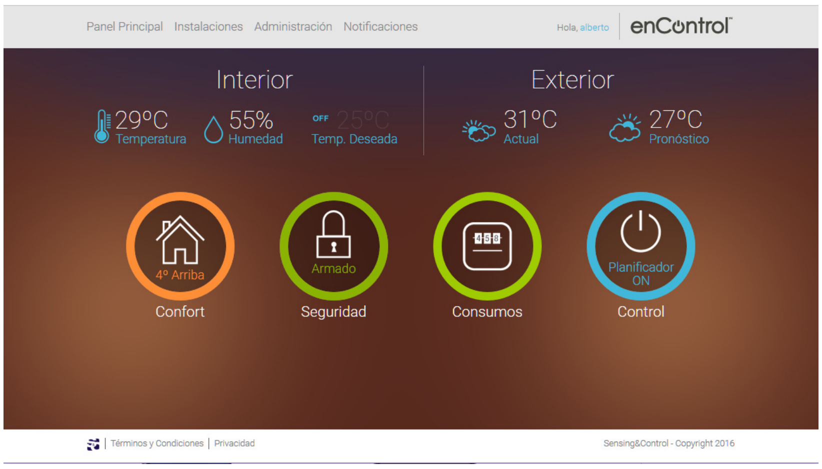


Figure 11: Main screen for all functionalities provided by enControl: comfort, security, energy consumption and automatic control of devices

[22] http://www.encontrol.io/

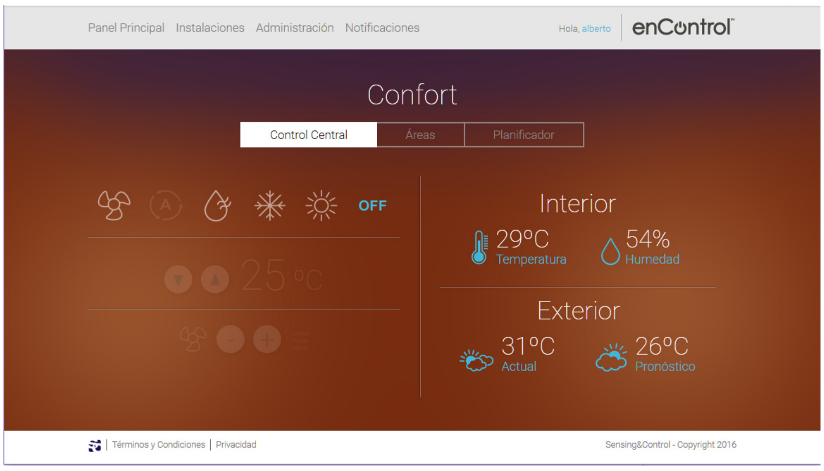


Figure 12: Main screen for comfort

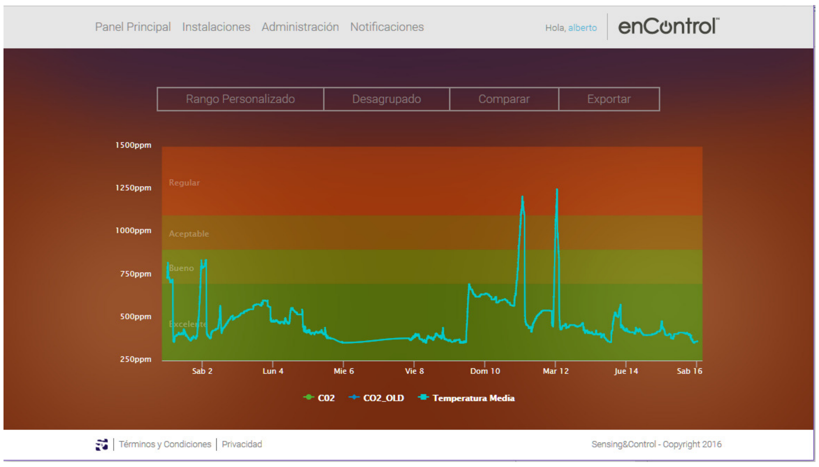


Figure 13: Indoor Air Quality

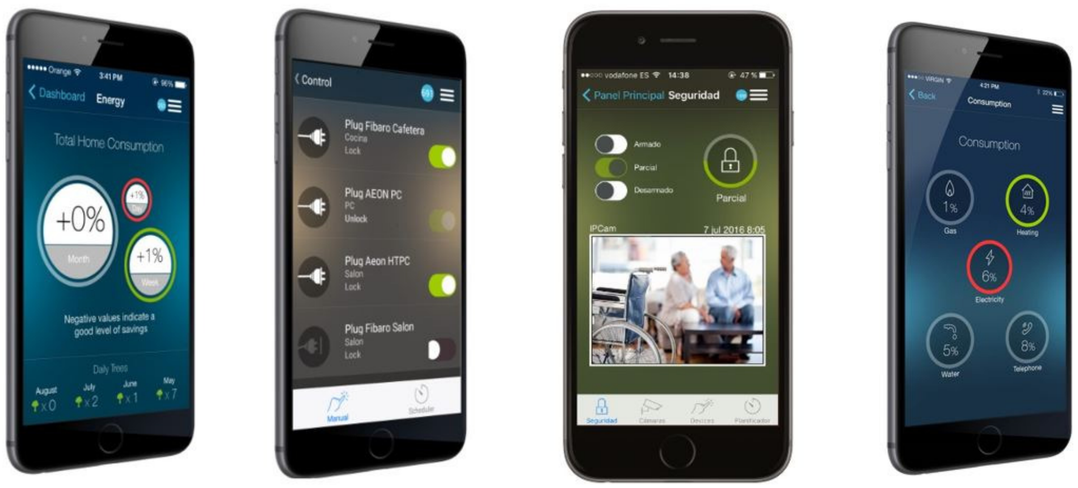


Figure 14: Examples of GUI for iPhone

5.1.1.5 Implementation

The initial implementation of the application will be done in JAVA. Once this implementation is tested, we plan to create C# implementation. Unit testing (register, access data, and notifications) and integration testing are expecting to be done by third party platforms using our data. A "stable/online" development environment of symbIoTe for testing will be required.

Currently, a number of tasks have been implemented in order to adapt the nAssist platform to the current symbIoTe release:

- Adaptation of the platform to host the process in charge of communicating with symbIoTe.
- Adaptation of current user/role access to the new structure.
- Implementation of the register sensor data process.
- Conversion of java libraries to c# to be deployed into our backend system.
- Creation of a dev environment with one installation and sensors related to the use case (smart healthy indoor air).
- Creation of virtual sensors to test interaction with the symbIoTe platform.

Current implementation is at the alpha stage (platform and sensors data registration).

5.1.1.6 Initial Functional Tests

The initial functional tests will be done with current home installations located in Barcelona. It is expected to get outdoor air quality data recorded by the Atmospheric Pollution Vigilance and Forecast Network[23] (the XVPCA) from the Ministry for Territory and Sustainability at the Generalitat of Catalonia. The outdoor air quality can be acquired through other federated platforms more dedicated to offer Smart City services, while the location-specific estimation will be provided by symbIoTe thought out the Enabler.

5.1.2 Application: Smart Area Controller

The mobile app will be capable of controlling cyber-physical devices located in a room, selected by the user, according to his/her needs. The main interface will allow the choice of an area for selecting the devices to control (registered by the platforms that reside in that space). For example, the user will be able to change the temperature in the room, or to move the curtains and change the luminance level.

The application will leverage on a specific enabler, which gives the possibility to filter the devices in the space, based on their position (building, floor, room, etc.). In this way, the application has to query the symbIoTe enabler in order to retrieve the list of the CPSs in the selected area and then to allow the user to control them.

[23] http://dtes.gencat.cat/icqa/start.do?lang=en

5.1.2.1 Design

Figure 15 shows a high-level diagram of the application communicating with the symbIoTe enabler and the symbIoTe compliant platforms involved in the use case.

The platforms register their devices, specifying information with respect to their indoor locations; the Administration console, accessible from the Enabler, allows the SSP administrator to manage the hierarchy of the resources' locations. Lastly, the mobile application is the user entry-point to interact with platform devices. In this specific use case scenario, no Web app is required.

The main benefit using symbIoTe is the possibility to use a single application for dynamic controlling the different devices and platforms present in the house.

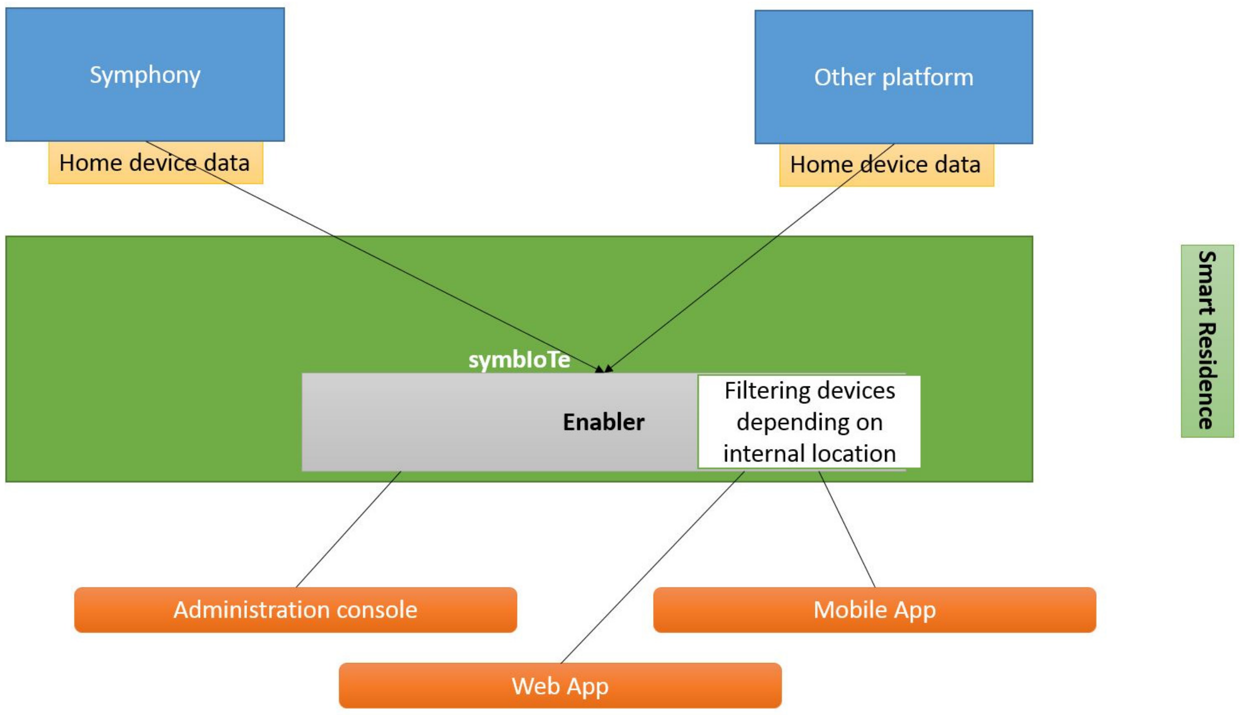


Figure 15: High-level architecture showing the involved platforms, applications and involved symbIoTe components (e.g., enabler) for Smart Area Controller and Home Comfort applications

5.1.2.2 Compliance Level

The app will be L3 compliant, according to the Smart Space definitions and features.

5.1.2.3 Platform

The application will be a mobile application, compatible with Android platform.

5.1.2.4 User Interaction

First of all, the user will select the area, in which she desires to control the devices (Figure 16 and Figure 17): it could be a room, a flat, the entire building or even a desk, according to the configuration made in the service enabler. Afterwards, the interactions will be related to the devices present, and it may vary according to the nature of the CPSs themselves.

5.1.2.5 Implementation

The language used for implementation is Java, since it is depending on the mobile platform of the application (Android). The application will be specific for this scenario purposes, but it can also be used for every use case, which need a direct control of local devices. Currently, the implementation of the application has not been started, because the Enabler has the priority. All the developments will be based on R3 symbIoTe release.

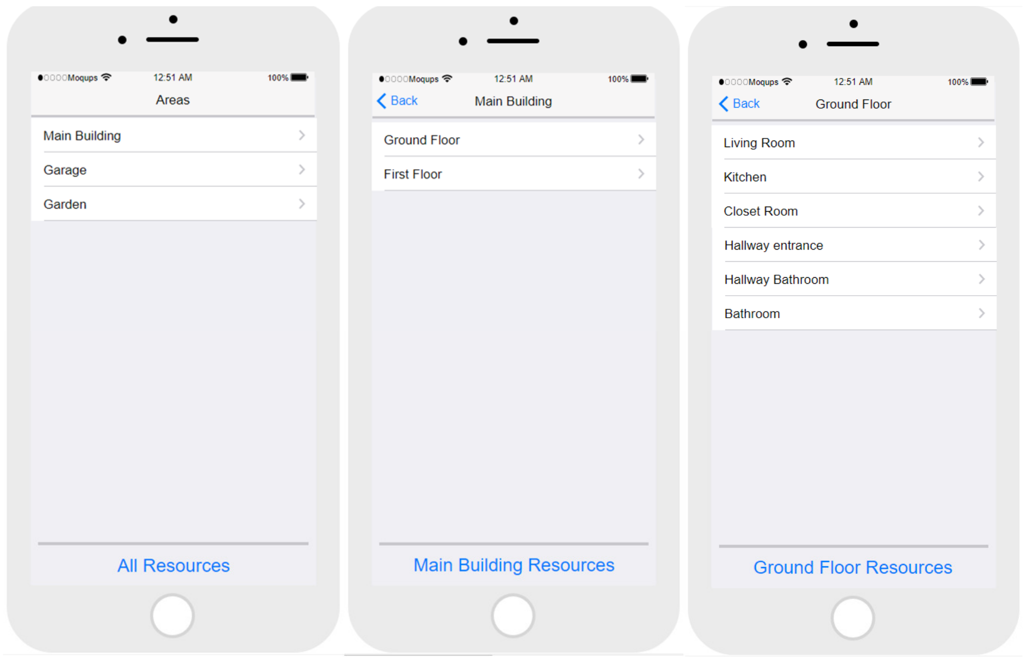


Figure 16: Example of a preliminary GUI for iPhone

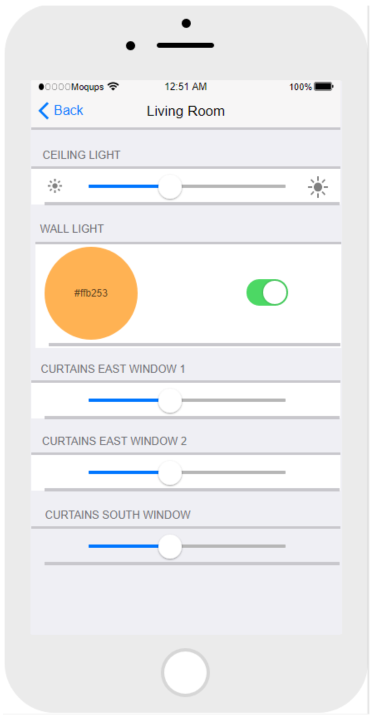


Figure 17: Example of a preliminary GUI for iPhone

5.1.2.6 Initial Functional Tests

The initial functional tests will be done with the current installations located in Pisa.

5.1.3 Application: Home Comfort

The L1 compliant application will be used to automatically control home devices, in order to keep comfort values for home environmental parameters like temperature, luminosity, etc. The application is composed by both a back-end and a front-end part: the first manages the core operations, constantly monitoring the surrounding and controlling devices, in order to reach the desired comfort state; the second acts as a configurator of comfort set-points.

Consequently, the backend will run on a server, having a frontend accessible via web for configuration purposes.

5.1.3.1 Design

As described in Section 5.1.1.1, the platforms register their devices, specifying information with respect to their indoor locations, so the SSP administrator configure the hierarchy of the resources' locations through the administration console. The Web Application, through a graphical user interface, allows the resident to configure all the set-points for the home devices (see Figure 15 for a high-level diagram of the components involved in the use case).

The main benefit using symbIoTe is the possibility to leverage on the interoperability between the various platforms present in the house for driving the environment towards a comfort state.

5.1.3.2 Compliance Level

The app will be L3 compliant, according to the Smart Space definitions and features

5.1.3.3 Platform

The backend application will be Linux based, and the frontend can be accessed through a web page.

5.1.3.4 User Interaction

The core of the application is a service, which does not need any interaction from the user, apart from the configuration of the comfort set-points.

5.1.3.5 Implementation

The language used for implementation of the backend is Python, for the frontend is HTML and Javascript.

Currently, the implementation of the application has not been started, because the Enabler has the priority. All the developments will be based on R3 symbIoTe release.

5.1.3.6 Initial Functional Tests

The initial functional tests will be done with the current installations located in Pisa.

5.1.4 Application: Smart Health Mirror

Based on the use cases described in Deliverable D1.1 (see Chapter 4.4.3), a smart health mirror application called SMILA (*S*mart *M*irror *I*ntelligent *L*iving *A*ssistant) will be developed for use in an elderly person's home. The smart health mirror will be comprised of a semi-transparent mirror with a (Samsung®) tablet hidden behind it, thus creating the illusion of information displayed directly on the surface of the mirror.

SMILA is thought to be an intelligent health assistant recognizing users, displaying context-related health information on screen and coordinating health measurements (weight and blood pressure) using voice input and output. Users are recognized using wearable Bluetooth beacons. Sensors (beacons, scales and blood pressure meters) are registered with the AIT KIOLA telehealth platform [5].

Moreover, commercially available fitness trackers (Fitbit, Nokia health) are included in the scenario. Instead of identifying users via Bluetooth beacons, users are identified by the fitness trackers they are wearing. Fitness trackers are managed by KIOLA and registered as sensors to symbIoTe core as well.

5.1.4.1 Design

As Figure 18 shows describes the interplay of all components involved in the smart residence health use-case. KIOLA stores health and person-related data and registers sensor information of (a) body scales and (b) BLE devices to symbIoTe. An additional plugin serves as a wrapper for commercially available fitness trackers. Using the plugin fitness trackers such as Fitbit or Nokia Health can also be registered with the symbIoTe core. The smart mirror android app SMILA scans for BLE-enabled devices (e.g. a BLE-enabled wristband or a Fitbitness wristband) and uses symbIoTe to resolve IDs to identify users. Analogously, data can be written back locally retrieved from a body scale.

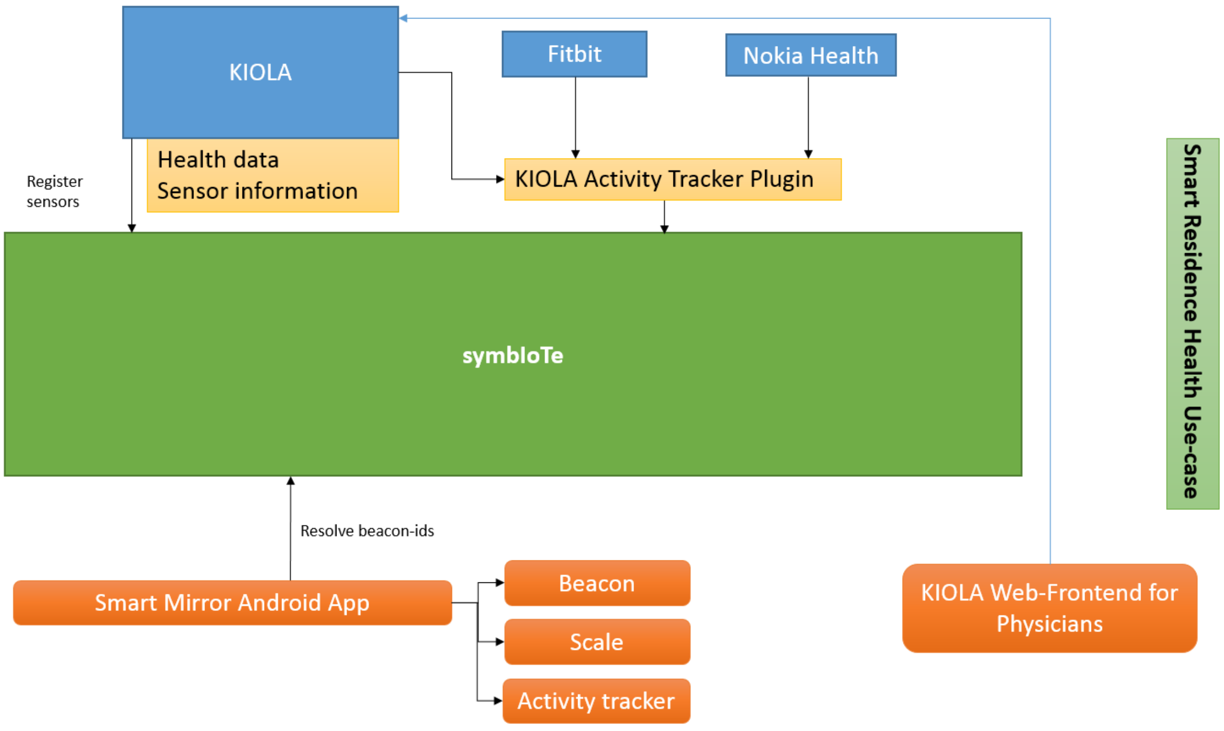


Figure 18: High-Level architecture of all involved Platforms

The main benefit of symbIoTe can be found in a unified environment for application developers to retrieve and store health-related information from various manufactures eliminating the need to integrate various different APIs in the local application.

5.1.4.2 Compliance Level

The application will be L1 compliant.

5.1.4.3 Platform

The application will be a mobile application, compatible with Android (V7.0) platform.

5.1.4.4 User Interaction

A user wearing a Bluetooth beacon (alternatively a fitness tracker) enters the room where the smart mirror is installed. The app running on the smart mirror detects the presence of the beacon and queries the symbIoTe core for a sensor with the given ID. SymbIoTe returns the resource access URL of the beacon in the KIOLA platform. The smart mirror app then accesses this URL retrieving detailed information on the given user. This information includes the first name of the user, a list of measurements the user is supposed to perform as well as data retrieved from the fitness tracker (steps taken this week). Moreover, a history of all past readings is included in the list. Based on this list, the mirror asks the user if he/she would like to perform the appropriate measurement. (e.g., measuring weight). The user voice-confirms the question and steps on the scale. The measurement is taken and finally – again after voice confirmation by the user – submitted back to the KIOLA database.

As the tablet application is stowed away behind the one-way-mirror, usual touch interactions will not be possible. Therefore, the smart mirror will rely on voice input and output as primary medium for communication.

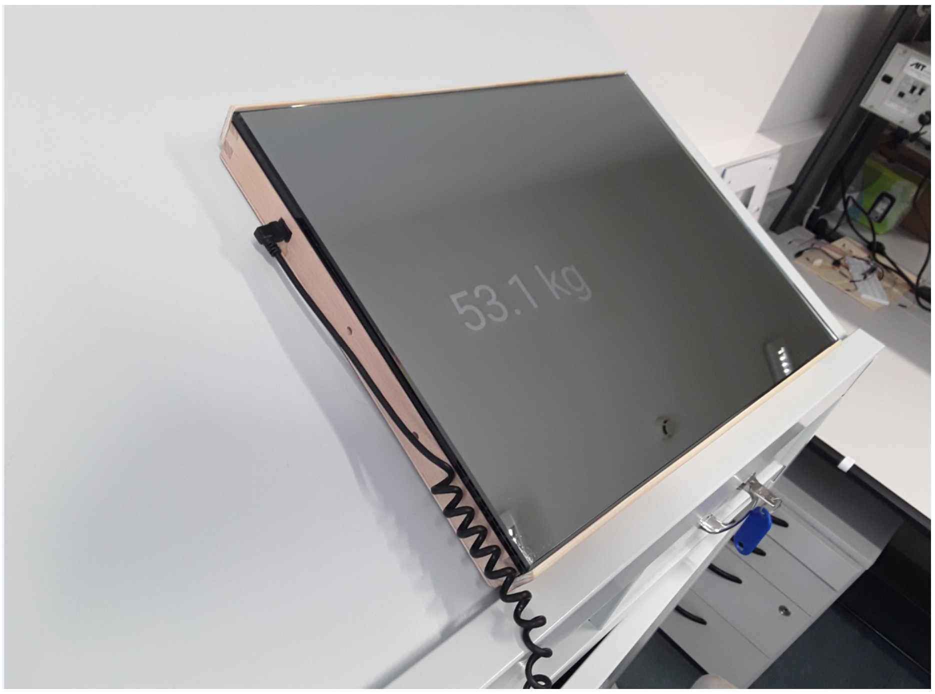


Figure 19: Smart Mirror “SMILA”: prototype while measuring weight

5.1.4.5 Implementation

Implementation of SMILA will be based mainly on Google Android (V7.0) and associated Java libraries. KIOLA is based on the open-source web framework Django. Unit testing is used for both SMILA and KIOLA and is coordinated by Jenkins Build Server. Automated build tests are run every night.

5.1.4.6 Initial Functional Tests

The smart mirror will be evaluated in two phases: First user testing is scheduled for November 2017 with students of the University of Technology in Vienna. Phase II includes testing with elderly people in March 2018 with an updated prototype based on the experiences made in the first evaluation phase.

5.2 Use Case Scenario “Smart Mobility and Ecological Routing”

The Smart Mobility and Ecological Routing Use Case addresses the problems regarding environment pollution and air quality in the major European cities. It does so by collecting air quality data from multiple IoT platforms in different countries and uses such measurements for runners, joggers and cyclists to plan the best routes to their destination.

Through symbIoTe, air quality measurement will be obtained from different platforms. Due to the nature of routing algorithms, these data go through substantial pre-processing with the purpose of associating air measurements to the map’s street segments.

With the streets correctly classified by their air quality, routing engines can take that information into account when computing the most ecological routes to the application's users. These paths can also benefit from other factors such as traffic and available parking, in case the platforms have access to these kinds of sensors.

Finally, users should be able to search for Points of Interest (POIs) following certain criteria, including data from sensors such as available parking or noise levels. Routes for the selected POI can be computed using the previously mentioned service.

All in all, this use case will showcase platform interoperability within the application and cloud domain, where more details can be found in section 6.4 of Deliverable D1.3.

There are three platforms providing services and data to the use case:

- OpenIoT from UNIZG-FER provides air quality data from users' wearables,
- openUWEDAT from AIT, provides air quality data from stationary sensors, and a routing service for the city of Vienna,
- MoBaaS (Mobility Backend as a Service, see [3]) from Ubiwhere provides their routing service.

Additionally, the OpenStreetMap[24] service will be used to obtain cities' Points of Interest. Due to the different features of both routing engines, it was decided to develop two different applications in order to fully show all the features of each service, without the need to reach a compromise between them to be usable in a single application. Through this situation, it is also possible to test the adaptability of the use case by developing a web and a mobile application.

In the next section, we provide details on the two applications: a mobile and a web based application for smart mobility and ecological routing.

5.2.1 Application: Smart Mobility and Ecological Routing Mobile Application

The mobile application aims at delivering users efficient routes to their destinations. These routes will, when available, direct the user to a parking spot near the destination and then provide a route, on foot, that avoids highly polluted areas. Additionally, the application also provides the user with the ability to search for POIs and, subsequently, obtain a route to the select POI.

As such, there are two main functionalities that the application should provide to the user:

- Computation of ecological routes
- POI search

The application will access these services through the Smart Mobility and Ecological Routing enabler, which handles the exchange of data and services between the platforms involved in the use case. As such, the application will communicate with the enabler to show the data and provide the services for the user.

[24] https://www.openstreetmap.org/

Users of the app will be presented with a map after logging in. This map will show the current air quality ratings of their city. Users then may choose an origin and destination points for their desired route and a preferred means of transportation and will be presented with the computed best ecological route. They will also be presented with the amount of pollution they are exposed to and how it can impact their lives. This way, the benefit of taking a more ecological route will become more visible and enhance the quality of life of the users.

Users can also request points of interest, selecting their preferences from a range of possible criteria, such as the type of POI, distance to a certain location, pollution of the area, parking availability of the area, etc. POIs close to the users' criteria will be presented. Users can then choose on and be presented with an ecological routing from the service previously described.

5.2.1.1 Design

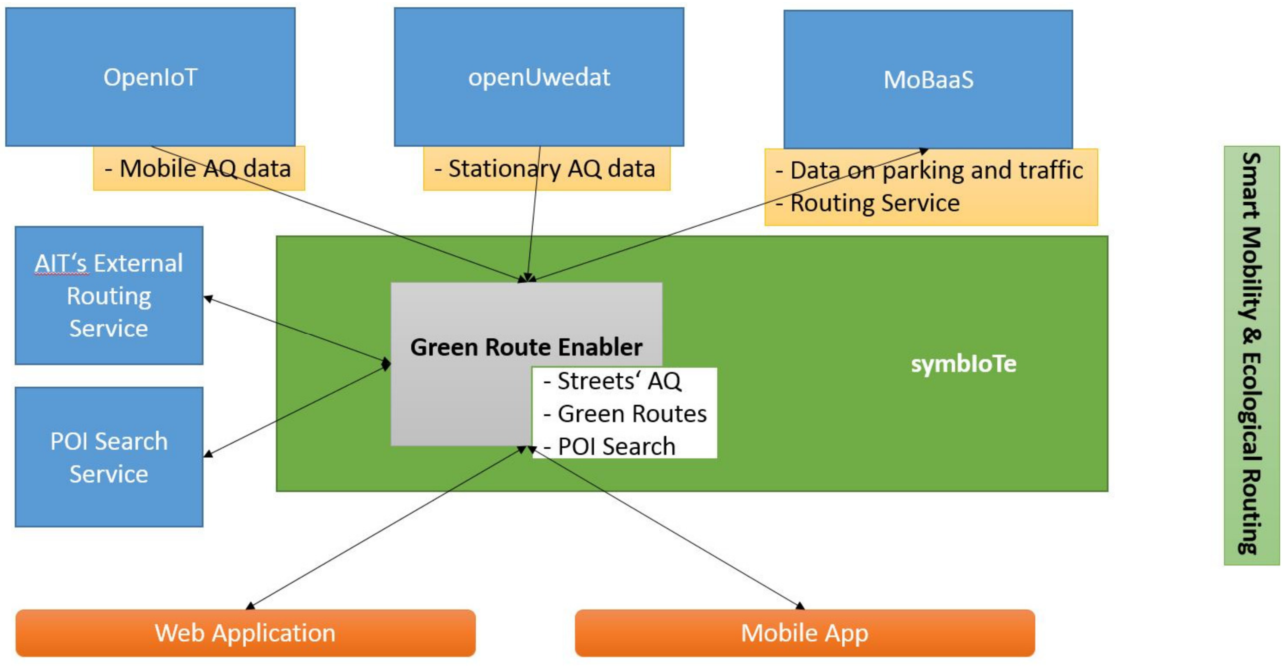


Figure 20: High-level architecture showing the involved platforms, applications and involved symbIoTe components (e.g., enabler) for Smart Mobility and Ecological Routing applications

As can be seen in Figure 20, the Green Route Enabler will orchestrate the activity in the use case. It will obtain air quality data from the platforms and interpolate it with the street segments of the map being used in order to obtain the air quality of a given street. These data will be provided to the routing services (either the ones residing within a platform or external services), which, combining with other data, such as traffic or parking, will compute green routes. Additionally, the data provided by the platforms can also be used to obtain POIs of interest to the users.

It is clear from the figure how symbIoTe is relevant to this use case. It shows how developers, using symbIoTe, can use different data from different platforms from different domains easily. This is very advantageous in the development process, helping developer create complex systems using various sources of data. In the context of smart cities, it will also show how advantageous it is for platform owners and cities to provide their data through the symbIoTe ecosystem, allowing developers/organizations to easily create valuable services to its citizens.

5.2.1.2 Compliance Level

The application will comply with symbIoTe's L1, with the possibility of extending to L2 in the future.

5.2.1.3 Platform

The application will be developed for Android mobile operating system.

5.2.1.4 User Interaction

The user primary means of interacting with the application will be through the map, where it will set, for example, start and end points of his destination or the area near which the user is looking for POIs. It will also be through this map that the user will be able to see the results of the requests, being the route to the destination or the POIs return from the search. See Figure 21 for a mockup of the mobile application.

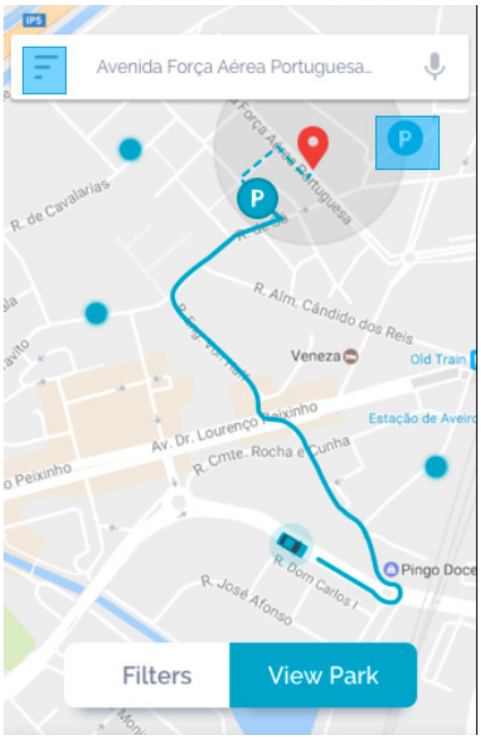


Figure 21: Smart Mobility Mobile App

5.2.1.5 Implementation

The core of the mobile application will be based on the Ionic framework[25], a free and open source mobile SDK. Ionic is, in turn, built on top of AngularJS[26] and Apache Cordova[27].

There is a mock-up of this application developed, although it still does not use the symbIoTe ecosystem and still does not incorporate air quality data. It will be possible to integrate the application with the symbIoTe ecosystem whenever the enabler components are completed.

5.2.1.6 Initial Functional Tests

It is expected that the use case will run trials in at least three European cities. Each trial will have at least 20 users, for a period of 30 days with different types of end-users which will actively use the ecological urban routing application and in parallel will contribute with air quality and traffic data.

5.2.2 Application: Smart Mobility and Ecological Routing Web Application

The web app for the use case will allow users to interact with openUwedat's routing service. On the web page, users will be able to define, on a map, the start and end of their trips. Additionally, they can define the location of their cars or bicycles. Other settings can be defined, such as the type of route (shortest distance, shortest time, best air quality) and the means of transportation. Users can then observe the offered route on the map, consult navigation instructions and check details of the presented route such as distance and duration.

5.2.2.1 Design

The design of the solution can also be observed in Section 5.2.1.1, where either the mobile or the web application use the same back-end system to obtain the information they need. It is possible to observe the versatility of the enabler, i.e., the benefit of symbIoTe, being able to offer cross-domain service to different types of applications.

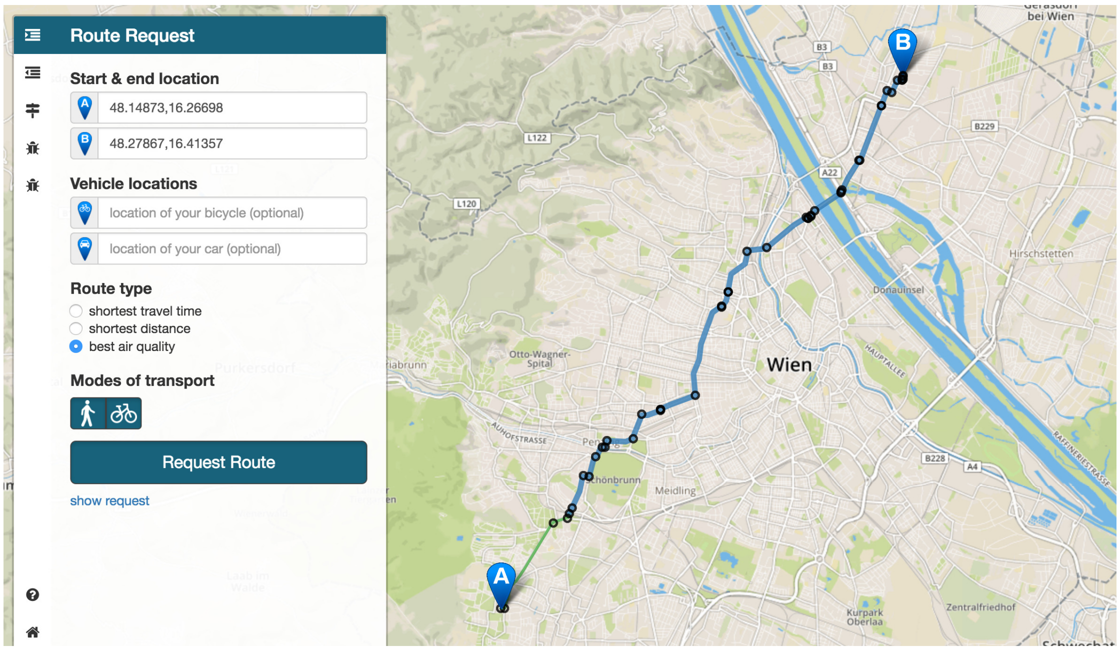
5.2.2.2 Compliance Level

The application will comply with symbIoTe's L1, with the possibility of extending to L2 in the future.

5.2.2.3 Platform

The application will be a web application, usable through a web browser (see Figure 22).

[25] https://ionicframework.com
[26] https://angularjs.org
[27] https://cordova.apache.org



Route Request

Start & end location

A 48.14873,16.26698

B 48.27867,16.41357

Vehicle locations

location of your bicycle (optional)

location of your car (optional)

Route type

shortest travel time [ ]
shortest distance [ ]
best air quality [✓]

Modes of transport

Request Route

show request

Figure 22: Routing Service - Web App

5.2.2.4 *User Interaction*

Similar to the mobile application in Section 5.2.1, the user will interact mostly with a map, where he can define the preferences and visualize the results (see Figure 22). Additionally, there is a side-menu where there are other settings which can be defined and additional information regarding the route that can be consulted.

5.2.2.5 *Implementation*

This use case will gather data from several sources. One of these sources are fixed stations. The data from these stations will be gathered and provided by openUwedat. The other source of data will be mobile sensors which provide their input through openIoT. The implementation status of integration of the underlying IoT platforms that provide raw data (openUwedat and OpenIoT) with the symbIoTe ecosystem is finished and it complies with the R2 version of the core services. Since no additional functionality beyond R2 is needed (and planned) to support the use case we expect only minor additional work to align platform symbIoTe plugins with further releases (R3 and others). Additionally, the use case relies on a specialized enabler that collects data from various sources, processes them and calculates interpolated pollutant values for geographical areas with insufficient physical measurements. The enabler and its sub-components are currently under development and will be kept in sync with further releases of symbIoTe software.

The web application is currently developed. Initial design of the application is already shown in Figure 22.

5.2.2.6 *Initial Functional Tests*

A small test application exists that exercises the functionality of the openUwedat plugin at integration level.

Isolated functional tests will be done at unit test level for parts of the interpolator logic. Further test scenarios that include the complete enabler logic are yet to be defined (as part of task 2.3)

The web application is an existing application that currently only works with metrics like distance and travel time. This means that most relevant parts of the application have already been sufficiently tested.

Final integration testing of the web application are during the trial in Vienna where users will collect air quality measurements and try out suggested routings through the city.

5.3 Use Case Scenario "EduCampus"

The EduCampus use case is inspired by the eduroam (EDUcation ROAMing) initiative, an international roaming services for users in research and education[28]. The key idea behind both concepts is to agree on a common framework to harmonize infrastructure services, in order to provide researchers, teachers and students easy and secure access to campus services when visiting campuses other than their own. While eduroam focuses on network access, the EduCampus use case aims for IoT middleware services.

The vision behind the EduCampus is following. When looking at the rapidly growing market for sensors included in smart devices, used in or attached to smart buildings, establishing smart campus infrastructures, there will be rich offering of services based on IoT middleware installations on a campus. Examples are climate control systems in workplaces, electronic access control systems, indoor location and navigation support, guidance systems for handicapped people, location based collaboration support, or room information and reservations systems as discussed in the EduCampus showcase below.

Sometimes these services will be unique to certain campus, but in many cases there will be very similar services on different campus, but realized in deployment specific ways. This will result in services, which are functionally identical for different campus solutions, but technical incompatible for visiting campus users. In any case there will be multi-platform deployments, consisting of different IoT-domains and also of different IoT-middleware products. By facilitating the symbIoTe interoperability framework for campus deployments, EduCampus aims to be the incubator for interoperable IoT-platform federations.

[28] https://www.eduroam.org/

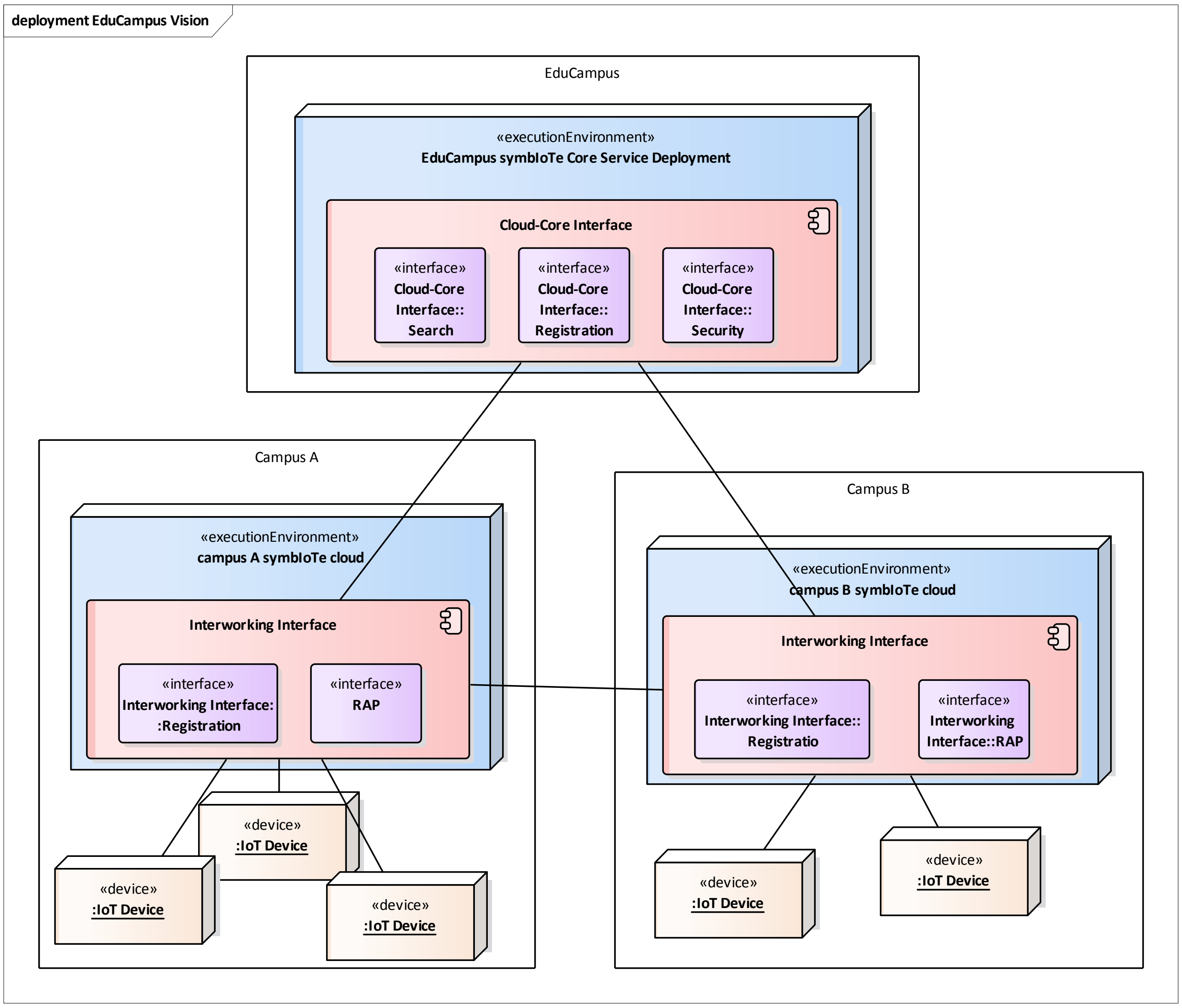

deployment EduCampus Vision
EduCampus
«executionEnvironment»
EduCampus symbIoTe Core Service Deployment
Cloud-Core Interface
«interface» Cloud-Core Interface:: Search
«interface» Cloud-Core Interface:: Registration
«interface» Cloud-Core Interface:: Security
Campus A
«executionEnvironment»
campus A symbIoTe cloud
Interworking Interface
«interface» Interworking Interface: :Registration
«interface» RAP
«device» :IoT Device
«device» :IoT Device
«device» :IoT Device
Campus B
«executionEnvironment»
campus B symbIoTe cloud
Interworking Interface
«interface» Interworking Interface:: Registratio
«interface» Interworking Interface::RAP
«device» :IoT Device
«device» :IoT Device

Figure 23: Conceptual View

5.3.1 Application: Searching for a Room

The <Searching for a Room> application is for a person visiting a campus and searching for specific room. That room might be an office of someone he wants to visit, a working place where he can do some homework, or a meeting room to meet some colleagues.

5.3.1.1 Design

The basic assumption for the EduCampus concept is, that each campus application is designed as in three-tier-architecture. The user will interact with a presentation tier, the application logic will be implemented in a logic tier, and a data tier will manage the data. The only place where interoperability extensions are allowed will be the logic tier. The presentation layer is considered to be the connection between the user and the application and may not be intercepted by any external components. The same is true for the underlying data tier, which is considered to be an internal and protected tier.

When implementing any kind of interoperability between independent campus information management systems, the only place will be the logic tier.

In the “Searching for a Room” application, a user from campus A is visiting campus B (Figure 24). The user is optionally equipped with a location Sensor (BLE beacon). The “Searching for a Room” application is a service within the presentation layer of his campus A information management system. When searching for a room, the application logic from campus A will detect that the user is not located within campus A. It will then use the symbIoTe services to search for and room information service close to the current location of the user.

The campus B information management system will provide access to resource if the appropriate credential can be provided.

The details how the semantic mapping between the symbIoTe client within campus A and the resource access proxy within campus B, is currently being developed within the extended Task T2.1.

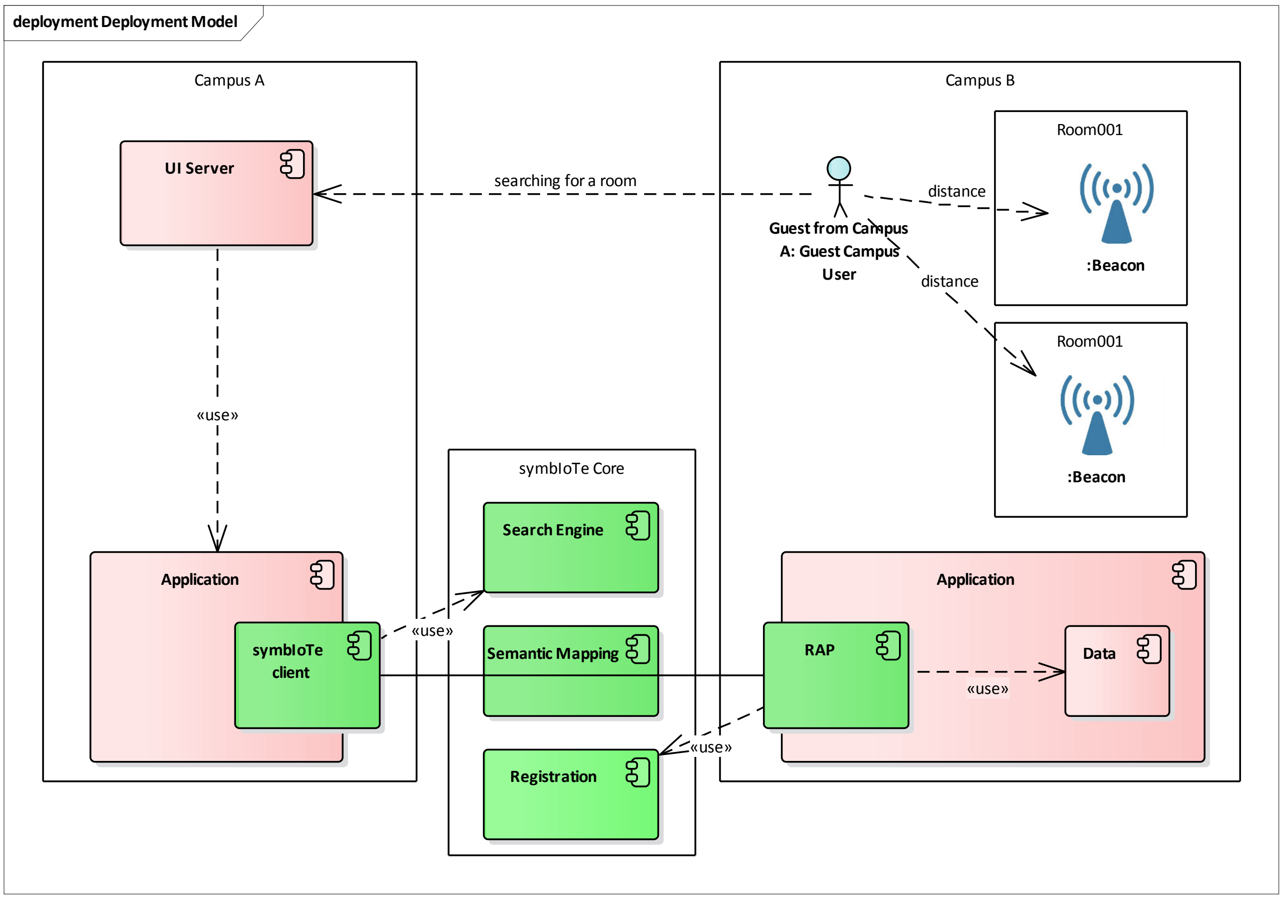


Figure 24: Deployment Concept

5.3.1.2 Compliance Level

The primary compliance level of symbIoTe integration will be L2.

5.3.1.3 Platform

Two IoT platforms will be included, namely Campus A and Campus B, as well as the accompanying development platforms, having the following specifications:

Campus A (Fraunhofer IOSB):

- Android SmartPhone with BLE technology for location sensing
- WebGenesis (Content Management System including a SensorThingsAPI Dashboard) as Presentation layer
- Java Server application for logic layer
- Outlook and Sensor Things Server for the data layer

Campus B (Karlsruher Institut für Technologie, KIT):

- Android SmartPhone with BLE technology for location sensing
- KIT Smart Campus presentation and logic layer
- KIT MORADA room information data base

5.3.1.4 User Interaction

The initial design of the user interaction has been shown in the document D1.2. At the current stage of the development no further details on user interactions are known. Figure 25 and Figure 26 show the current design.

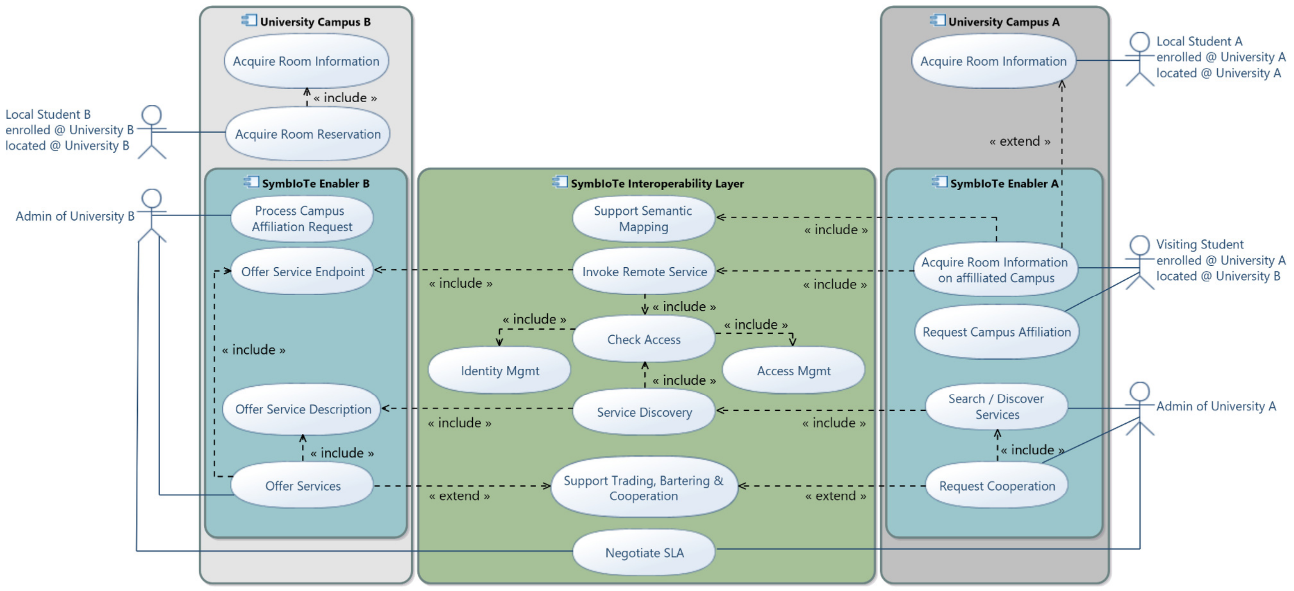


Figure 25: EduCampus User Interaction

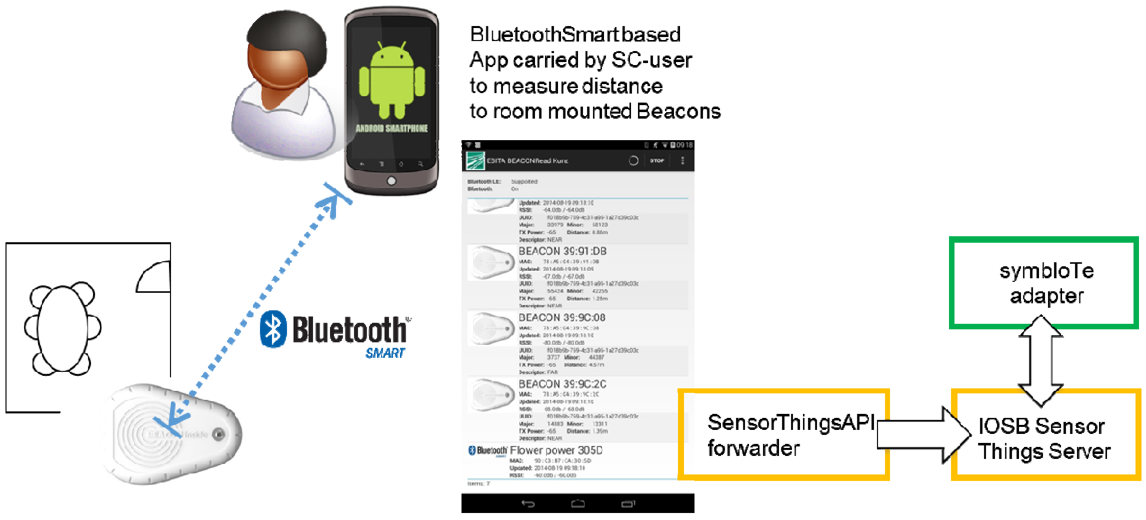
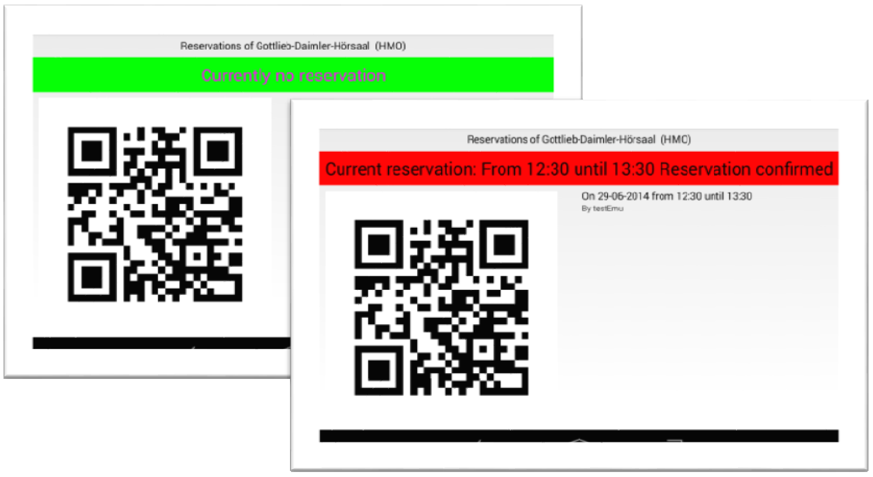


Figure 26: Bluetooth Low Energy based proximity measurement

The design for the EduCampus user interface has not yet started. It will be similar to an earlier design shown in the figure below, which is a status display for room information (see Figure 27). A mobile application will be available for navigation, room information and reservation. A prototype is shown in Figure 28.


Reservations of Gottlieb-Daimler-Hörsaal (HMO)

Currently no reservation

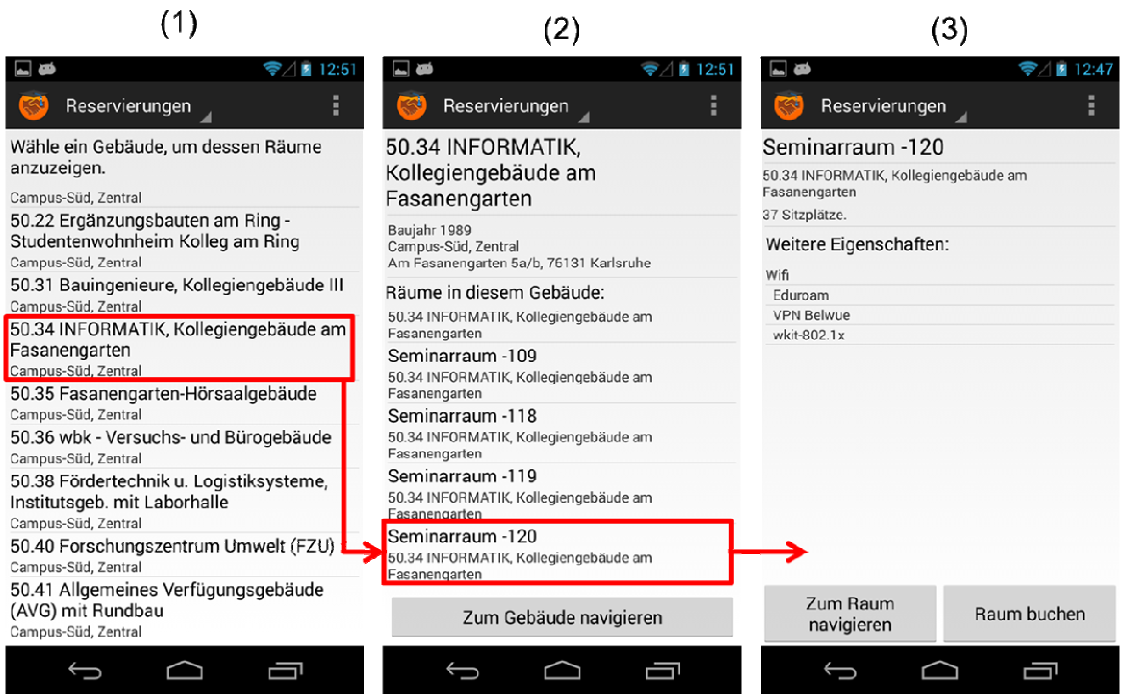
Reservations of Gottlieb-Daimler-Hörsaal (HMO)

Current reservation: From 12:30 until 13:30 Reservation confirmed

On 29-06-2014 from 12:30 until 13:30

By testEmu

Figure 27: Room Information Board for reservation status



(1)

Wähle ein Gebäude, um dessen Räume anzuzeigen.
Campus-Süd, Zentral
50.22 Ergänzungsbauten am Ring - Studentenwohnheim Kolleg am Ring
Campus-Süd, Zentral
50.31 Bauingenieure, Kollegiengebäude III
Campus-Süd, Zentral
50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
Campus-Süd, Zentral
50.35 Fasanengarten-Hörsaalgebäude
Campus-Süd, Zentral
50.36 wbk - Versuchs- und Bürogebäude
Campus-Süd, Zentral
50.38 Fördertechnik u. Logistiksysteme, Institutsgeb. mit Laborhalle
Campus-Süd, Zentral
50.40 Forschungszentrum Umwelt (FZU)
Campus-Süd, Zentral
50.41 Allgemeines Verfügungsgebäude (AVG) mit Rundbau
Campus-Süd, Zentral

(2)

50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
Baujahr 1989
Campus-Süd, Zentral
Am Fasanengarten 5a/b, 76131 Karlsruhe
Räume in diesem Gebäude:
50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
Seminarraum -109
50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
Seminarraum -118
50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
Seminarraum -119
50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
Seminarraum -120
50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
Zum Gebäude navigieren

(3)

Seminarraum -120
50.34 INFORMATIK, Kollegiengebäude am Fasanengarten
37 Sitzplätze.
Weitere Eigenschaften:
Wifi
Eduroam
VPN Belwue
wkit-802.1x
Zum Raum navigieren
Raum buchen

Figure 28: Building and Room Information

5.3.1.5 *Implementation*

The implementation languages for the EduCampus Use Case will be Java and JavaScript.

Location information will be measured by using BLE technology with Bluetooth beacons attached to certain locations.

5.3.1.6 *Initial Functional Tests*

The functional test will be based on the installation within the campus of Fraunhofer IOSB in Karlsruhe, Germany. Several rooms will be equipped with Bluetooth beacons and a room information and reservation service will be installed. Several employees will be equipped with mobile clients to test the services.

Parallel to the IOSB installation there will deployment within the University Of Karlsruhe (KIT) with similar features. The detailed planning of initial test is expected for the end of 2017.

5.4 *Use Case Scenario “Smart Stadium”*

Smart Stadium enhances the user experience of visitors coming to a stadium. In the retail context, it provides that both visitors and retailers get closer even in large distances across the stadium.

Although being an IoT project, Smart Stadium does not involve sensors and actuators, which is usually the first thing that comes to one’s mind when mentioning IoT devices. In this use case, however, smartphones, tablets and smart TVs are the IoT players.

Different IoT platforms can live together in the stadium, offering access to their devices to all other platforms and client applications through symbIoTe. Visitors are identified by their smartphones, while retailers (both moving carts and physical shops) are identified by their TPV and beacons. From the visitor's point of view, Smart Stadium brings the opportunity for detecting closest retailers, place orders independent of where they are, for receiving products they bought directly in their seat.

On the other hand, retailers can broadcast their offers and promotions to all visitors inside the stadium, or those that are moving near specific areas inside the stadium. Retailers can send their promotions to large SmartTVs (Promowalls) strategically placed throughout the stadium.

5.4.1 Application: Visitor application

The visitor application, with no commercial name yet, provides visitors in the stadium access to all retailer information as well as an entry point of news and promotions. As soon as the user arrives to the stadium, the app registers user's location based on the proximity to beacons. From now on, the user is discoverable and accessible thanks to this application and its backend.

5.4.1.1 Design

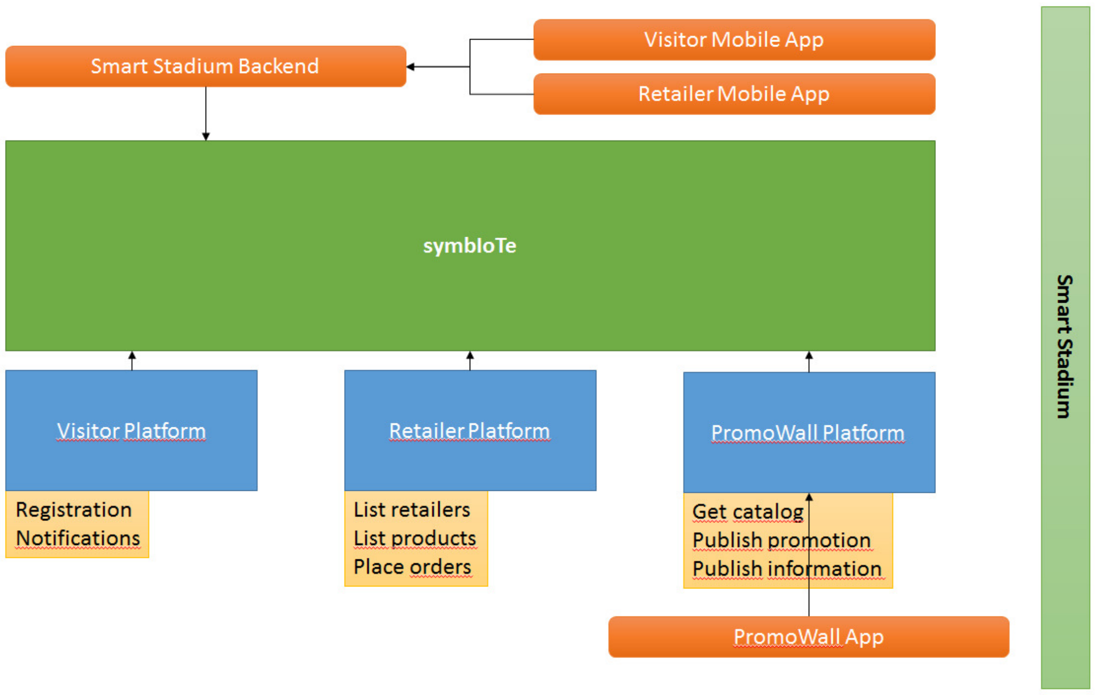


Figure 29: Components and interactions for all smart stadium use cases

The main benefit symbIoTe provides to us is the discoverability for new incoming devices to the stadium as well as the continuous status updates. It also standardizes the communication process by defining all information models, and rules to add custom ones.

The application is divided into a regular client and server architecture (Figure 29), where the server is the one that knows and interacts with symbIoTe. This way mobile app capabilities can grow with no deep details on where those services come from (new platforms, a new enabler merging data from different platforms, etc.). The backend does not yet exist, and it will be developed in the context of this use case. Visitor platform will provide access to known services registered in symbIoTe. The platform will act as a facade. All IoT devices are localized using a custom symbolic location based on proximity to beacons spread throughout the stadium. All beacons emit unique identifiers that can be used to locate IoT devices inside the stadium.

For example, a mobile app running on a Bluetooth low energy (BLE) capable device detects the following known beacons:

- Beacon 1 tagged as 'door 14' at a distance of 1 meter (near)
- Beacon 2 tagged as 'corridor 3' at a distance of 15 meters (far)
- Beacon 3 tagged as 'floor 1', at a distance of 1 meter (near)

The device is then located at the symbolic location "near door 14, near floor 1, far corridor 3". Physical shops can be easily located by using a specific beacon at the entrance door.

The following diagram (Figure 30) depicts the device registration performed by both visitor and retailer applications as well as all components involved in the Smart Stadium use case.

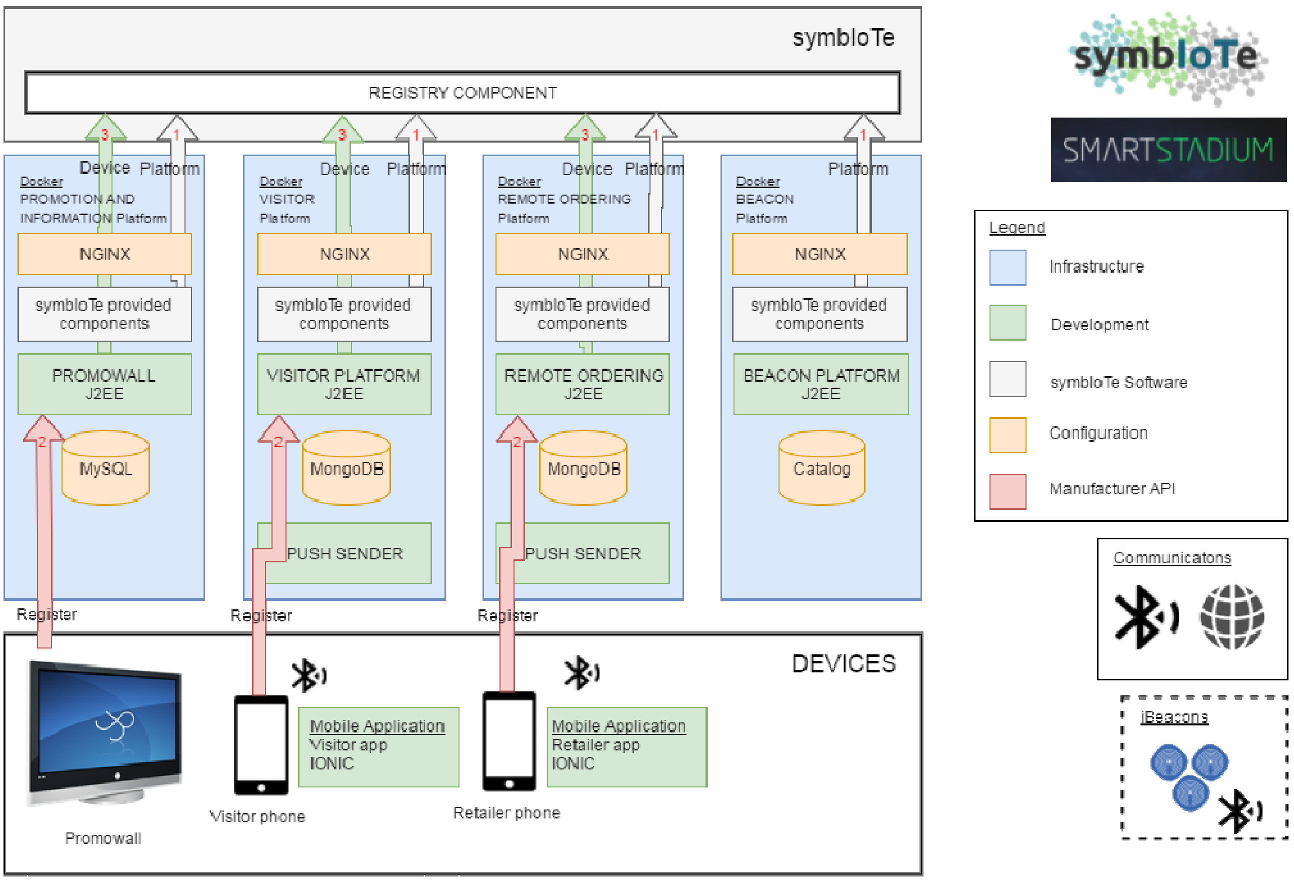


Figure 30: Device Registration

5.4.1.2 *Compliance Level*

The primary compliance level of symbIoTe integration will be L1.

5.4.1.3 Platform

The visitor application solution will be implemented using the following technology stacks:

- Mobile app: hybrid application developed using Cordova as a native envelope and Ionic and AngularJS as application core.
- App will run on Android devices with BLE capability to locate near beacons.
- Backend: J2EE stateless RESTful services implemented using Spring Framework and MongoDB.

5.4.1.4 User interaction

The visitor application brings to people arriving to the stadium all information and services they need to get updated of sport events and products they might be interested in, for example, sport stuff, food and drinks. The other great functionality the app provides to the user is passive to the user: receiving incoming data from devices/people with granted access to symbIoTe and Visitor Platform related devices: push notifications. Figure 31 depicts the registration process of the visitor device once he/she enters the stadium.

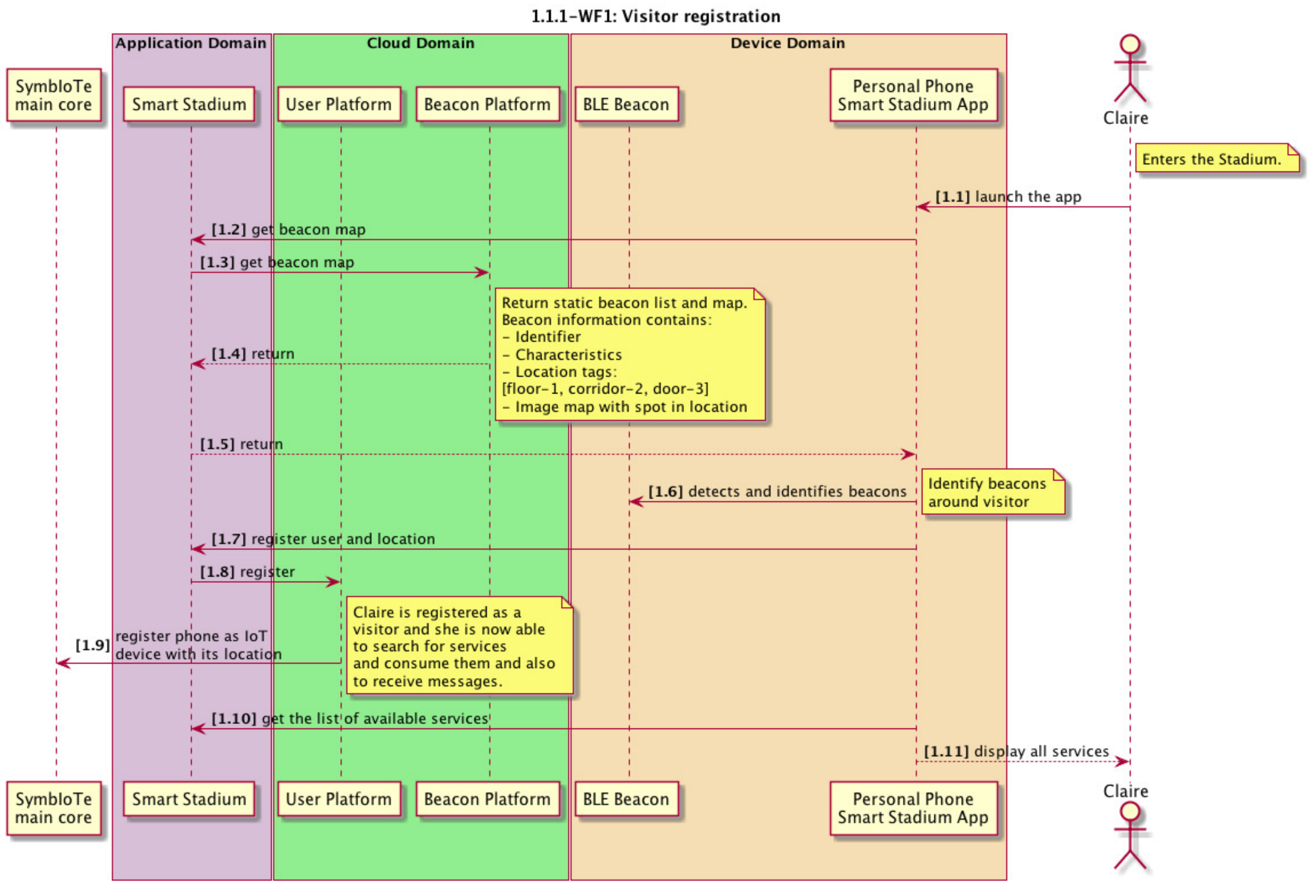

1.1.1-WF1: Visitor registration
Application Domain
Cloud Domain
Device Domain
SymbIoTe main core
Smart Stadium
User Platform
Beacon Platform
BLE Beacon
Personal Phone Smart Stadium App
Claire
Enters the Stadium.
[1.1] launch the app
[1.2] get beacon map
[1.3] get beacon map
Return static beacon list and map.
Beacon information contains:
- Identifier
- Characteristics
- Location tags:
[floor-1, corridor-2, door-3]
- Image map with spot in location
[1.4] return
[1.5] return
[1.6] detects and identifies beacons
Identify beacons around visitor
[1.7] register user and location
[1.8] register
Claire is registered as a visitor and she is now able to search for services and consume them and also to receive messages.
[1.9] register phone as IoT device with its location
[1.10] get the list of available services
[1.11] display all services

Figure 31: Workflow “Visitor registration”

Figure 32 describes the workflow being performed when the visitor buys something via the visitor app. In a few words, visitor app looks for shops near the visitor and displays the products they sell; visitor prepares an order with some products (and maybe coupons and discounts) and sends it to the retailer, waiting for confirmation.

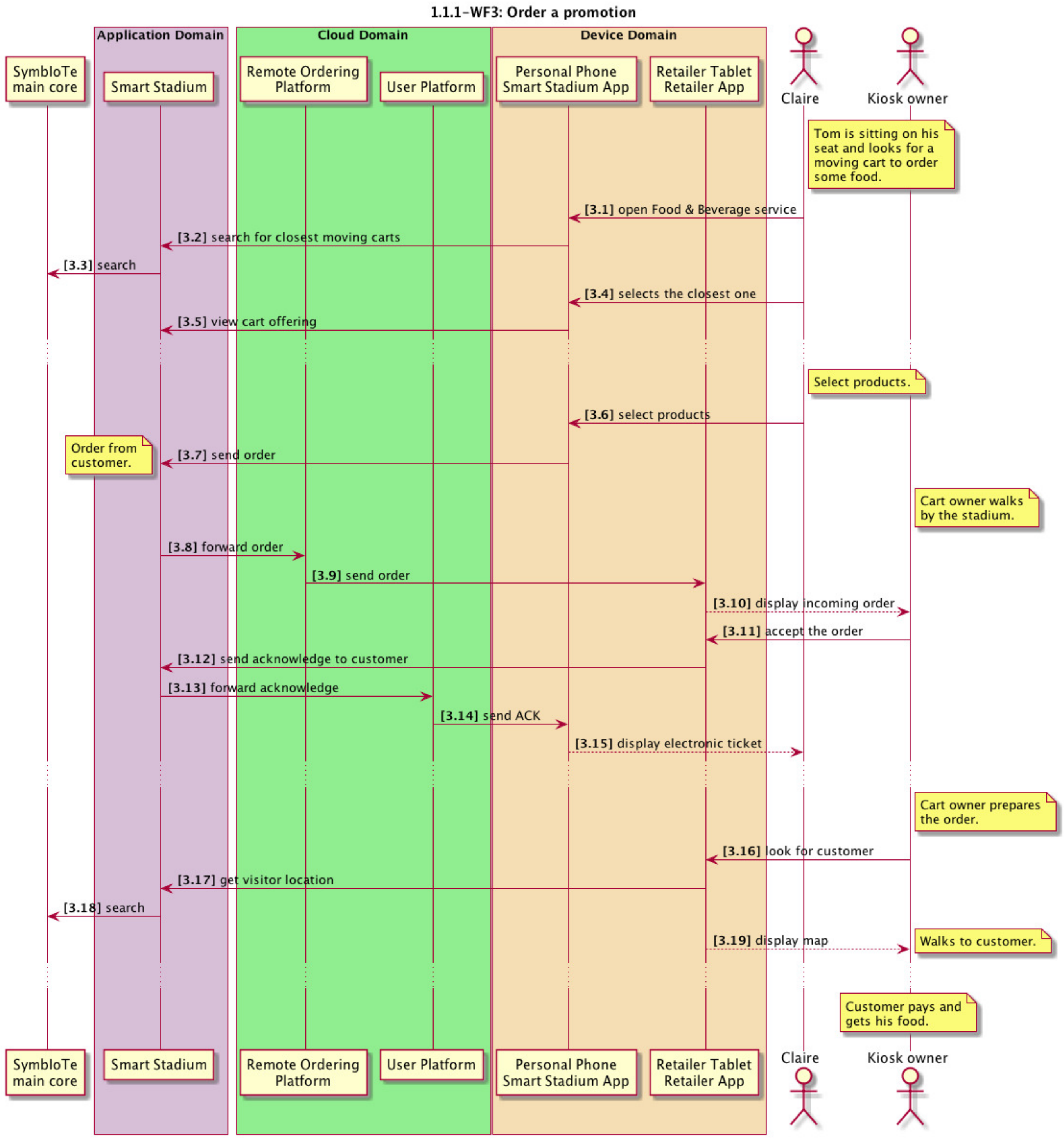

1.1.1-WF3: Order a promotion
Application Domain
Cloud Domain
Device Domain
SymbIoTe main core
Smart Stadium
Remote Ordering Platform
User Platform
Personal Phone Smart Stadium App
Retailer Tablet Retailer App
Claire
Kiosk owner
Tom is sitting on his seat and looks for a moving cart to order some food.
[3.1] open Food & Beverage service
[3.2] search for closest moving carts
[3.3] search
[3.4] selects the closest one
[3.5] view cart offering
Select products.
[3.6] select products
Order from customer.
[3.7] send order
Cart owner walks by the stadium.
[3.8] forward order
[3.9] send order
[3.10] display incoming order
[3.11] accept the order
[3.12] send acknowledge to customer
[3.13] forward acknowledge
[3.14] send ACK
[3.15] display electronic ticket
Cart owner prepares the order.
[3.16] look for customer
[3.17] get visitor location
[3.18] search
[3.19] display map
Walks to customer.
Customer pays and gets his food.

Figure 32: “Order a Promotion” workflow

Finally, the sequence diagram shown in Figure 33 depicts the arrival of notifications coming from a symbIoTe client to the mobile application.

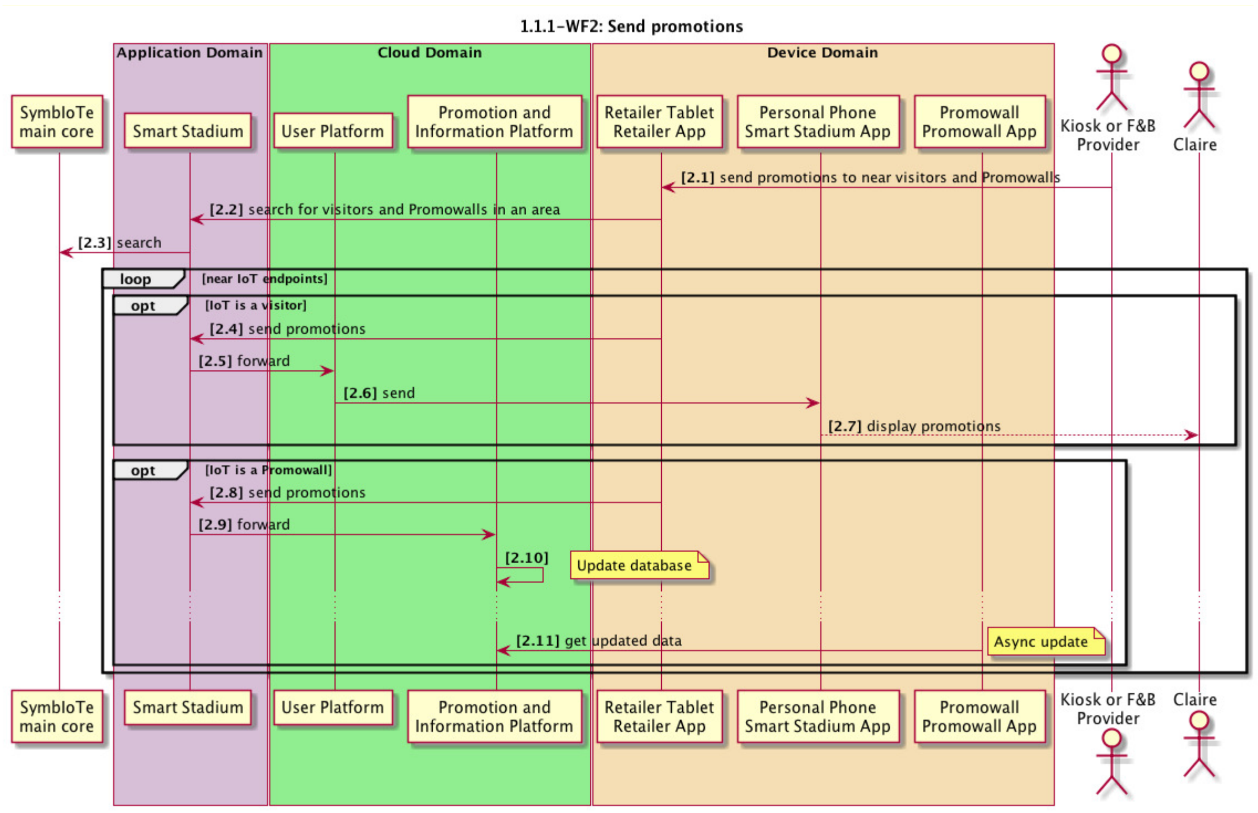


Figure 33: Workflow “Sending promotions to both PromoWalls and visitors”

The communication between the Visitor platform and the mobile app is intended to be implemented via push messages.

Application visuals are already available, as shown in Figure 34.

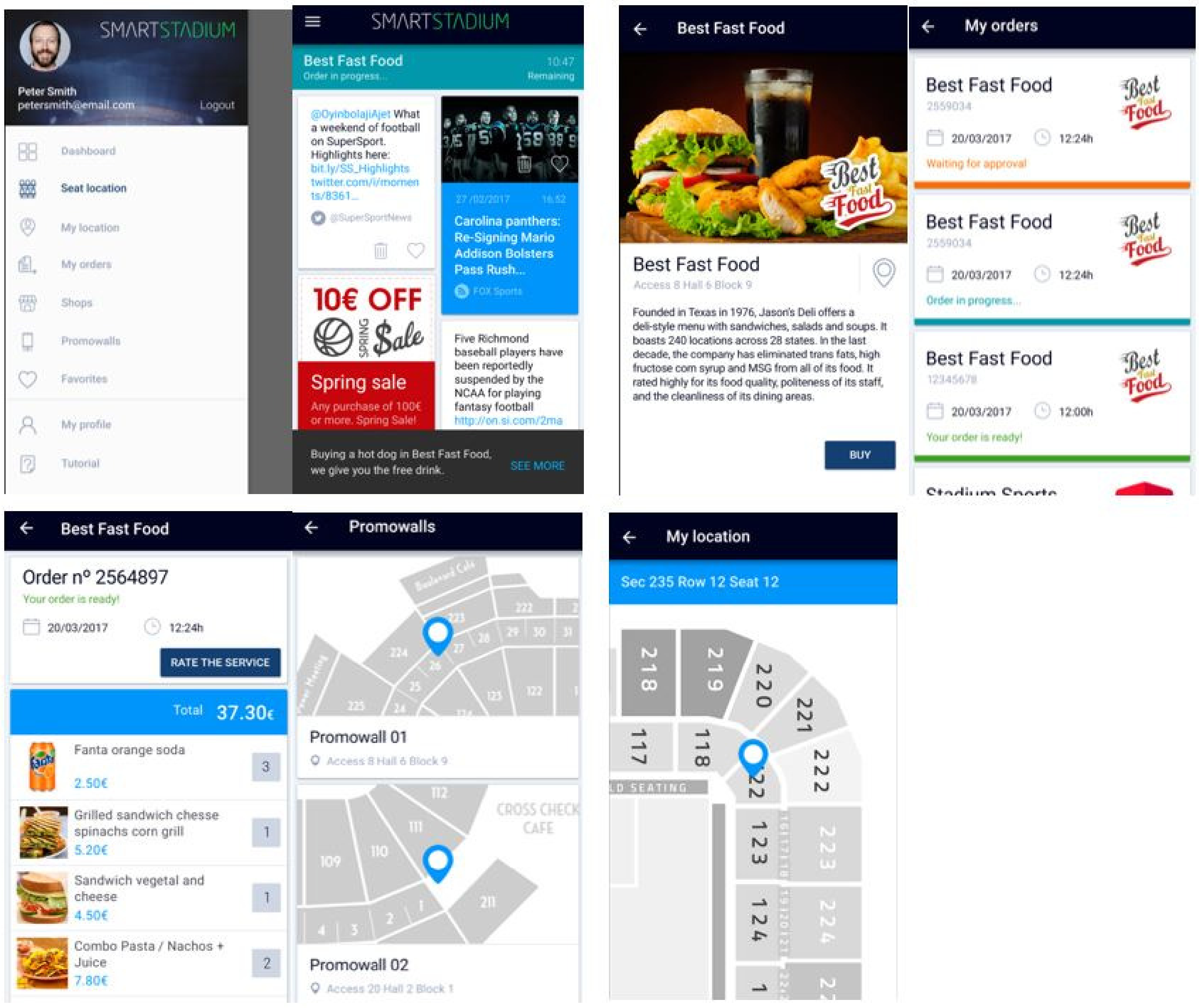

SMARTSTADIUM
Peter Smith
petersmith@email.com
Logout
Dashboard
Seat location
My location
My orders
Shops
Promowalls
Favorites
My profile
Tutorial
Best Fast Food
Order in progress...
10:47
Remaining
10€ OFF
Spring Sale
Spring sale
Any purchase of 100€ or more. Spring Sale!
Carolina panthers: Re-Signing Mario Addison Bolsters Pass Rush...
Buying a hot dog in Best Fast Food, we give you the free drink.
SEE MORE
Best Fast Food
Access 8 Hall 6 Block 9
Founded in Texas in 1976, Jason's Deli offers a deli-style menu with sandwiches, salads and soups. It boasts 240 locations across 28 states. In the last decade, the company has eliminated trans fats, high fructose corn syrup and MSG from all of its food. It rated highly for its food quality, politeness of its staff, and the cleanliness of its dining areas.
BUY
My orders
Best Fast Food
2559034
20/03/2017
12:24h
Waiting for approval
Order in progress...
Your order is ready!
Stadium Sports
Order nº 2564897
Your order is ready!
RATE THE SERVICE
Total 37.30€
Fanta orange soda 2.50€ 3
Grilled sandwich chesse spinachs corn grill 5.20€ 1
Sandwich vegetal and cheese 4.50€ 1
Combo Pasta / Nachos + Juice 7.80€ 2
Promowalls
Promowall 01
Access 8 Hall 6 Block 9
Promowall 02
Access 20 Hall 2 Block 1
CROSS CHECK CAFE
My location
Sec 235 Row 12 Seat 12

Figure 34: Visuals for the Smart Stadium use case Visitor scenario application

5.4.1.5 Implementation

As mention above, there are two pieces of software involved in this application which are implemented in the following way:

- Backend: J2EE application implemented using the following frameworks and tools
    - Spring Framework: core framework
    - Apache Camel: to implement relevant processes
    - Spring Data: data layer abstraction
    - MongoDB: NoSQL database
    - Apache CXF: implement RESTful services
    - CAS: security for REST API
- Mobile application: hybrid application

- Cordova: envelop providing access to native capabilities as well as platform specific application stores
- Ionic and AngularJS: core framework to develop application logic and UI
- Flux: data flow pattern for large applications

In order to guarantee code quality, the backend will be analyzed using SonarQube[29] and JaCoCo[30] (code coverage), and the RESTful API will be tested using Postman. On the other hand, mobile application will be tested using functional test cases.

Moreover, this use case required a modification of the Core Information Model to add a generic IoT device, as until then it was only covering sensors and actuators.

Currently mobile application is being implemented, in and advanced stage but disconnected from server. On the other hand, backend just started its implementation based on R2 symbIoTe release. It's currently on an early stage with all components being connected.

5.4.1.6 Initial Functional Tests

Test cases are not yet defined at this stage of the pilot definition.

5.4.2 Application: Retailer Application

The retailer application, with no commercial name yet but intended to be an application per retailer (custom UI), provides retailers the opportunity to publish their services and products to anyone in the stadium. This application let retailers manage the order inbox and all the orders being processed and delivered. It also lets the seller to emit specific discounts and coupons to either visitors' devices or Promowalls located in certain sections of the stadium.

5.4.2.1 Design

The main benefit of using symbIoTe is the device discoverability via the Search registry to access all kind of symbIoTe enabled platforms that are currently defined but much more that could raise in the future.

The application is divided into a regular client and server architecture (Figure 29), where the server is the one that knows and interacts with symbIoTe. This way client app capabilities can grow with no deep details on where those services come from (new platforms, a new enabler merging data from different platforms, etc.). The backend does not yet exist and it will be developed in the context of this use case. Remote Ordering platform will provide access to known services registered in symbIoTe, related to the Smart Stadium use case. The platform will act as a facade. All IoT devices are localized using a custom symbolic location based on proximity to beacons spread throughout the stadium.

[29] https://www.sonarqube.org/
[30] http://www.eclemma.org/jacoco/trunk/index.html

All beacons emit unique identifiers that can be used to locate IoT devices inside the stadium, as is described in Section 5.4.1.

5.4.2.2 Compliance Level

The primary compliance level of symbIoTe integration will be L1.

5.4.2.3 Platform

The retailer application solution will be implemented using the following technology stacks:

- Desktop app: web application developed using Electron as a native envelope and Ionic and AngularJS as application core.
- App will run on a RaspberryPi device with a plugged display.
- Backend: J2EE stateless RESTful services implemented using Spring Framework and MongoDB.

5.4.2.4 User interaction

The retailer application allows sellers to make their products and services accessible to anyone in the stadium, as well as the ability to send messages and offers to them.

The sequence diagram in Figure 35 depicts the process of sending promotions to visitors located in specific areas of the stadium.

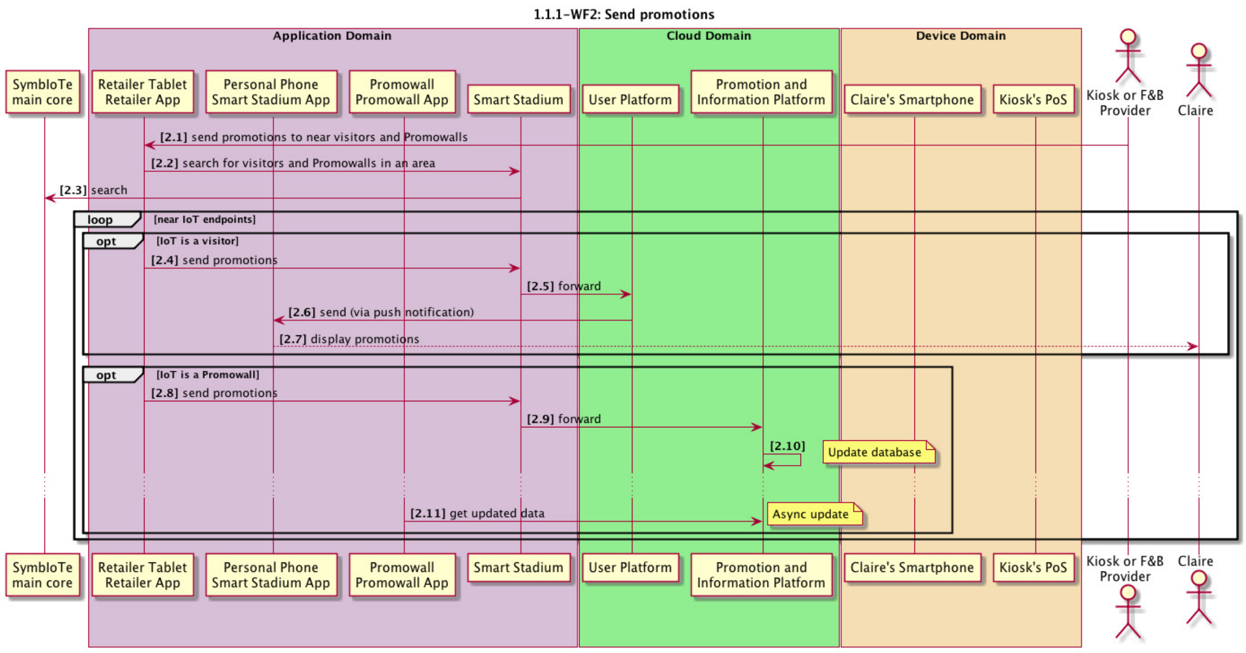


Figure 35: Workflow “Sending promotions to both PromoWalls and visitors”

The sequence diagram in Figure 36 shows the purchasing process from the retailer’s perspective: the reception of the purchase order and the acceptancy or rejection by the retailer.

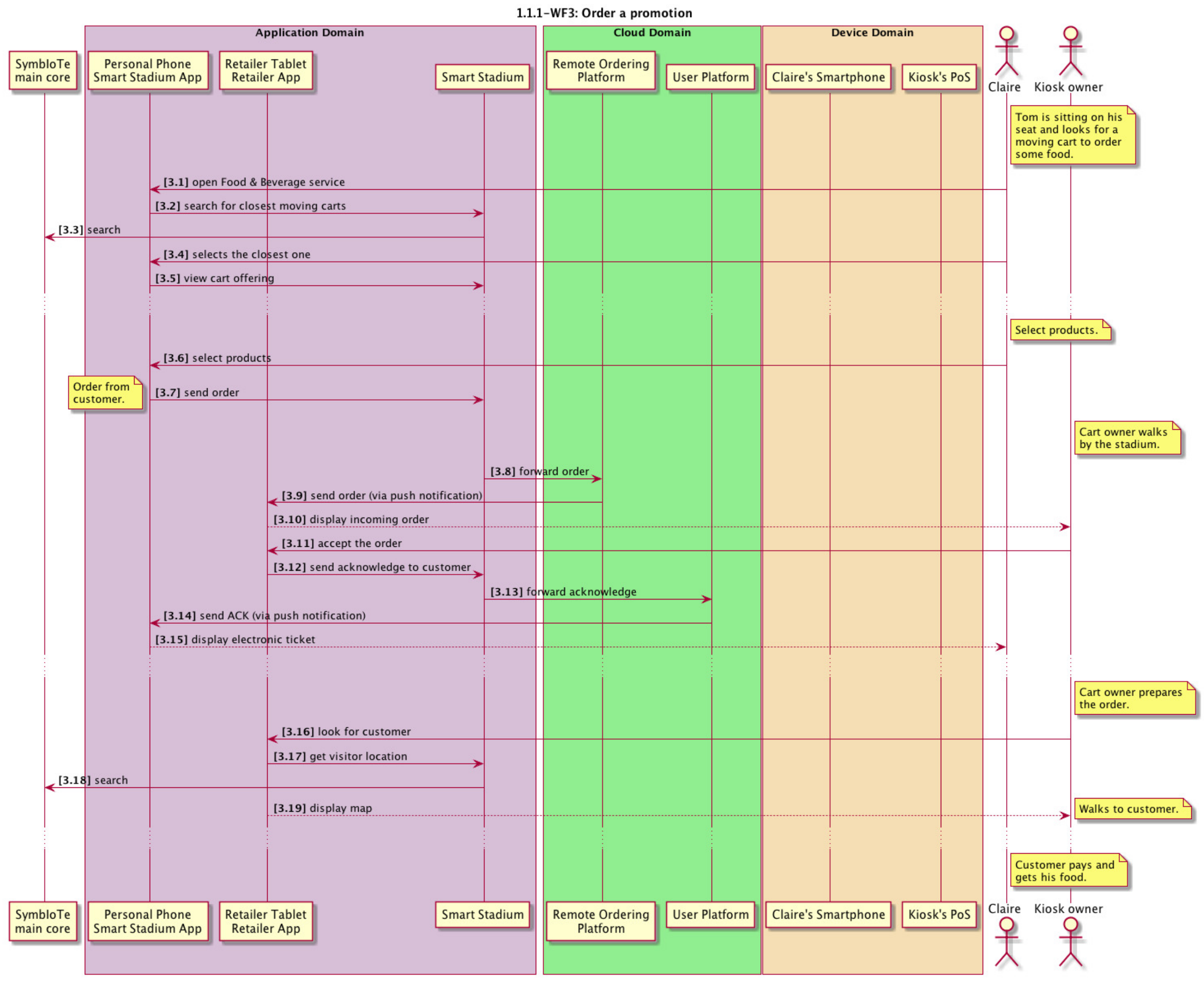

1.1.1-WF3: Order a promotion
Application Domain
Cloud Domain
Device Domain
SymbIoTe main core
Personal Phone Smart Stadium App
Retailer Tablet Retailer App
Smart Stadium
Remote Ordering Platform
User Platform
Claire's Smartphone
Kiosk's PoS
Claire
Kiosk owner
Tom is sitting on his seat and looks for a moving cart to order some food.
[3.1] open Food & Beverage service
[3.2] search for closest moving carts
[3.3] search
[3.4] selects the closest one
[3.5] view cart offering
Select products.
[3.6] select products
Order from customer.
[3.7] send order
Cart owner walks by the stadium.
[3.8] forward order
[3.9] send order (via push notification)
[3.10] display incoming order
[3.11] accept the order
[3.12] send acknowledge to customer
[3.13] forward acknowledge
[3.14] send ACK (via push notification)
[3.15] display electronic ticket
Cart owner prepares the order.
[3.16] look for customer
[3.17] get visitor location
[3.18] search
[3.19] display map
Walks to customer.
Customer pays and gets his food.

Figure 36: Workflow "Receive and acknowledge order"

The communication between the Remote Ordering Platform and the desktop app is intended to be implemented either via push messages (if working on Electron), or polling, as TPV app is intended to be always in foreground. Application visuals are already available with examples in Figure 37.

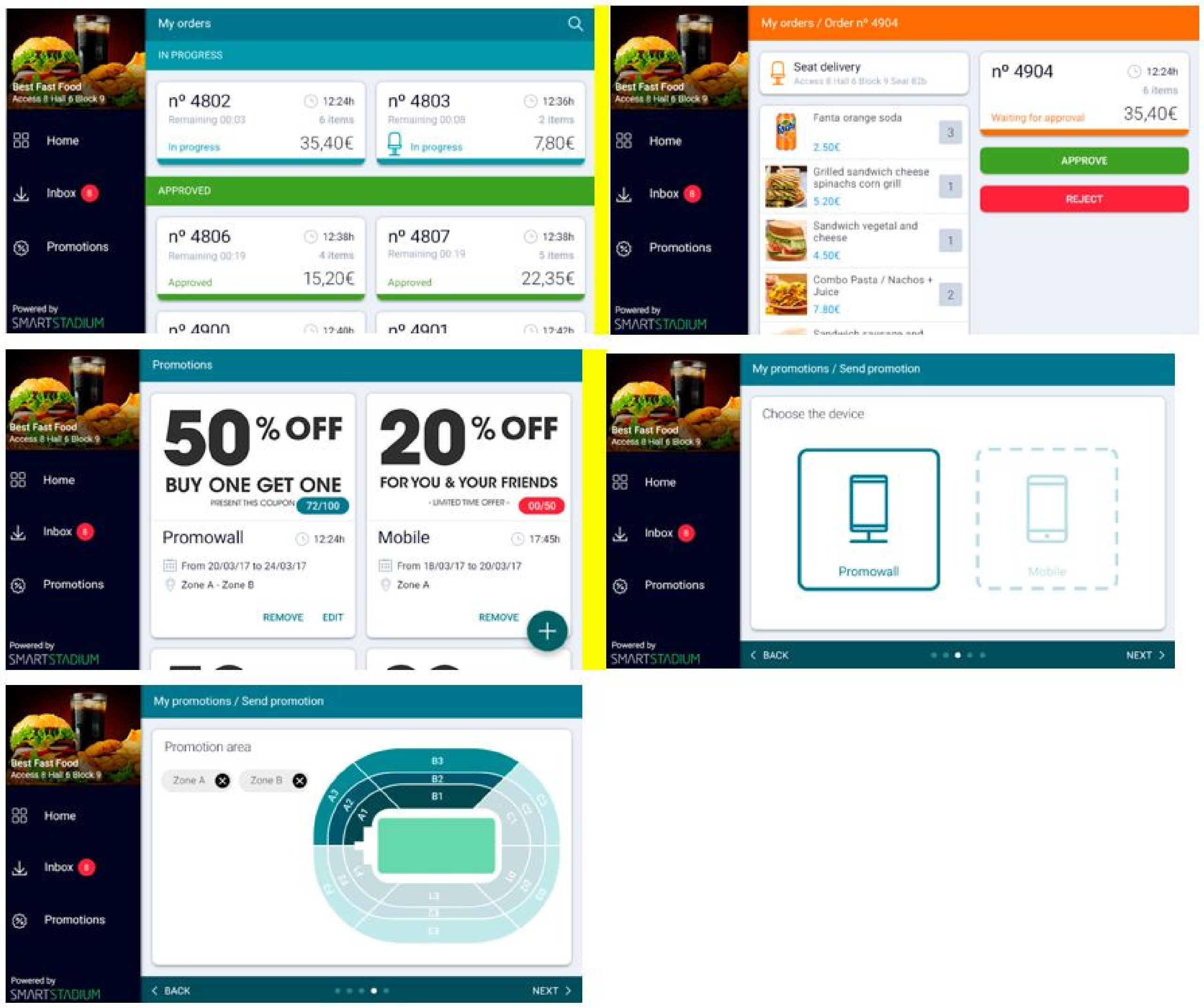


Figure 37: Visuals for the Smart Stadium use case Retailer scenario application

5.4.2.5 Implementation

As mention above, there are two pieces of software involved in this application that are implemented in the following way:

- Backend: J2EE application implemented using the following frameworks and tools
  - Spring Framework: core framework
  - Apache Camel: to implement relevant processes
  - Spring Data: data layer abstraction
  - MongoDB: NoSQL database
  - Apache CXF: implement RESTful services
  - CAS: security for REST API
- Mobile application: desktop application developed using HTML5

- Electron: envelop providing access to OS capabilities as well as platform specific installer and runtime
- Ionic and AngularJS: core framework to develop application logic and UI
- Flux: data flow pattern for large applications

In order to guarantee code quality, the backend will be analyzed using SonarQube and JaCoCo (code coverage), and the RESTful API will be tested using Postman. On the other hand, mobile application will be tested using functional test cases. Moreover, this use case required a modification of the Core Information Model to add a generic IoT device, as until then it was only covering sensors and actuators.

Currently mobile application is being implemented, in and advanced stage but disconnected from server. On the other hand, backend just started its implementation based on R2 symbIoTe release. It's currently on an early stage with all components being connected.

5.4.2.6 Initial Functional Tests

Test cases are not yet defined at this stage of the pilot definition.

5.4.3 Application: Promowall

Promowall is an existing solution from Worldline that offers the ability to publish stylish promotions and limited coupons to customers in two different channels: Promowall mobile app and large touch-screen Smart TVs, the *Promowalls*.

5.4.3.1 Design

The main benefit of using symbIoTe is the broadcasting of the Promowall published information thanks to the discoverability of new devices and the ability to introduce new symbIoTe enabled platforms that could use Promowall.

The Promowall backoffice application was implemented as a monolithic piece of software containing a RESTful API, used by the mobile app, and a backoffice GUI implemented using ZK Framework (server-side rendering).

The Promowall solution is divided into three applications (see Figure 29):

- Promowall backoffice (out of Smart Stadium use case, because it's replaced by the Retailer application described above).
- Frontend HTML5 application running on Promowall devices displaying promotions, relevant information and the QR code to activate promotions using any QR scanner application from their devices (Promowall mobile app is not even required).
- Promowall mobile application for final users.

5.4.3.2 Compliance Level

The primary compliance level of symbIoTe integration will be L1.

5.4.3.3 Platform

The Promotion and Information platform is nothing else than the symbIoTe enabled Promowall backend.

The retailer application solution is implemented using the following technology stacks:

- RESTful API: J2EE web application using Spring Framework.
- SmartTV frontend: HTML5 application
- Mobile application: Android native application running on a BLE capable device

5.4.3.4 User interaction

The Promowall solution relevant for Smart Stadium use case involves the interaction of visitors with Promowalls and smartphones running the Promowall app. The interaction is depicted in Figure 38, while the real screenshots of the Android mobile and Web application of Promowall are provided in Figure 39 and Figure 40, respectively.

The interaction is depicted in the following storyboard (also notice embedded comments).

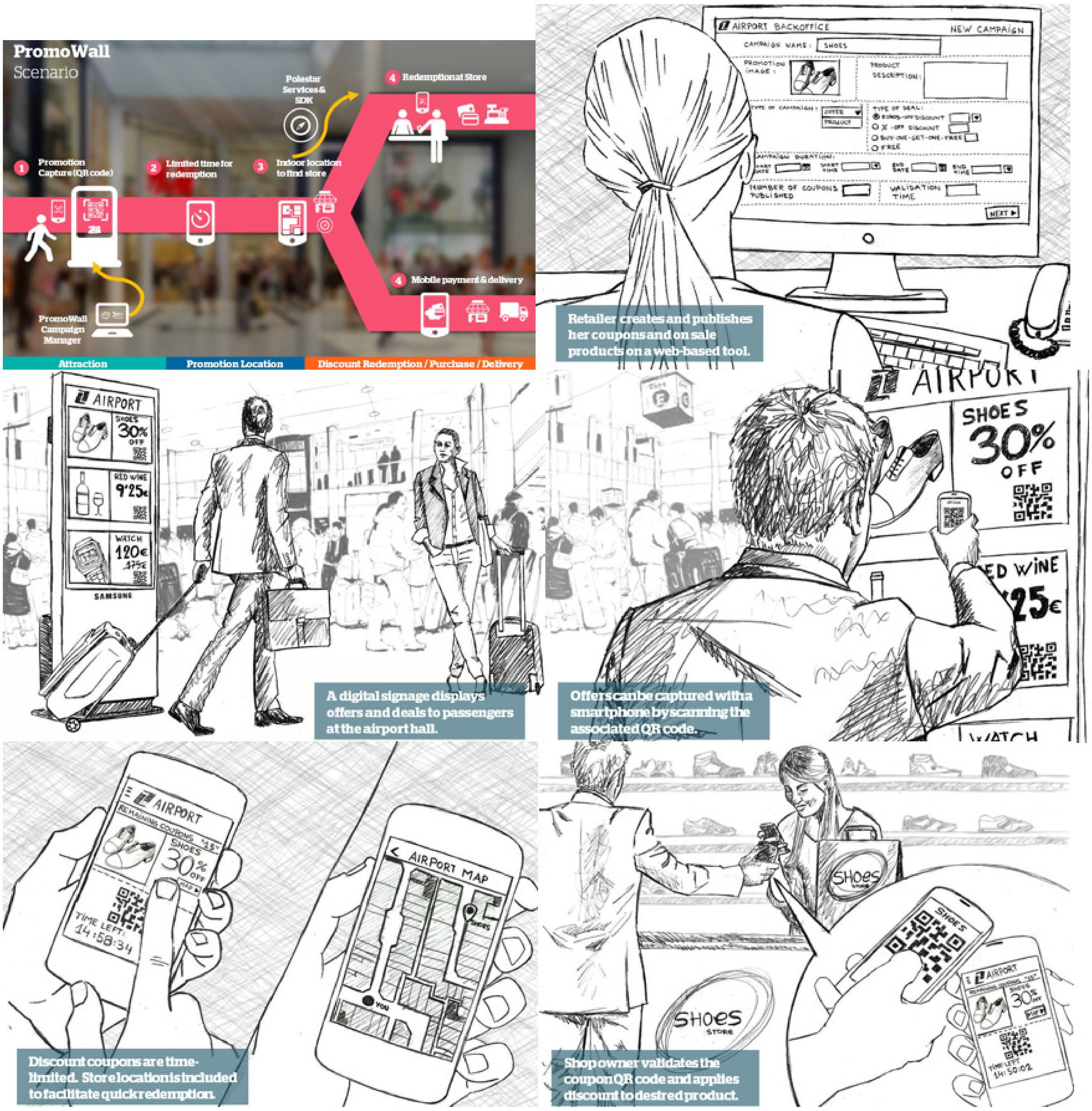

PromoWall
Scenario
1 Promotion Capture (QR code)
2 Limited time for redemption
3 Indoor location to find store
Polestar Services & SDK
4 Redemptional Store
4 Mobile payment & delivery
PromoWall Campaign Manager
Attraction
Promotion Location
Discount Redemption / Purchase / Delivery
Retailer creates and publishes her coupons and on sale products on a web-based tool.
A digital signage displays offers and deals to passengers at the airport hall.
Offers can be captured with a smartphone by scanning the associated QR code.
Discount coupons are time-limited. Store location is included to facilitate quick redemption.
Shop owner validates the coupon QR code and applies discount to desired product.
AIRPORT BACKOFFICE
NEW CAMPAIGN
AIRPORT
SHOES 30% OFF
RED WINE 9'25€
WATCH 120€
AIRPORT MAP
TIME LEFT: 14:58:34
SHOES

Figure 38: Promowall interaction

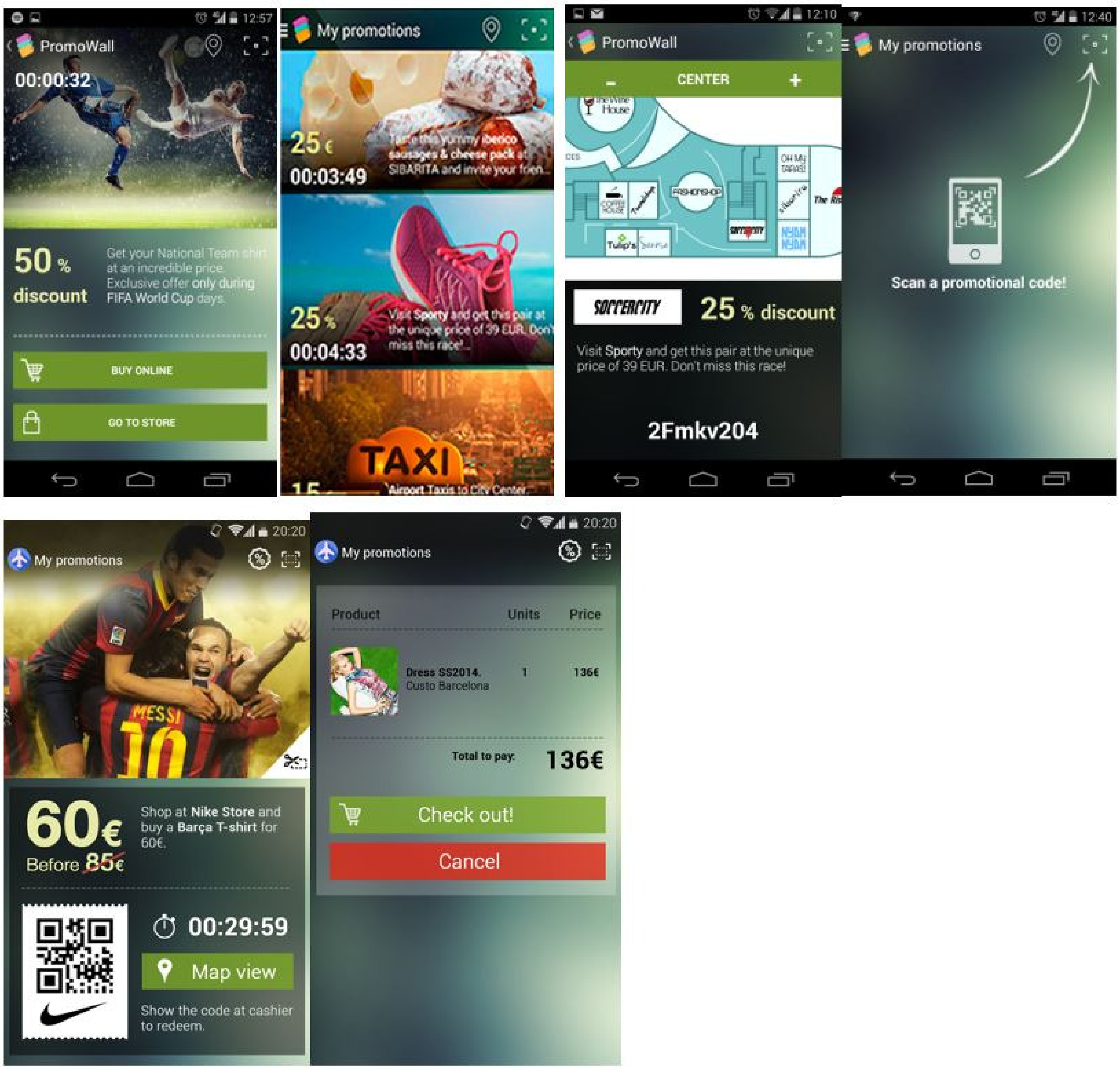

PromoWall
00:00:32
50 % discount
Get your National Team shirt at an incredible price. Exclusive offer only during FIFA World Cup days.
BUY ONLINE
GO TO STORE
My promotions
25 €
00:03:49
Taste this yummy iberico sausages & cheese pack at SIBARITA and invite your frien...
25 %
00:04:33
Visit Sporty and get this pair at the unique price of 39 EUR. Don't miss this race!...
TAXI
Airport Taxis to City Center
CENTER
25 % discount
Visit Sporty and get this pair at the unique price of 39 EUR. Don't miss this race!
2Fmkv204
Scan a promotional code!
60€
Before 85€
Shop at Nike Store and buy a Barça T-shirt for 60€.
00:29:59
Map view
Show the code at cashier to redeem.
Product
Units
Price
Dress SS2014. Custo Barcelona
1
136€
Total to pay:
136€
Check out!
Cancel

Figure 39: Promowall Smartphone application screenshots

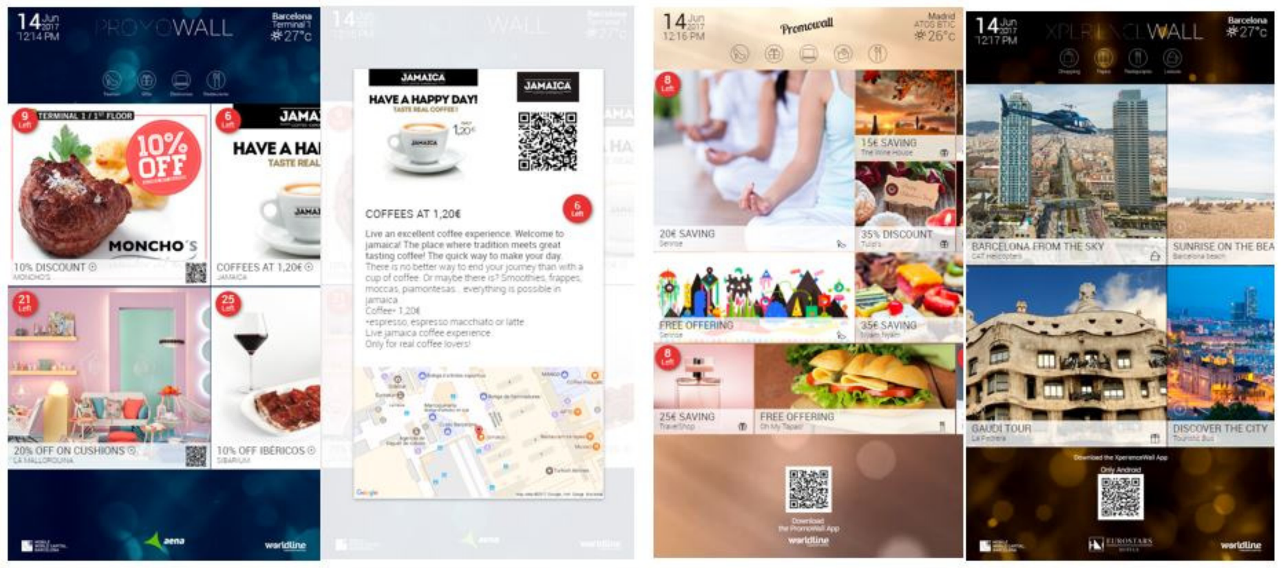


Figure 40: Screenshots of Web applications running on Promowall devices

5.4.3.5 Implementation

As mention above, there are three pieces of software involved in this application that are implemented in the following way:

- Backend: J2EE application implemented using the following frameworks and tools
    - Spring Framework: core framework
    - MySQL database
    - Apache CXF: implement RESTful services
- SmartTV frontend: HTML5 site
    - jQuery: core library
    - Mustache.js: logic-less HTML template library
    - Hammer.js: gesture library for web
    - Isotope: dynamic masonry tile layout
- Mobile application: Android native application

This application does not require any special attention. The effort on this platform is to convert this in a real *symbIoTe platform*.

Currently Promowall application is finished but it is being adapted to a symbIoTe enabled platform based on R2 symbIoTe release. This process is on an early stage yet.

5.4.3.6 Initial Functional Tests

This application is fully developed and it does not need any new functional tests.

5.5 Use Case Scenario "Smart Yachting"

The focus of Smart Yachting is to provide advanced services for the Yachting industry based on IoT solutions. From an implementation viewpoint, the use case focuses on two specific showcases: Smart Mooring and Automated Supply Chain.

The former aims to automate the mooring procedure of the port, in itself a quite bureaucratic and tedious process, since Marinas operate in strongly regulated contexts. For the use case, the workflow logic is provided by a Navigo application (Portnet).

Automated Supply Chain aims to automatically identify the needs for goods and services on board of the Yacht, so that automated requests for offers can be issued on the marketplace platform of the Port, provided by another application of the Navigo infrastructure (Centrale Acquisti).

Both showcases exploit data from IoT sensors to automatically acquire information from the Yacht and to pass them to the aforementioned business applications that are connected to the Port infrastructure.

Therefore, the context for the use case is based on the following hypotheses:

- the Port has an IoT platform
- the Yacht has an IoT Platform on board
- these two platforms interoperate through symbIoTe
- no need for these platforms to be the same, as long as they are both symbIoTe enabled (for example in the use case the Port has Navigo Digitale as the IoT platform, while the Yacht is supposed to have Nextworks' Symphony on board)
- the Port IoT Platform connects with sensors in its area
- the Port is seen as a Smart Space, that interacts with Smart Devices through both LoRaWAN and WiFi
- the Mooring and the Supply Chain Management systems are connected to the symbIoTe ecosystem through "Enablers".

The latter choice should facilitate the integration of these business applications with symbIoTe, by encapsulating the technical details of the whole IoT infrastructure and exposing only the minimum set of methods that must be implemented, to guarantee the communication and the data exchange envisioned in the use case.

This is particularly important, since each Port that in the future might adhere to symbIoTe for implementing the Smart Yachting use case, must develop the integration of their Mooring and Supply Chain Management systems (not necessarily based on Navigo's products): we want therefore that this integration could be as simple and straightforward as possible.

5.5.1 Application: Smart Mooring

As said, Smart Mooring aims to simplify, through M2M interactions, the mooring authorization workflow. It allows the Port's workflow management system to automatically retrieve those data from the Yacht that are needed for the workflow authorization.

We want to intercept a particular phase of the Mooring process that starts when the Yacht is approaching – at a distance – the destination port and ends when it finally berths into one of its piers.

We assume that the initial mooring request (a sort of "booking" for the boat in the Port) always starts off-line or in any case outside symbIoTe: it might be performed through a phone call or a web access to the Mooring workflow management system. For example, when the boat is still far from the destination port (e.g., in the port of origin) the yachtsman calls the destination port to “book” its arrival at a specific time: the mooring workflow is therefore initiated by the port personnel. Instructions are then exchanged between the boat and the port authority personnel (e.g., the boat is expected to arrive at the destination port on a specific date and time and to berth on a specific pier).

With Smart Mooring we are implementing a scenario through symbIoTe enabled systems:

- sensors in the Harbor can detect when the Yacht is approaching
    - *No need for a phone call to announce the arrival*
    - through M2M interactions the mooring workflow can be identified and updated automatically
- data from sensors (e.g., latest routes, fuel consumption, emissions...) can be automatically attached to this workflow
    - *No need to fill-in paper files*
    - *A larger amount of information can be tracked*
    - *No risk of errors or of wrong/false data reported*
- automatic communication can be sent to the port personnel to await the incoming ship
    - *Simplifying and empower Port's organization*
    - *Detecting when the vessel has finally arrived at the berth through sensors*

There should not be any need for the yachtsman to physically go to the Port Authority, unless problems are detected and reported.

5.5.1.1 Design

A simplified architectural view of the present showcase is provided in the diagram that follows.

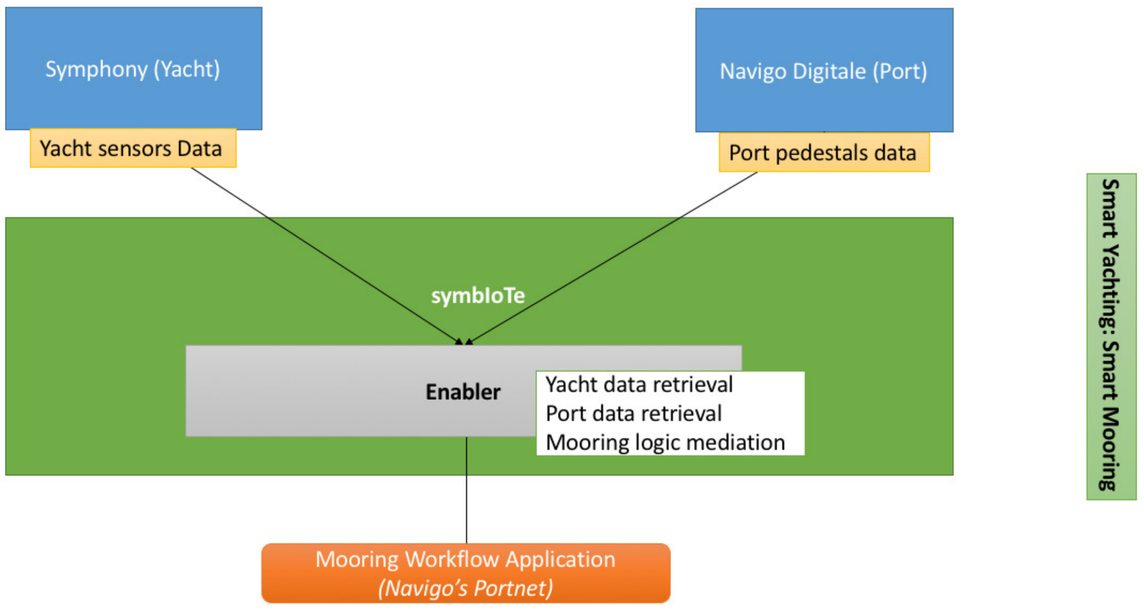


Figure 41: Architectural view of the Smart Mooring showcase

As indicated in the diagram, the Mooring Workflow application interacts, through symbIoTe's components, with the Yacht's IoT platform to receive data from sensors that must be attached to the authorization workflow, while sensors in the Port area, managed by its IoT platform, can recognize when the Yacht has finally berthed on the assigned pier.

For the use case, we are aiming to integrate IoT platforms and applications of project partners, namely Nextworks' Symphony and Navigo's Navigo Digitale and Portnet. Actually, to make it relevant, it is essential that in perspective the Smart Yachting use case can be adopted by the largest number of Yacht manufacturers and Port Authorities. As seen, we envision the Yacht as a Roaming Device: the more ports support Smart Yachting services, the more useful they will be seen by the Yachtsmen (and by Yacht Manufacturers).

Allowing other Mooring Applications, beyond Navigo's Portnet, to become symbIoTe-enabled is at the same time essential and a critical factor, since there isn't any standard, nor a market leader in this arena. In order to encourage software vendors to adopt this model, we must simplify their work: that is why, as indicated at the beginning of this section, it has been decided to encapsulate the integration details within an enabler, which will hide all the possible complications and provide simple cooperation mechanisms.

The design of this showcase was elaborated in respect of the general project requirements. In particular, the most relevant requirements have been taken into account, and these are:

- #4 – IoT services MUST appear to application developers in a homogeneous manner. Data source/identity shall be exposed to application developers.
- #5 – The system MUST monitor the availability of the IoT services registered by IoT platform operators.

- #11 – The information from IoT services and IoT devices MUST have the units in which the data is described associated to standard unit of the common information model (meters, kg, etc.). The encoding of units should adhere to a standard (e.g. UCUM).
- #12 – The common information model MUST support geo-reference information
- #19 – symbIoTe MUST distinguish IoT devices which are fixed (geo-location does not change over time) and mobile (their location changes).
- #21 – The system SHOULD allow application developers to create their own enablers (focusing on a single domain or be cross-domain), defining their own logic, etc.
- #61 - symbIoTe smart spaces SHOULD be able to operate without a permanent Internet connection.

The reason for their importance becomes clear when reading the description of the showcase that follows.

The use of symbIoTe will allow Mooring Workflow applications used in Marinas – like Navigo's Portnet – to deeply simplify their processes and data entry phases, by automating the retrieval of information from sensors of the approaching Yacht and of the Port.

5.5.1.2 Compliance Level

The main assumptions that we are making for implementing the Smart Mooring showcase are:

- Level 3 compliance: as said, we see the Yacht as a Smart Device (SDEV) and the Port as a Smart Space (SSP). The implementation of the showcase becomes similar to the symbIoTe scenario of a Smart Device (the Yacht) entering a Smart Space (the Port).
- Level 4 compliance: we assume that the Yacht maintains its ID when moving between Ports. The Yacht will be therefore seen as an example of a Roaming Device

By default the use case will also imply Level 1 symbIoTe compliance.

5.5.1.3 Platform

Smart Mooring will be based on the Navigo Digitale IoT platform and on the Navigo's business application Portnet, the latter integrated in symbIoTe through an Enabler. On the Yacht side, Nextworks' Symphony IoT platform will be used.

5.5.1.4 User Interaction

From a functional viewpoint, Smart Mooring aims to reduce user activities at a minimum. The following description therefore is not centered around human behavior but on a pure M2M interaction.

Together with the team working on the Smart Space middleware (WP4) and the Enabler (WP2) implementations, Navigo has refined the present showcase, to make sure that it can fit with the general project vision and that it can be L3 & L4 compliant. The hypotheses that we are considering for Smart Mooring involve several interactions between the Boat, the Port IoT System and the symbIoTe components. In details:

- Since the Yacht is a roaming device, its ID must be maintained in the Registry. Here two specific properties will be associated (and updated) for each Yacht: ConnectionStatus and ConnectedInPort. The former registers how (and if) the Yacht is connected and can assume as possible values: “disconnected”, “LoRaWAN” and “WiFi”. The latter, when ConnectionStatus is not “disconnected”, will take as the value the ID of the Port where the Yacht is connected.
- LoRaWAN connectivity will be used: we assume therefore that a specific LoRaWAN controller is attached to the SSP middleware.
- When approaching the port, the vessel – as a SDEV – is detected by the SSP LoRaWAN controller and registered by it to the SSP Innkeeper (which, in turn, updates the Yacht/SDEV properties in the symbIoTe Core Registry).
- The mooring application receives through its enabler a notification that a new Smart Yacht SDEV has been registered to the SSP of the port: this event activates the mooring workflow.
- When the Yacht is near the port, again through LoRaWAN, a Wi-Fi password is transmitted to the Yacht SDEV, which starts a full Internet connection. Therefore the Yacht, as a SDEV, first connects via LoRaWAN and then connects via Wi-Fi when the latter signal is strong enough. This connection change allows Portnet’s Enabler to request the retrieval of data from boat sensors.

In particular, the information that we assume to fetch from Yacht’s sensors are:

- Navigation routes from the GPS and the Wheelhouse sensors. We assume that Portnet receives an array of the most recent routes, including the one from the port of departure
- Yacht speed (in knots)
- Average Fuel Consumption per nautical mile (in litres)
- Fresh, Grey and Black Water tanks level (in litres)
- Service Fuel and Storage Fuel Oil tanks level (in litres)
- Port Exhaust and Starboard Exhaust temperature (in degree Celsius).

Portnet also needs to know when the Yacht has finally berthed on the specified pier. We assume to have a RFID/beacon sensor on each Yacht (e.g., installed on the on-board device) and a RFID/Beacon Receiver on each pier, managed by the Port IoT platform.

We are also assuming that the Mooring application logic is mainly on the port side, but there might be a specific "Smart Mooring" device on board, to simplify the management of the sensors involved in the showcase (LoRaWAN sensor, Wi Fi antenna, RFID/Beacon). Since this device can be very simple and "low-cost" (and therefore very easy to install), it can be used stand-alone in smaller boats (below 15 metres): this should open the possibility to exploit Smart Mooring to a larger amount of boats, not only high-end yachts.

The interactions described above are depicted in the following UML sequence diagrams (see Figure 42 and Figure 43). We have divided the mooring process in two parts, where the first one describes the moment in which the Yacht is approaching the port. In details:

- The Mooring Management System (Portnet) awaits for the incoming Yacht; it requests to the enabler to detect the Yacht arrival. The enabler will query, at a specified frequency (e.g. every 5 minutes), the Registry to detect when the Yacht's ConnectionStatus becomes different from "disconnected".
- The Yacht LoRaWAN sensor connects to the Port's Antenna and sends its ID. The LoRaWAN controller, which will be supervised by the Smart Space Innkeeper, will update the ConnectionStatus and ConnectedInPort properties of the Yacht in the Registry.
- The Enabler queries once again the Registry: this time the ConnectionStatus = "loRaWAN", so it notifies Portnet of the incoming ship.
- Portnet automatically starts the mooring procedures for the specific Boat and alerts the Port Authority operators and Port personnel.

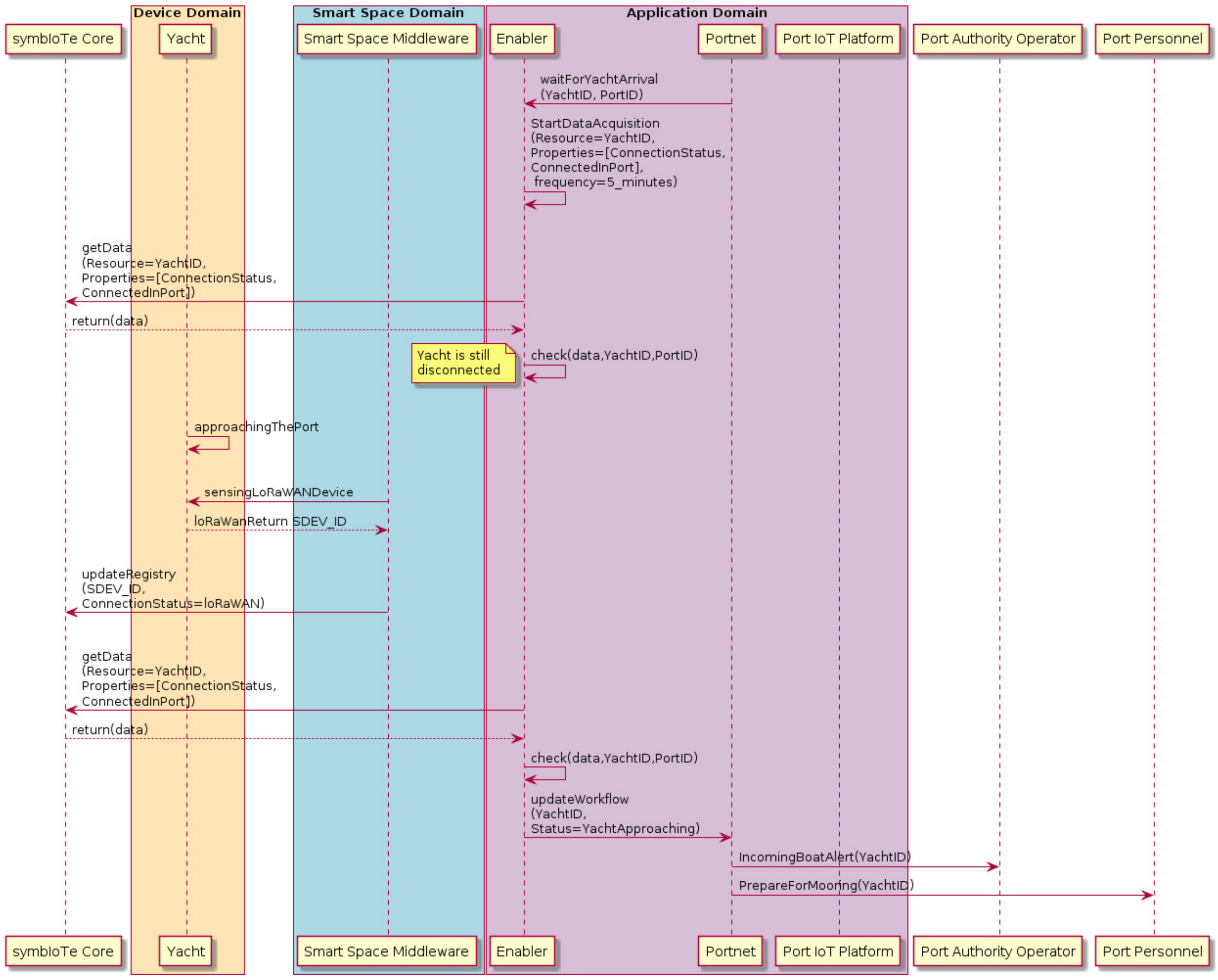

Device Domain
Smart Space Domain
Application Domain
symbIoTe Core
Yacht
Smart Space Middleware
Enabler
Portnet
Port IoT Platform
Port Authority Operator
Port Personnel
waitForYachtArrival (YachtID, PortID)
StartDataAcquisition (Resource=YachtID, Properties=[ConnectionStatus, ConnectedInPort], frequency=5_minutes)
getData (Resource=YachtID, Properties=[ConnectionStatus, ConnectedInPort])
return(data)
Yacht is still disconnected
check(data,YachtID,PortID)
approachingThePort
sensingLoRaWANDevice
loRaWanReturn SDEV_ID
updateRegistry (SDEV_ID, ConnectionStatus=loRaWAN)
getData (Resource=YachtID, Properties=[ConnectionStatus, ConnectedInPort])
return(data)
check(data,YachtID,PortID)
updateWorkflow (YachtID, Status=YachtApproaching)
IncomingBoatAlert(YachtID)
PrepareForMooring(YachtID)

Figure 42: Sequence Diagram of the Yacht approaching the Port

The second sequence diagram shows the arrival of the Yacht in the harbor and the accomplishment of the mooring procedure. In details:

- Through LoRaWAN the Yacht receives the credentials to connect to the Port's Wi-Fi network: they will be sent by the SSP middleware (note: the actual process to implement this is still under investigation in WP4)
- The Yacht activates the Wi-Fi connection when the signal is strong enough
- The symbIoTe-Agent of the Yacht connects to the SSP
- The SSP middleware updates the Yacht property ConnectionStatus to "WiFi" in the Registry.
- The Enabler queries once again the Registry: this time the ConnectionStatus = "WiFi", so it communicates to the Enabler to stop data acquisition for the Yacht's properties.
- The Enabler now requests to acquire data from the Yacht, by invoking its two services getLatestSensorDataService and getLatestRouteService.
- The Yacht grants access and returns its data
- The Enabler receives these data, verifies them, stops data acquisition and passes the information to Portnet
- Portnet updates the workflow, processes the incoming data and sends alerts to both the Port Authority Operator and the Port Personnel
- The Enabler requests to receive data from the Port IoT platform, to verify if the Yacht has arrived at the specified pier. Technically speaking it queries the value of the presence sensor attached to the Port's pier.
- the Yacht arrives at the pier and through RFID sensors the Port IoT platform acknowledges its presence
- The Enabler queries the Port IoT platform and finally verifies that the Yacht has arrived at the correct destination. It passes the information to Portnet.
- Portnet sends and alert to the Port Authority Operator and closes the mooring workflow.

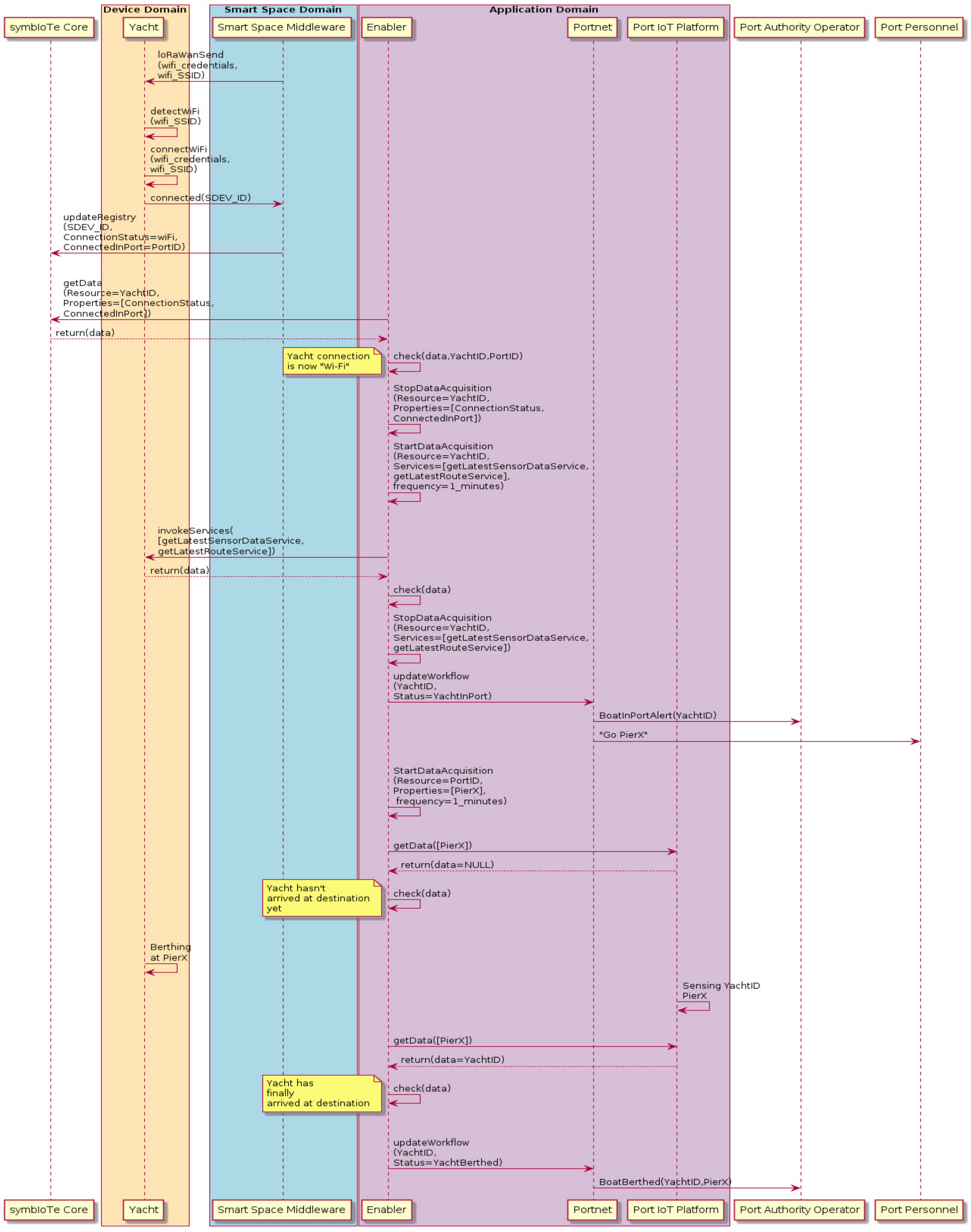

Device Domain
Smart Space Domain
Application Domain
symbIoTe Core
Yacht
Smart Space Middleware
Enabler
Portnet
Port IoT Platform
Port Authority Operator
Port Personnel
loRaWanSend (wifi_credentials, wifi_SSID)
detectWiFi (wifi_SSID)
connectWiFi (wifi_credentials, wifi_SSID)
connected(SDEV_ID)
updateRegistry (SDEV_ID, ConnectionStatus=wiFi, ConnectedInPort=PortID)
getData (Resource=YachtID, Properties=[ConnectionStatus, ConnectedInPort])
return(data)
Yacht connection is now "Wi-Fi"
check(data,YachtID,PortID)
StopDataAcquisition (Resource=YachtID, Properties=[ConnectionStatus, ConnectedInPort])
StartDataAcquisition (Resource=YachtID, Services=[getLatestSensorDataService, getLatestRouteService], frequency=1_minutes)
invokeServices([getLatestSensorDataService, getLatestRouteService])
return(data)
check(data)
StopDataAcquisition (Resource=YachtID, Services=[getLatestSensorDataService, getLatestRouteService])
updateWorkflow (YachtID, Status=YachtInPort)
BoatInPortAlert(YachtID)
"Go PierX"
StartDataAcquisition (Resource=PortID, Properties=[PierX], frequency=1_minutes)
getData([PierX])
return(data=NULL)
Yacht hasn't arrived at destination yet
check(data)
Berthing at PierX
Sensing YachtID PierX
getData([PierX])
return(data=YachtID)
Yacht has finally arrived at destination
check(data)
updateWorkflow (YachtID, Status=YachtBerthed)
BoatBerthed(YachtID,PierX)

Figure 43: Sequence Diagram of the Yacht arriving at the Port

5.5.1.5 GUI Design

No specific GUI is needed for this showcase. The only GUIs are those of the Portnet application (beyond the scope of symbIoTe).

5.5.1.6 Implementation

The following programming languages are used for the development of the showcase:

- Navigo Digitale IoT platform: its back-end has been developed in Python while the front-end is a web application
- Portnet is a Java web application, currently implemented in Liferay. A new version, based on the PHP Drupal 8 framework, is currently under development (outside of the scope of symbIoTe)
- Specific logic for the integration of Portnet with the Enabler will be implemented in Java.

The Navigo Digitale IoT platform already exists and its Level 1 compliance to symbIoTe is currently under development. As indicated, Navigo is working on a new version of Portnet. The latter module will heavily rely on symbIoTe' Smart Space Middleware, which is still in the design phase.

5.5.1.7 Initial Functional Tests

A specific test plan will be defined to cover all kinds of tests, from functional to integration and possibly load testing.

5.5.2 Application: Automated Supply Chain

The Automated Supply Chain (ASC) workflow can be seen as a particular case of the latter part of the Mooring use case. The Centrale Acquisti web application (by Navigo), through an enabler, accesses the resources of the Yacht (sensors) to retrieve information about the needs (of goods or services) on board.

Like in the previous case, we assume that the showcase will always starts off-line or in any case outside symbIoTe.

5.5.2.1 Design

A simplified architectural view of the present showcase is provided in Figure 44.

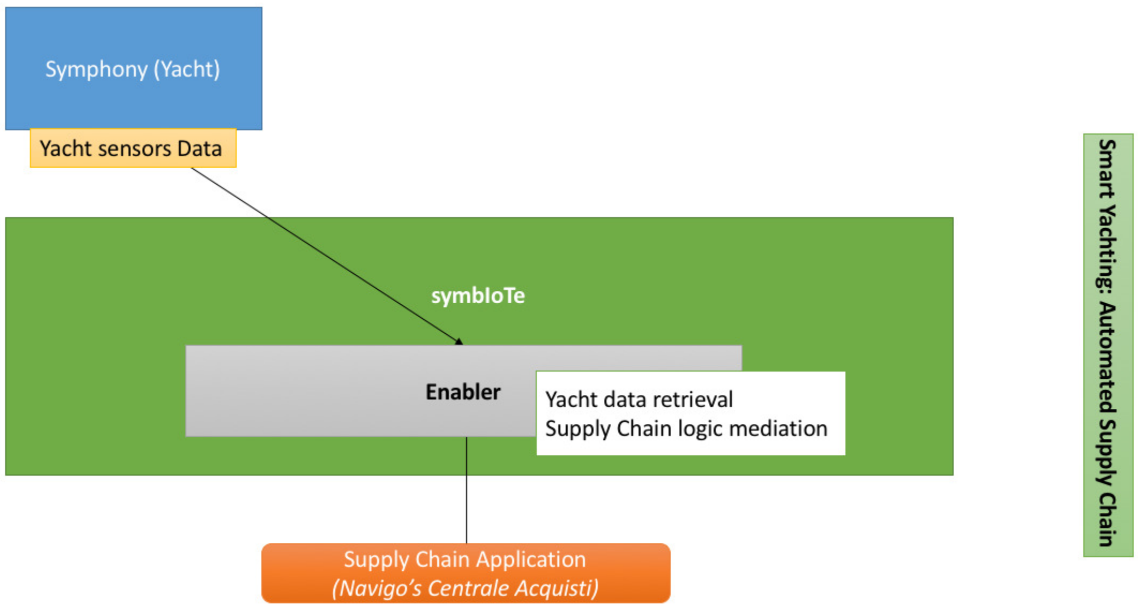

Symphony (Yacht)
Yacht sensors Data
symbIoTe
Enabler
Yacht data retrieval
Supply Chain logic mediation
Supply Chain Application
(Navigo's Centrale Acquisti)
Smart Yachting: Automated Supply Chain

Figure 44: Architectural view of the Automated Supply Chain showcase

Similarly to the case of Smart Mooring, we have an application here (Navigo's Centrale Acquisti) that exploits M2M and symbIoTe to automatically get the list of the possible needs of goods and services on board of the Yacht, as detected by its IoT platform (for the use case, Nextworks' Symphony). Again, we aim to involve other third party software vendors that provide applications similar to Centrale Acquisti: in order to simplify their integration task, we propose the use of an enabler to mediate the interaction with symbIoTe infrastructure.

The design of this showcase was elaborated in respect of the general project requirements. In particular, the most relevant requirements that have been taken into account are:

- #4 – IoT services MUST appear to application developers in a homogeneous manner. Data source/identity shall be exposed to application developers.
- #5 - The system MUST monitor the availability of the IoT services registered by IoT platform operators.
- #15 – The information model of the system SHOULD comply to standardized ontologies where possible and SHOULD try to be compatible to the data model of the other IoT-EPI projects. *(In the Automated Supply Chain showcase, the schema.org ontology is used)*
- #21 – The system SHOULD allow application developers to create their own enablers (focusing on a single domain or be cross-domain), defining their own logic, etc.

The use of symbIoTe in the Centrale Acquisti application will simplify how Yachtsmen can resupply or execute maintenance tasks on the Yacht by automatically finding possible sellers or service providers in the area, even on their first visit in the (symbIoTe enabled) Port.

5.5.2.2 Compliance Level

This showcase can be implemented as a Level 1 compliance scenario.

5.5.2.3 Platform

Automated Supply Chain will be based on the integration in symbIoTe of the Navigo's business application Centrale Acquisti through an Enabler. On the Yacht side, Nextworks' Symphony IoT platform will be used.

5.5.2.4 User Interaction

While the yacht is berthed in the "smart" port, the yachtsman connects to the Centrale Acquisti supply chain application through a common web browser. He/she gives the authorization to Centrale Acquisti to connect through symbIoTe to the machine data of the boat, that automatically provides indication about needs on board, whether of maintenance or resupply nature. Needs are expressed by identifying the categories of the possible suppliers that might fulfil them: for this purpose the schema.org ontology will be used. For example the schema:ElectronicsStore class will be used when some maintenance on the electric systems of the Yacht have been identified.

The following sequence diagrams illustrates the steps through which the Automated Supply Chain will take place (Figure 45). In details:

- The Enabler requests to acquire data from the Yacht, by invoking its two services getMaintenanceNeedsService and getConsumableNeedsService.
- The Yacht grants access and returns its data
- The Enabler receives these data, verifies them, stops data acquisition and passes the information to Centrale Acquisti
- Centrale Acquisti tries to find a possible match with the suppliers of goods and services in the port area.
- It returns the yachtsman with a list of proposals

The yachtsman will manage these proposals in the Centrale Acquisti web application. No specific GUI is needed for this showcase. The only GUIs are those of the Centrale Acquisti application (beyond the scope of symbIoTe).

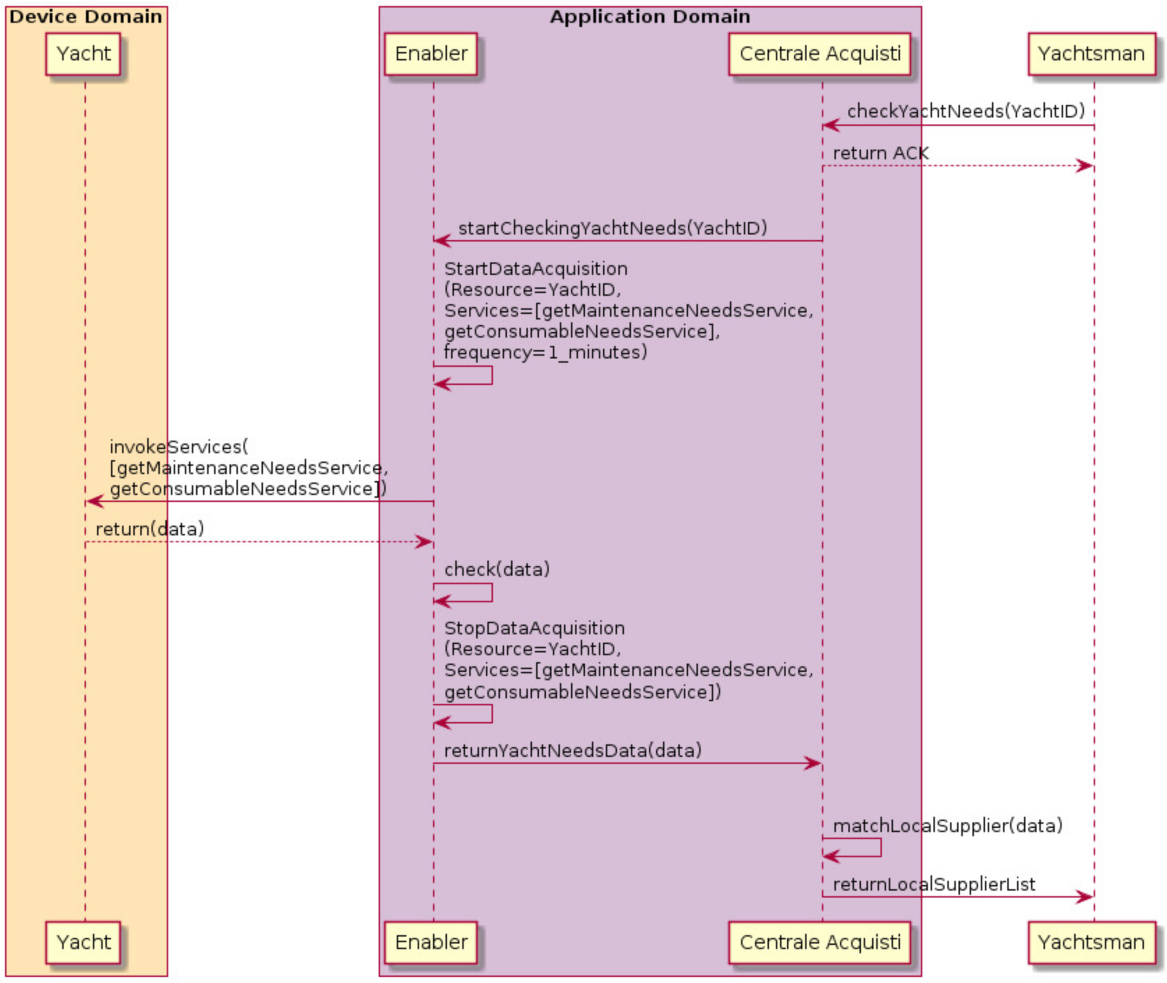

Device Domain
Application Domain
Yacht
Enabler
Centrale Acquisti
Yachtsman
checkYachtNeeds(YachtID)
return ACK
startCheckingYachtNeeds(YachtID)
StartDataAcquisition (Resource=YachtID, Services=[getMaintenanceNeedsService, getConsumableNeedsService], frequency=1_minutes)
invokeServices([getMaintenanceNeedsService, getConsumableNeedsService])
return(data)
check(data)
StopDataAcquisition (Resource=YachtID, Services=[getMaintenanceNeedsService, getConsumableNeedsService])
returnYachtNeedsData(data)
matchLocalSupplier(data)
returnLocalSupplierList

Figure 45: Sequence Diagram of the Automated Supply Chain showcase

5.5.2.5 Implementation

The following programming languages are used for the development of the showcase:

- Navigo Digitale IoT platform: its back-end has been developed in Python while the front-end is a web application
- Centrale Acquisti is a PHP application, implemented with the WordPress framework
- Specific logic for the integration of Centrale Acquisti with the Enabler will be implemented in Java.

The Navigo Digitale IoT platform already exists and its Level 1 compliance to symbIoTe is under development. Centrale Acquisti already exists while its Enabler is currently in the design phase.

5.5.2.6 Initial Functional Tests

A specific test plan will be defined to cover all kinds of tests, from functional to integration and possibly load testing.

5.6 Platforms and their Integration Status

Several existing IoT platforms are being integrated and used in before mentioned applications, as can be seen in Figure 10 to Figure 44. The full list is given in Table 7 showing some basic information and the integration status. Details on their integration are provided in the Appendix section.

Table 7: Information on IoT platforms and their integration status

Platform	Intended Compliance Level	Owner	Use Case	Related Applications	Integration Status
OpenIoT	L1 / L2	FER	Smart Mobility	Mobile Routing Application Web Routing Application	openIoT is fully integrated at compliance Level L1 for Release R2
openUwedat	L1 / L2	AIT	Smart Mobility	Mobile Routing Application Web Routing Application	Integration with symbIoTe finished for Release R2. Tested with a small specialized testing application and with the demo web application. Work still missing: Tap into relevant data providers for fixed stations' data.
openUwedat	L1 / L2	AIT	Smart Residence	Smart Healthy Indoor Air	See Status from the UC "Smart Mobility"
MoBaaS	L1 / L2	UW	Smart Mobility	Mobile Routing Application Web Routing Application	Integration with symbIoTe done for release 1, still needs to be done for release 2.
Symphony	L1 / L3	NXW	Smart Residence	Smart Area Controller Smart Home Comfort	L1 compliance implemented
Symphony	L3 / L4	NXW	Smart Yachting	Portnet	Symphony is already L1

				Centrale Acquisti	compliant (see use case smart residence).
KIOLA	L1	AIT	Smart Residence	Smart Health Mirror	L1 compliance implementation in progress
nAssist	L1 / L2	S&C	Smart Residence	Smart Healthy Indoor Air	symbIoTe compliance under development[31]
IOSB Building Management	L2	IOSB	Edu Campus	Searching for a room	L1 compliance implementation in progress
KIT SmartCampus	L2	IOSB/KIT	Edu Campus	Searching for a room	L1 compliance implementation in progress
Beacons Platform	L1	WLI	Smart Stadium	Visitor Application Retailer Application	Early stage of development, as explained in Status from the Smart Stadium use case.
User Platform	L1	WLI	Smart Stadium	Visitor Application	Early stage of development, as explained in Status from the Smart Stadium use case.
Promotion/Information Platform	L1	WLI	Smart Stadium	Visitor Application Retailer Application PromoWall Application	Early stage of development, as explained in Status from the Smart Stadium use case.
Remote ordering platform	L1	WLI	Smart Stadium	Visitor Application Retailer Application	Early stage of development, as explained in Status from the Smart Stadium use case.
Navigo Digitale	L1	Navigo	Smart Yachting	Portnet	symbIoTe compliance under development
				Centrale Acquisti	

[31] See Section 5.1.1.5

6 Conclusions

The Internet has already evolved into a highly innovative and competitive marketplace for applications, services, and content. Due to the widespread access to the Internet and availability of mobile devices, new requirements have emerged due to the growing number of broadband users worldwide. The users are in demand of novel applications that simplify their daily activities in various situations and environments, being that of a smart home or accessing services when visiting shopping malls and stadiums. Moreover, the lower entry barriers for non-technical users to become content and service providers, and the available IoT platforms and services on the market have also added to the list of requirements and demands. All these requirements pose new challenges and call for a middleware solution to enable services (e.g., tracking and correlating health data from different IoT platforms in a smart home environment) and interface points for different use cases, application fields, and services.

This challenge is now addressed through the symbIoTe project, in which a set of defined use cases covering different domains provide a firm basis for developing applications running on top of the symbIoTe prototype, i.e., core services. The focus in this Deliverable D5.2 is set on reporting the ongoing symbIoTe system integration (i.e., symbIoTe prototype), as well as on the involved, symbIoTe compliant, applications aimed at the end users.

In the process of system integration the symbIoTe development team has followed the microservice architecture, thus providing a better scalability, performance and code maintenance. This was due to the nature of providing IoT services in general, which requires high level of distribution and performance. This has proven to be a good choice, as was observed in the stress testing runs that included varied number of users and concurrent request, and showing that the symbIoTe release R2 is relatively stable and that it can handle large amount of request for medium sized datasets.

The applications related to the five Use Cases are described in detail, and ready for an implementation and integration:

- The four Smart Residence applications cover the indoor, house environment involving air quality control, health monitoring, and comfort and device control.
- The two Smart Mobility and Ecological Routing web and mobile applications allow for analytical services based on collected data from several different IoT platforms.
- EduCampus scenario is meant to ease a student’s life by, e.g., allowing booking a room (for visiting, meeting) in a different campus.
- The five Smart Yachting and Smart Stadium applications come with functionalities and usages involving provision of location based services, e.g., information of available services, purchases, offers, automatic supply chains, etc.
- Smart Stadium in particular does not involve any specific sensors, but devices such as smartphones, tablets and TVs, and these play as the IoT elements.

Most of the use cases are utilizing IoT platforms at Level L1 or L2. Some use case scenarios make use of the symbIoTe platform L3 and L4 functionality.

The follow up Deliverable, D5.4, scheduled for month 30, will address the final symbIoTe prototype and implemented applications running on top of it.

7 References

[1] G. Dünnebeil, “D1.1: Initial Report on Use Cases,” 2016.

[2] R. Duro and G. Dünnebeil, “D1.3: Final Specification on Use Cases and Initial Report on Business Models,” 2016.

[3] P. Skočir, “D1.4: Final Report on System Requirements and Architecture,” 2017.

[4] R. Vitorino, K. Katsaros and J. Garcia, “D5.1: Implementation Framework,” 2016.

[5] M. Drobics, K. Kreiner and H. Leopold, “Next Generation ICT PLatform to Harmonize Medical, Care and Lifestyle Services,” *Adances in Intelligent Systems and Computing,* 2016.

8 Acronyms

AAL	Ambient Assisted Living
AIT	Austrian Institute of Technology GmbH
API	Application Programming Interface
APP	Application
ATOS	ATOS Spain SA
BLE	Bluetooth Low Energy Beacon
CLD	Cloud Domain (symbIoTe domain layer)
CO2	Carbon Dioxide
CPS	Cyber Physical Systems. A mechanism controlled or monitored by computer-based algorithms
DoW	Description of Work
eduroam	EDUcation ROAMing
FER	Faculty of Electrical Engineering and Computer Science, University of Zagreb
GPS	Global Positioning System
H2020	“Horizon 2020” EU Research and Innovation Programme
HTTP	Hypertext Transfer Protocol
IAQ	Indoor Air Quality
ICOM	Intracom Sa Telecom Solutions
ICT	Information and Communication Technology
IOSB	Fraunhofer Gesellschaft zur Förderung der Angewandten Forschung ev
IoT	Internet of Things
ITU-T	ITU Telecommunication Standardization Sector
JSON	Javascript Object Notation, a human readable data exchange format
JVM	Java Virtual Machine
KIOLA	Telehealth Service Platform
KIT	Karlsruhe Institute of Technology
LoRa	LoRa Alliance TechnologyLow Power Wide Area Network
LoRaWAN	
MMSI	Maritime Mobile Service Identity
NAVIGO	Na.Vi.Go. Societa Consortile a Responsabilita Limitata
NXW	Nextworks

OData	Open Data Protocol, an open protocol to allow the creation and consumption of queryable and interoperable RESTful APIs
openUwedat	AIT's platform to manage observations and related data
OSM	Open Street Map
P2P	Peer-to-Peer
PIM	Platform Information Model
POI	Point of Interest
RAP	Resource Access Proxy
RDF	Resource Description Framework, a description standard for semantical relations
REST	REpresentational State Transfer
S&C	Sensing & Control Systems SL
SD	Device Domain (symbIoTe domain layer)
SPARQL	A query language for semantically linked data sets (see RDF)
SSP	Smart Space Domain (symbIoTe domain layer)
symbIoTe	Symbiosis of Smart Objects across IoT Environments
TPV	Third Party Verification
ToC	Table of Contents
UC	Use Case
UNIDATA	Unidata Spa
UNIVIE	Universität Wien
UNIZG-FER	Sveučilište u Zagrebu, Fakultet Elektrotehnike i Računarstva
URL	Uniform Resource Locator
UW	Ubiwhere Lda
VIP	Vipnet d.o.o.

9 Appendix

9.1 Platform Integration Scenarios

There are two foreseen scenarios to integrate a platform into the cloud: The first scenario means that you have to modify the Resource Access Proxy (RAP). The RAP includes a class with two prepared, empty methods that you can modify to add you functionality. The advantage of this approach is that you do not need to fiddle around with Gradle and Spring, such that you can concentrate on the core task to add the needed functionality. The disadvantage is that you need to modify source code of other people and, if the RAP code is modified itself, you need to merge your changes with changes you pull from git. Another disadvantage is that there is no explicit initialization phase for the prepared class so if your platform needs extended initialization you are on your own here.

The alternative approach is an “external” plugin that exists in an own process. The only constraint in this case is the API to the RAP itself, which must adhere to the predefined channels and messages. Beyond that you have all the freedom of design you want including using another implementation language. Of course, that also means you have all burdens that come with that freedom.

9.1.1 “Internal Integration”: Integration of OpenIoT

OpenIoT[32] has a simple interface so the integration approach is to use the “internal” plugin (i.e. platform-specific plugin of RAP). This sub-section presents details how to set up RAP, i.e., it shows some code examples of platform-specific plugin of RAP that process the user request and provides data. The first two examples are based on the specific class[33] that is provided in RAP repository and can be easily extended and modified to implement platform specific plug-in for RAP.

The PlatformSpecificPlugin class has already prepared two methods which are used in serving data requests (the pull mode): readResourceHistory and readResource (see Listing 1).

readResourceHistory returns all available observations for selected sensor, and `readResource` returns only last measured value. The sensor is identified by the value of `resourceId` field, which represents the platform-native id used during registration (i.e., the generic part of RAP handles translation between symbIoTe id and platform-native id). The return value is an instance of the `Observation` class, a helper class provided in the code base of the symbIoTe library. The commented block provides one hardcoded example to demonstrate to a developer how to create the `Observation` object (Listing 1).

[32] http://www.openiot.eu/
[33] eu.h2020.symbiote.plugin.PlatformSpecificPlugin

```
public List<Observation> readResourceHistory(String resourceId) {
    List<Observation> value = new ArrayList();
    // INSERT HERE: query to the platform with internal resource id
    // example
    //Observation obs1 = observationExampleValue();
    //Observation obs2 = observationExampleValue();
    //value.add(obs1);
    //value.add(obs2);
    value = getObservations(resourceId);
    return value;
}

public List<Observation> readResource(String resourceId) {
    List<Observation> value = new ArrayList();
    value.add(getObservations(resourceId).get(0));
    return value;
}
```

Listing 1: Configuration of RAP platform-specific plugin for pull requests.

The second example demonstrates how to set-up push mode of data delivery. This is done in two parts, the first part sets subscription mechanism to your platform (see Listing 2) and the second part pushes data toward an end-user using RAP (see Listing 3). The subscription to your underlying mechanism can be called from the `receiveMessage` method of the `PlatformSpecificPlugin` class. Within the switch/case statement a developer can set up the subscription based on the user request. The list of resources (i.e. sensors) which an end-user has requested to continuously receive data updates is stored in the `mess.getResourceInfoList()`. The resources are identified by their platform-native id, same as with previous example. The similar code is used to delete continuous request, i.e., to unsubscribe from data pushes.

```
public String receiveMessage(String message) {
    ...
    switch (access) {
    ...
        case SUBSCRIBE: {
            // insert here subscription to resource
            ResourceAccessSubscribeMessage mess =
(ResourceAccessSubscribeMessage) msg;
            log.info("Subscription requested");
            if (subscribe(mess.getResourceInfoList())) {
                log.info("Subscription successful");
            }
            break;
        }
        case UNSUBSCRIBE: {
            // insert here unsubscription to resource
            ResourceAccessSubscribeMessage mess =
(ResourceAccessSubscribeMessage) msg;
            log.info("Unsubscription requested");
            if (unsubscribe(mess.getResourceInfoList())) {
                log.info("Unsubscription successful");
            }
            break;
        }
...
}
```

Listing 2: Configuration of RAP platform-specific plugin for processing subscription requests

Data push is done upon receiving a data update, and the example is presented on the following code snippet. RAP pushes to end-users instances of `Observation` class, so if the underlying subscription mechanism returns different data objects it is necessary to create an `Observation` class from them. The rabbitMQ details where the generic part of RAP expects data updates is provided in the example.

```
Observation pushObs = receivedDataUpdate();
ObjectMapper mapper = new ObjectMapper();
mapper.configure(SerializationFeature.INDENT_OUTPUT, true);
mapper.setSerializationInclusion(JsonInclude.Include.NON_EMPTY);
byte[] dissMSG = mapper.writeValueAsBytes(pushObs);
TopicExchange t = new TopicExchange("symbIoTe.rapPluginExchange-
notification");
rabbitTemplate.convertAndSend(t.getName(), "symbIoTe.rapPluginExchange.plugin-
notification", dissMSG);
```

Listing 3: Configuration of RAP platform-specific plugin for push notification delivery

9.1.2 "External integration": Integration of openUwedat

For several reasons openUwedat[34] chose the path of an "external" plugin. These reasons were partly technical. Especially the not existing "init" phase was a showstopper for the "internal" approach. Other influences for the decision were more personal like inexperience with the spring framework by the integrator.

At the editorial deadline of this deliverable openUwedats implemented just the reading of measurements in pull mode. Implementing push mode is planned for the near future.

9.1.2.1 Technical environment

openUwedat uses the Jackson package to convert from java to JSON and back.

The general approach is very similar to the one chosen by openIoT. The major difference is that the openUwedatRAPPlugin runs in its own process. The communication via RabbitMQ between the RAP and the plugin is the same. One major difference though is that openUwedat needed to write its own support environment to handle RabbitMQ connections.

Where ever possible classes from the symbIoTe library where taken as containers to prepare JSON. Unfortunately the communication between the RAP and is not covered by classes from that library[35]. As a workaround the RAP itself was placed on the class path as well at compile time as at runtime.

9.1.2.2 Registring the plugin

Registering the platform is easy. You create a new instance of a class named RegisterPluginMessage, convert that to JSON and send it via RabbitMQ.

The essence of that code is shown in Listing 4.

[34] https://www.ait.ac.at/en/research-fields/environmental-and-crisis-disaster-management/uwedat/

[35] This issue is filed in JIRA under SYM-322

```
public static void registerPlugin(Connection conn, String plugin_platform_id,
boolean hasNotifications, boolean hasFilters) {
        try {
            RegisterPluginMessage msg = new
RegisterPluginMessage(plugin_platform_id, hasNotifications, hasFilters);
            ObjectMapper mapper = new ObjectMapper();
            mapper.configure(SerializationFeature.INDENT_OUTPUT, true);
            mapper.setSerializationInclusion(JsonInclude.Include.NON_EMPTY);
            String json = mapper.writeValueAsString(msg);

            Channel channel=conn.createChannel();

            byte[] rawMessage=json.getBytes("UTF-8");
            AMQP.BasicProperties messageProperties
             =new AMQP.BasicProperties.Builder()
                    .contentEncoding("UTF-8")
                    .deliveryMode(2)
                    .build();


            channel.basicPublish("symbIoTe.rapPluginExchange",
"symbIoTe.rapPluginExchange.add-plugin", messageProperties, rawMessage);
        } catch (Exception e ) {
             …
        }
    }
```

Listing 4: Code to register plugin

9.1.2.3 Providing measurements

To provide measurements a callback routine was registered with RabbitMQ. This callback is activated on incoming requests from the RAP. (Listing 5)

```
private static void registerAndArmAccessCallback(Connection conn) throws IOException
{

          Channel channel=conn.createChannel();
        channel.exchangeDeclare("plugin-exchange", "topic");
        String queueName = "RAPPlugin.openUwedat";
        channel.queueDeclare(queueName, false, true, true, null);
        channel.queueBind(queueName, "plugin-exchange", "get");   // TODO: add more
routing keys here

          Consumer consumer = new DefaultConsumer(channel) {
              @Override
              public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body)
                  throws IOException {

                   /**
                    * This routine handles incoming messages, tries to get a
response and sends it back to the initiator of the incoming message.
                    */
                   String jsonReply=openUwedatRAPMain.handleRequests(consumerTag,
envelope, properties, body);

                  AMQP.BasicProperties replyProps = new AMQP.BasicProperties
                          .Builder()
                          .correlationId(properties.getCorrelationId())
                          .contentType("text/plain")
                          .build();

                 Channel channelReply=channel;

                 channelReply.basicQos(1);

                 channelReply.basicPublish( "", properties.getReplyTo(), replyProps,
jsonReply.getBytes("UTF-8"));


                  return;

              }
            };

            String consumerTag=channel.basicConsume(queueName, true, consumer);
            System.out.println("Bound to the queue with consumer tag "+consumerTag);

    }
```

Listing 5: Provide data through RabbitMQ

Decoding and handling of the request is done in another routine which looks as follows:

```
protected static String handleRequests(String consumerTag, Envelope envelope,
BasicProperties properties,
                    byte[] body) throws UnsupportedEncodingException {


        String json = "";
        String message="";
        try {
             message=new String(body, "UTF-8");

             System.out.println("Handling a request for a message of "+message);

            ObjectMapper mapper = new ObjectMapper();
            ResourceAccessMessage msg = mapper.readValue(message,
ResourceAccessMessage.class);
            ResourceAccessMessage.AccessType access = msg.getAccessType();
            switch(access) {
                case GET:
                    ResourceAccessGetMessage getMsg=(ResourceAccessGetMessage)msg;
                    ResourceInfo info = getMsg.getResourceInfo();
                    List<Observation> observationList =
readResource(info.getInternalId());
                    json = mapper.writeValueAsString(observationList);
                    break;
                case HISTORY:
//                    List<Observation> observationLst =
readResourceHistory(info.getPlatformResourceId());
//                    json = mapper.writeValueAsString(observationLst);
                    throw new Exception("Access type " + access.toString() + " not
yet supported");
                case SET:
//                    ResourceAccessSetMessage mess = (ResourceAccessSetMessage)msg;
//                    writeResource(info.getPlatformResourceId(), mess.getValue());
                    throw new Exception("Access type " + access.toString() + " not
yet supported");
            }
        } catch (Exception e) {
            System.err.println("Error while processing message:\n" + message + "\n"
+ e);
        }

        return json;

      }
```

The routines readResource, readResourceHistory and writeResource are already openUwedat specific.