Symbiosis of smart objects across IoT environments

688156 - symbIoTe - H2020-ICT-2015

Implementation Framework

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

Title: symbIoTe Implementation Framework

Type: Public

Editor(s): Ricardo Vitorino, Konstantinos Katsaros, João Garcia

E-mail: rvitorino@ubiwhere.com, konkat@intracom-telecom.com, jmgarcia@ubiwhere.com

Author(s): George Papoutsis, Konstantinos Katsaros, Vasilis Glykantzis and Sergios Soursos, ICOM; Ricardo Vitorino and João Garcia, UW; Mario Kusek and Ivana Podnar, UNIZG-FER; Tomasz Rajtar, Mikolaj Dobski, Tomasz Nowak, Szymon Mueller and Marcin Plociennik, PSNC; Savio Sciancalepore, CNIT; Elena Garrido Ostermann and Jose Antonio Sanchez Murillo, ATOS; Karl Kreiner, Kurt Edegger, Christoph Ruggenthaler and Jasmin Pielorz, AIT; Michael Jacoby, IOSB;

Doc ID: D5.1-symbIoTe_Implementation_Framework

Amendment History

Version	Date	Author	Description/Comments
v0.1	30/5/2016	George Papoutsis, (ICOM)	Table Of Contents
V0.2	15/06/2016	Konstantinos Katsaros (ICOM)	Updated Table of Contents
V0.3	23/06/2016	Ricardo Vitorino (UW)	Updated Table of Contents and Assigned Contributors
V0.4	03/07/2016	Mario Kusek (UNIZG-FER) & Tomasz Rajtar (PSNC)	Completed Chapter 3
V0.5	20/07/2016	Konstantinos Katsaros (ICOM)	Updated Chapter 3 and ordering of sections and contents of the "Implementation Framework Procedures" section
V0.6	21/07/2016	Konstantinos Katsaros (ICOM), Vasilis Glykantzis (ICOM)	Completed Chapter 4, Added Section 5.1 on search tools
V0.7	28/07/2016	João Garcia (UW), Savio Sciancalepore (CNIT), Mario Kusek (UniZG), Elena Garrido Ostermann (ATOS), Jose Antonio Sanchez Murillo (ATOS), Konstantinos Katsaros (ICOM), Karl Kreiner (AIT), Kurt Edegger (AIT), Micheal Jacoby (IOSB)	Completed Chapter 5
V0.8	29/07/2016	Mikolaj Dobski (PSNC)	Added missing tools in chapter 4, fixed repository description
V0.9	29/07/2016	Tomasz Nowak (PSNC)	Added anomaly detection tools to chapter 5
V0.10	01/08/2016	Mikołaj Dobski (PSNC)	Added some missing elements and review of V0.7,0.8,0.9
V0.11	02/08/2016	João Garcia (UW)	Reviewed document, rewrote some sections of chapter 5, updated abbreviations
V0.12	08/08/2016	João Garcia (UW)	Updated the document with the answers given to the reviewers questions
V0.13	10/08/2016	Tomasz Nowak (PSNC)	Added anomaly detection introduction
V0.14	12/08/2016	Mikołaj Dobski (PSNC), Szymon Mueller (PSNC), Christoph Ruggenthaler (AIT), Marcin Plociennik (PSNC), Konstantinos Katsaros (ICOM)	Reworked the development cycle, updated git branching, CI integration and development frameworks; Reworked Netflix OSS chapter; SQA section added
V0.15	23/08/2016	Tomasz Nowak (PSNC), Ivana Podnar (UNIZG), João Garcia (UW)	Completed Security Coding Guidelines; Completed Eclipse OM2M description; Correct minor details with abbreviations
V0.16	29/08/2016	Ivana Podnar (UNIZG-FER); Elena Garrido Ostermann (ATOS); Jose Antonio Sanchez (ATOS); João Garcia (UW)	Fixed issues raised in review; Extended the Executive Summary; Added conclusions.

V1.0	31/08/2016	Vasilis Glykantzis (ICOM); Jasmin Pielorz (AIT);Sergios Soursos (ICOM)	Reworked project roles and feature planning; Final review; Minor final edits

Legal Notices

The information in this document is subject to change without notice.

The Members of the symbIoTe Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the symbIoTe Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Table of Contents

(This page is left blank intentionally.)

1 Executive Summary

The aim of deliverable D5.1, entitled “symbIoTe Implementation Framework”, is to define a workflow that the development tasks in WP2, WP3 and WP4 have to follow. Respectively, it addresses in particular the software developers of symbIoTe and serves as a guideline that ensures that quality standards are kept. At the same time, it provides a starting point for developers joining the project later on and facilitates a continuous integration of the software developed in the symbIoTe project. The deliverable in particular states the tools that will be used to support the defined workflow. Additionally, a set of technologies relevant to the development of symbIoTe software is listed, from which a selection of tools to be integrated in the project is made.

The document starts by defining the various roles within the project, inspired by the Agile approach for project development. The considered roles are Product Owner, Scrum Master, Component Owners and Development Team.

The deliverable then describes the system release iteration, which states the main implementation workflow from feature planning to system release. This workflow follows a a cycle of four steps, briefly described below:

- *Feature Planning*, where new features and system validation tests are specified;
- *Development*, where the collaborative implementation for the system takes place using the implantation framework procedures and tools defined in this deliverable, along with the continuous integration process;
- *System testing*, where the entire integrated system is tested;
- *System release*, following successful system testing, the system is then released and a new release cycle can commence.

The regularity of the releases is defined (standard release cycles of three months were selected), along with the frequency of regular meetings that help track the project’s progress and to solve problems that might come up.

To facilitate the developers in the implementation of the envisioned features, three types of internal reports are defined: the *release specification* report, which describes the feature planning phase, the *component release* report, which supports the consistent documentation of the implementation process and the *system release* report, which captures the results of the system release cycle.

Moreover, certain Software Quality Assurance standards and procedures are defined and will be followed in order to produce software of good quality. Finally, coding guidelines are proposed with the aim of preventing bugs and reducing the time developers spend fixing the code. Open Web Application Security Project’s (OWASP) Application Security Verification Standard is proposed to ensure proper security measures are used and specific Coding Guidelines are suggested for Java code development.

Various tools are listed that will support the implementation framework: Atlassian JIRA will be used for feature planning and issue tracking; Git will provide version control; github.com will be used as a code repository; Travis will be used for Continuous Integration; Bintray will be used as an artifact repository; and Gradle will be used as a building tool.

The document ends with the results of technology scouting, which are presented as a list of technologies that are relevant for the symbIoTe project. They will serve as a basis from which the technologies to be integrated into to the project will be chosen.

2 Introduction

The goal of this deliverable is to document one of the first outcomes of Task 5.1, which is the definition of the implementation framework for the symbIoTe architecture. It will serve as a guide for all development tasks in the technical work packages; namely WP2, WP3 and WP4. Given that the project's consortium is composed of 14 organisations spread across Europe, who are developing the same architecture and implementing a rather complex software system, there is a real need for setting up and adapting common methodologies, tools and workflows for the implementation of symbIoTe components. The implementation framework will facilitate the following: 1) the minimisation of conflicts between partners' contributions and promote code coherence within different software modules; 2) maintenance of stable integrated software versions, which allows more frequent releases, and finally 3) enhance the publication of packages such as libraries and SDKs on public open-source repositories.

While Section 3 details the workflow that will be adopted in the development of the prototype, Section 4 presents the tools that support the defined workflow and Section 5 lists a set of technologies relevant to symbIoTe that can later be chosen to be integrated in the project.

3 Implementation Framework Procedures

The symbIoTe workflow is inspired by the widely adopted Agile approach in software development, which provides a series of advantages, including the active engagement of all stakeholders, tight scheduling and control of development progress, flexible adaptation to evolving requirements, etc. (see Appendix A: Agile). However, the necessary adaptations and specific rules within the distributed software development teams are needed to reflect the specificities of a European research project.

The following subsections state the roles to be filled by the members of the consortium, the various steps in the system release iteration, the release cycle frequency along with the regularity of conference calls that support the releases, the reports that document the progress of the release and software quality assurance and coding guidelines to be followed.

3.1 Roles

The overall implementation workflow is supported by different roles assigned to the members of the consortium. The definition of the roles inherits features from the Agile approach (see Appendix A: Agile). In particular, the symbIoTe implementation framework includes the roles shown in the following table, for which a set of associated responsibilities is defined. These responsibilities are further explained in this section, in the context of the overall system release iteration.

Role	Responsibilities
Product Owner	• Feature specification
Scrum Master	• Feature planning • Assign component owners • Integration • Source tree set-up & maintenance • Tool-chain management • System Release
Development team	• Implement code locally • Commit often • Write unit tests • Participate/perform system tests
Component owners	• Coordinate development within individual partners • Provide support to Scrum Master for feature planning

Table 1 - Roles and associated responsibilities in the symbIoTe implementation framework.

In contrast to the common Agile practices, the consortium has decided to assign the role of Product Owner to a group of people instead of an individual. Particularly, the task leaders of the implementation tasks (see Table 2 – Assignment of roles to symbIoTe tasks), along with the Technical Coordinator (also being the T1.4 Task Leader), form a hierarchical, flexible and adaptable structure for the role of the Product Owner. The Technical Coordinator leads the efforts based on the input from the task leaders, whose role is to handle the details of feature specification for their own tasks. This was dictated by the large size and the highly distributed nature of the development team, as well as the complexity of the architectural design, e.g. various domains and interoperability-related compliance levels.

Additionally, we introduce the role of Component Owners, with the purpose of facilitating feature planning and coordination of the development process. As described in the next section, Component Owners are defined to handle the development process within subsets of the development team responsible for individual components of the architecture. These subsets may be defined within a single or more partners. The introduction of the Component Owner role aims to address the complexity of the development process in SymbIoTe, stemming from the large development team and the rich set of architectural components across various domains (application, cloud, smart space and smart device domain).

Product Owner – T1.4 Task Leader (UNIZG-FER) + Task Leaders							
Scrum Master – Integrator T5.1 Task Leader (ICOM)							
T2.2	T2.3	T3.2	T3.3	T4.1	T4.2	T4.3	T5.2
Developers	Developers	Developers	Developers	Developers	Developers	Developers	Developers

Table 2 – Assignment of roles to symbIoTe tasks.

3.2 System release iteration

System release iterations define the main implementation workflow. The structure of the system release iteration is illustrated in the following figure:

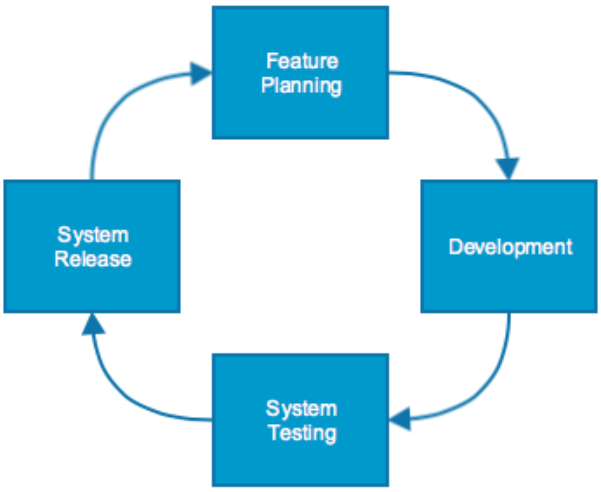

Figure 1: System release iteration in symbIoTe.

The following subsections contain a more detailed description of all the steps depicted in the previous figure.

3.2.1 Feature planning

In this step, the Product Owner specifies the list of features that have to be implemented during the current iteration. To this end, Task Leaders specify the list of desirable features, which are subsequently reviewed, modified or extended by the Technical Coordinator, who has the general overview of the project. The finalization of the feature specification process should be done in cooperation with the task leaders. This final feature specification is derived based on the overall architecture design of symbIoTe and detailed design performed within each task in order to guide the development team during the implementation process. The Scrum Master supports the feature specification process by providing input related to the current software architecture and its relation to the broader system architecture e.g., information regarding the inter-component communication interfaces, providing insights on best practices for the flow of information within certain procedures/features. Apart from the specification of new features, the feature planning stage includes a detailed specification of system tests whose purpose is to validate the interoperability of the components (i.e., the components that implement the desired features) and to ensure that the operation of the system is correct and in line with the intended behaviour. To this end, the system test specification includes several execution scenarios, with precise descriptions of the inputs and outputs to/of the system. The feature and system test specifications are documented in the Release Specification report (see Section 3.4.1).

Subsequently, the Scrum Master proceeds with the planning of the development process, which first includes the prioritization of the feature implementation tasks. The Scrum Master assigns the specified features and corresponding components to groups of developers (i.e., per partner), subject to the granularity of the particular component/feature. Component owners are responsible for the distribution of these tasks to individual developers within their local team (per partner). Moreover, Component owners monitor the

development process within their teams and provide feedback to the Scrum Master for the feature planning procedure e.g., fine-tuning of effort estimation for particular features.

The feature assignment includes a thorough explanation of the work that has to be done, by both the Scrum Master and the Product Owner. The feature/component assignment is done by the Scrum Master, who issues the corresponding tickets using the Feature planning tool (see Section 4.1). At the same time, the Scrum Master assigns the system tests to members of the development team (and possibly other partners, not involved in the development, for further validation). Similarly to component assignments, the assignment of system tests can be performed on a per partner level, i.e. either by groups of developers or a subset of the development team.

3.2.2 Development

The development stage comprises several steps, forming a development iteration as shown in Figure 2. The objectives of this iterative process are to allow the continuous integration of the source code as well as quick detection and correction of errors.

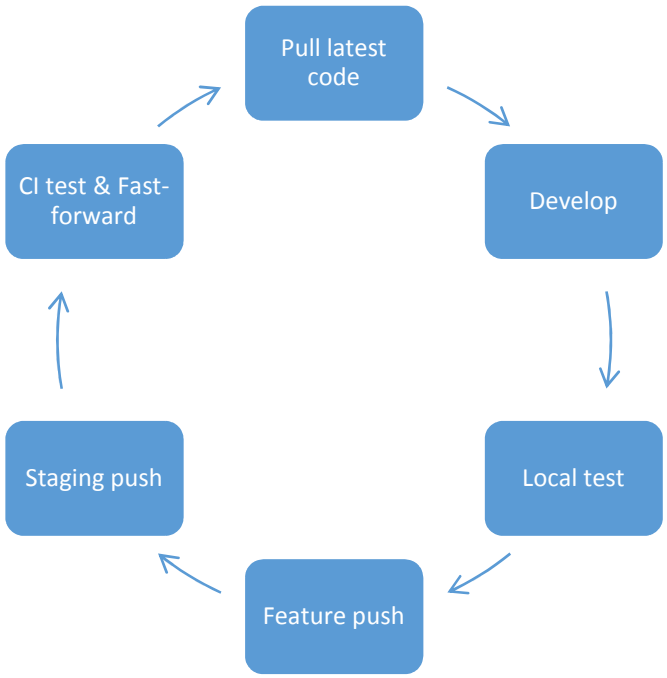

Figure 2: Development iteration process.

The following subsections contain a more detailed description of each of the development steps present in the previous figure.

3.2.2.1 Pull latest code

Developers accept their assigned tickets, pull the latest stable code from the repository and respective develop branch in order to add their features/fix bugs/etc. (see Section 4.4). The source code files (i.e. the source code developed and committed by the development team) are maintained in a project repository space with the assistance of a code repository tool (see Section 4.4).

3.2.2.2 Develop

Developers work on their features, bug fixes or code refactoring tasks on a dedicated feature branch derived from the develop branch. The latter action is performed using a Version Control tool (see Section 4.3). The update frequency for versioning follows the progress of the development and local preliminary validation of the developed components.

3.2.2.3 Local test

The developers test their code locally, i.e. they perform Unit tests.

3.2.2.4 Feature push

Developers periodically commit their changes locally and push them to the remote feature branch to maintain a backup of their work. This happens for finished and unfinished changes.

3.2.2.5 Staging push

When the developers complete their code and it is ready to be merged into the main develop branch, they push it to a temporal staging branch on the remote repository. When the unit tests are successful, the developer(s) create(s) a pull request in order to merge the code.

3.2.2.6 CI test & fast-forward

The continuous integration server hooks to changes on a staging branch and, upon detection, triggers the changed component build in its own environment, runs all test relevant to that module and, should all of them pass, forwards them from the staging branch to the develop branch. If, however, some tests fail, the developer is notified that the push was rejected and that the faulty code needs to be fixed first before the next push request.

3.2.3 System testing

After successful integration tests, system testing is performed to thoroughly test the entire integrated system. This stage performs the tests specified in the Release Specification document, produced during the Feature Planning phase. All identified bugs are reported by issuing tickets, which lead back to the development stage. The system testing stage is led by the Scrum Master and is carried out by a subset of the development team as well as other project participants, specified along with the tests during the Feature Planning phase.

3.2.4 System release

The release of the system is carried out once the system testing phase has been successful. The next release cycle can then commence.

3.3 Scheduling and events

The system release cycle duration, as well as the overall scheduling and events, depends on the characteristics of the symbIoTe project. Namely:

- symbIoTe has a large development team with approximately 20 developers;
- symbIoTe has a distributed development team with developers from 12 contributing partners across Europe.

Due to this, high communication and coordination overheads are expected, which have to be kept as low as possible. In turn, the envisioned collaboration between all members needs to provide the flexibility to establish the necessary communication channels, though ensuring the accomplishment of the set objectives of the related tasks. As a result, the consortium has decided to follow a system release cycle of 3 months. This duration ensures four releases within each year of the project and allows sufficient time for the large development team to communicate and coordinate. Such a long release cycle is in contrast with many incarnations of the Agile methodology, which target for shorter release cycles in the context of smaller development teams e.g., 4-6 members. During the course of each release cycle, the following conference calls will be scheduled to ensure the close monitoring of the overall process: the resolution of emerging problems and the progress of the overall implementation tasks.

3.3.1 Weekly conference calls (max 30 min)

The entire Developer Team and the Scrum Master will participate in these conference calls. Their maximum duration will be 30 minutes. Their purpose is to ensure team alignment and the establishment of a communication channel for the consistent reporting of achieved progress and emerging problems. Based on these calls, all participants have an updated view of the current status of the overall process. However, these weekly calls are not intended to engage in problem solving (see also next). The minutes of these conference calls will be put on Confluence, so that developers who have not been present in the call can quickly review the most important decisions.

3.3.2 Monthly conference calls (max 60 min)

The Developer Team, the Scrum Master and the Product Owner will participate in these conference calls. Their maximum duration will be 60 minutes. Their purpose is to keep track of development progress in view of the broader goals of the feature planning, as well as to identify major obstacles that need action. Feedback from the Scrum Master and the Developer Team will be given to the Product Owner, including implications to the architectural design. As for the weekly conference calls, minutes will be kept on Confluence.

3.3.3 Ad-hoc conference calls for problem solving

The purpose of these calls is to get into in-depth technical discussions to solve problems identified during the development phase. The Scrum Master and the members of the development team that are affected by the identified problem will participate in these conference calls. The Scrum Master is responsible for identifying the need for an ad-hoc conference call, as well as organizing it and announcing important decisions via Confluence.

3.4 Reports

Three types of reports will support the overall system release. These reports will be used internally to support the development and management of symbIoTe and will be uploaded to confluence. They are described below.

3.4.1 Release specification report

The purpose of this report is to describe the feature planning stage. This includes a detailed specification of the release features, which is prepared by the Product Owner with support of the Scrum Master, whose role is to shed light on aspects influencing implementation decisions. The specification of the functionality is accompanied with a detailed specification of system tests, which will later on provide guidelines for the System Testing phase. The Scrum Master, together with the Product Owner, is responsible for describing the specification of the system tests. These tests will be targeted to capture all intended functional aspects. Additional support may be provided by other members, e.g. contributors to the tasks T5.3, T5.4 and T5.5.

3.4.2 Component Release report

This report aims to support the consistent documentation of the implementation process, providing detailed information about each implemented component including the supported features, open issues, changes compared to previous versions, unit tests and existing bugs. Component release reports are prepared by the corresponding individual members of the development team and aim at providing detailed up-to-date information about a particular component. The selected feature planning and issue-tracking tool (see Section 4.2) will be used for the preparation of the component release reports in a systematic way.

3.4.3 System Release report

This report captures the results of the completed system release cycle, including a detailed report on the developed features, the system wide changes applied, tests performed and their results, remaining open issues and unresolved bugs. The System Release Report aims at providing a thorough overview of the entire release, once major obstacles or bugs have been overcome.

3.5 Software Quality Assurance

Software Quality Assurance (SQA) defines the set of quality standards and procedures that software components developed during the symbIoTe project MUST adhere to. The aim of defining and monitoring SQA is to produce production quality software. To this end, following the system release iteration cycle (see Section 3.2) and reporting (see Section 3.4) procedures, the produced software MUST comply with the quality criteria principles described below.

3.5.1 Coding Style

Coding style identifies good practices and provides guidelines that allow easier maintainability of the code. Style guidelines MUST be proposed and defined by the software development team for each of the components. The used style guidelines MUST

be consistent with the well-known conventions for the programming language used for component implementation[1]. Compliance to coding conventions MUST be automatically validated by the tools for code checking during the implementation process.

3.5.2 Unit Testing

Unit testing allows the detection of failures in changed functions or areas of the code at an early stage. Minimum acceptable code coverage SHOULD be 70% for the code developed under the symbIoTe project. Mock objects[2] MAY be used in the implementation of unit tests. Automated tests MUST be carried out by the use of unit testing frameworks triggered by Continuous Integration tool job definitions (see Section 4.5). The Development Teams MUST implement the required job definitions.

3.5.3 Integration Testing

Integration testing evaluates if the various developed components interact correctly in real scenarios (individual units are combined and tested as a group). It MUST check that operations with other components are functioning. Components MUST pass all the integration tests defined. Such tests are triggered automatically by continuous integration server for all components which had their dependencies updated.

3.5.4 System Testing

Software testing verifies a complete, integrated system. The purpose of this test is to evaluate the system's compliance with the specified requirements. Test suites MUST be defined and monitored.

3.5.5 Performance Testing

To validate the non-functional system requirements, particularly the given performance and latency aspects, the system components will be tested accordingly in the performance test phase before releasing a new version. There, a strong focus is laid on guaranteeing defined load and response requirements of the entire System Under Test. To generate the load patterns and capture the performance metrics, well-known test harness tools such as Apache JMeter[3] MUST be used.

3.5.6 Documentation

Development teams MUST provide a thorough documentation about all the software components being released under the symbIoTe project scope. This documentation will be stored in Confluence. The lifecycle of producing new documentation for any component SHOULD follow the source code development process. Documentation can be different (developer documentation, deployment and administration configuration, installation and configuration guides) depending on the target audience and MAY vary by component. Documentation MUST be versioned and have references to the licences specified in the project.

[1] For example, the following is applicable for Java: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
[2] https://en.wikipedia.org/wiki/Mock_object
[3] http://jmeter.apache.org

3.6 Security Coding Guidelines

Several studies have found that fixing defects in software at an early stage can save a lot of time down the road. In the special case of symbIoTe, preventing bugs from appearing is even more important, especially if the defects can compromise the security of sensitive information processed by the system or allow intruders to access protected information.

Defining rules, which ensure that proper security measures are used, contributes to the maintenance of the quality and security of the software. Several standards exist regarding this matter. Open Web Application Security Project (OWASP) Application Security Verification Standard (AVSV) is one of them and is actively developed in an open way. The most recent version at the time of writing is 3.0.1a[4]. It defines three security verification levels: level 1 is meant for all software, level 2 for applications that contain sensitive data and level 3 for the most critical applications. Precise verification steps are grouped in 19 lists (V1-V19) displaying an increasing level of security as the above-mentioned levels rise. Each list is finalized with a references section, which contains links to instructions, cheat sheets, and descriptions of mentioned technologies.

ASVS Level 2 is proposed to be achieved in symbIoTe development. Fulfilling its requirements ensures the protection against OWASP Top Ten[5] vulnerabilities and beyond, which also includes several items for proper design and documentation of the project. It is meant to be used:

- as a security architecture guidance;
- by the developers when implementing the project;
- by the security task team when testing the resulting components and system.

For Java additional guidelines that help prevent the introduction of security bugs should be followed - Java Coding Guidelines[6] from Software Engineering Institute at Carnegie Mellon University. In conformance to the recommendation “Rec.: Priority and Levels”[7] rules with priorities P12-P27 must, with P6-P9 should, and with P1-P4 may be enforced.

All resources (including source code) in the project must be UTF-8 encoded to prevent bugs and problems for the international development teams.

[4] https://github.com/OWASP/ASVS - PDF and DOC versions of v. 3.0.1a
[5] https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013
[6] https://www.securecoding.cert.org/confluence/display/java/Java+Coding+Guidelines
[7] https://www.securecoding.cert.org/confluence/display/java/Rec.%3A+Priority+and+Levels

4 Implementation Framework Tools

4.1 Tool Chain Overview

The system releases iteration process described in section 3.2 is supported by a set of tools outlined in the following table:

	Feature planning & issue tracking	Version control	Code repository	Continuous integration	Artifacts repository	Building tool
Tool	Atlassian JIRA	Git	github.com	Travis	Bintray	Gradle

Table 3 – Overview of tools assigned to specific tasks of the system release process.

The JIRA tool will support the management of the implementation process, allowing the assignment of features to developers and the subsequent tracking of the development process. The Scrum Master is responsible for managing this tool with the collaboration of the development team members who will accept assigned features and report their progress.

A remote repository (already set up at https://github.com/symbiote-h2020) will be used for code hosting and versioning, as a common point of reference for the development team. Project members will be able to download the latest versions, update changes from other members, implement their features locally and finally commit changes back. The Git tool enables developers to keep precise track of the changes and versions of the code, within and across releases, to merge their changes with other members as well as to identify and resolve potential conflicts. The whole process will follow a particular branching model, described in Section 4.3.1.

When developers commit code changes, automatic build and testing of the source code with the Travis Continuous Integration server will be triggered. This operation allows obvious bugs and component interoperability issues to be instantly discovered and fixed. This setup will support the integration phase of the system release iteration (see Section 3.2.3). The building of the system is handled by a building tool, whose purpose is to alleviate programming concerns (e.g. dependency handling) and to enhance the performance of the build process (e.g. incremental builds, build caching). If the push to the remote repository successfully passes the tests on the continuous integration server, then test results, namely artefacts, will be uploaded to a repository. The following sections provide a description of those tools and how they can help the software developer.

4.2 Feature planning & Issue Tracking

Atlassian JIRA[8] is a software implementation management and issue-tracking tool used to organize and plan the development work. The Product Owner, the Scrum Master and the developers use JIRA to design, describe and group together all requirements into a

[8] https://www.atlassian.com/software/jira

Product Backlog. The planned features and corresponding development requirements (User stories) can be ranked in order of priority and importance. Once ranked, the stories are assigned to developmental cycles.

The board feature of JIRA then allows the tracking of the entire process, avoiding the need for detailed, off-line documentation efforts by the development team. This way the Scrum Master can monitor the entire symbIoTe development process from one place.

JIRA main features consist of:

- Software implementation management and planning, roadmap of milestones and future versions of the software;
- Integration with GitHub, allowing tracking commits, monitoring source code edits, and drilling through to source files;
- Ticketing, where developers are provided with access to JIRA by the JIRA administrator and can view their tickets, milestones and other information from their web browser, with the purpose of tracking and managing bugs and features.

4.3 Version Control

Currently the most popular version control tools are Git and SVN. As opposed to SVN, Git offers a series of features that make it advantageous for inherently distributed projects such as symbIoTe. To list a few[9]:

- Git is a distributed system, allowing each developer to have a full local copy of the entire repository, including history, which allows access to history in extremely fast manner;
- Allowing a full local copy (clone) of the repository, Git allows full functionality when disconnected from the network;
- Each developer has a complete backup of the repository, increasing resilience;
- Git provides better support for branching, including support for multiple local branches.

symbIoTe developers will use Git as a version control system in order to create multiple repositories to hold code for particular symbIoTe modules. Moreover, within those repositories, they create local branches that can be entirely independent of each other. The creation, merging, and deletion of those lines of development can be conducted very easily. Furthermore, with simple commands, developers can create requests to the main remote Github repository, in which all symbIoTe developers commit their code.

4.3.1 Branching model

During our implementation, we will follow the Gitflow branching model[10], which makes use of extensive branching to facilitate the parallel development of features, preparation for product releases and bug fixes. This model includes two different types of branches; the **main** branches and the **supporting** branches.

[9] A thorough discussion can be found here: https://git.wiki.kernel.org/index.php/GitSvnComparsion
[10] http://nvie.com/posts/a-successful-git-branching-model/

The symbIoTe organisation profile at Github will expose multi-tiered repositories. For each coherent module, a separate repository to log the development process will be created. Moreover, a central symbIoTe repository will be created which, instead of holding actual code, will utilise the git submodules feature to easily track and coordinate global system releases.

Each repository will have at least two main branches; the **master** and the **develop** branch. These branches have an unlimited lifetime. We consider the **master** branch to be the main branch where the source code of HEAD always reflects a **production-ready** state. On the other hand, we consider the **develop** branch to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. This is where any automatic *nightly build* is built from.

Apart from the main branches, our symbIoTe implementation framework also uses three types of supporting branches; **feature**, **release** and **hotfix** branches. The main difference compared to the main branches is that these branches always have a limited lifetime, since they will be eventually removed.

A **feature** branch is created from the develop branch each time a new feature needs to be developed. The essence of this type of branch is to support parallel development of features among teams and ease the tracking of features. Eventually, it will either merge back into the develop branch upon successful implementation of the feature or be discarded after disappointing experimentation.

Release branches are created from the develop branch to facilitate the final preparation of a new product release, when all the desired features of the next release have been integrated. In principle, only minor bug fixes and code fine-tuning (e.g. adding comments or metadata) take place in this type of branch. Ultimately, the release branch will merge back into the master branch to mark a new product release. This approach allows new features to be added to the develop branch, while the new product release is being prepared. Naturally, the release branch is also merged with the develop branch to update it with the last minute changes.

When a critical bug emerges, a **hotfix** branch is created from the master branch to tackle the problem. As soon as the bug is fixed, the hotfix branch is merged with the master to indicate a new minor product release. It is also merged with the develop branch to ensure proper functionality of future releases. By following this strategy, the work of other teams in new features is not halted while the bug is being fixed.

Moreover, a temporary **staging** branch will appear every so often on the remote repository. The **develop** branch is locked from being updated by the developers. Upon being done with their tasks developers will be required to push their changes from the **feature** branch into a temporary **staging** branch. The continuous integration server hooks to changes on the **staging** branch and upon detection triggers the changed component build in its own environment, runs all test relevant to that module and should all of them pass, forwards them from **staging** branch to the **develop** branch. Should however some tests fail, then the developer is notified that he/she wanted to contribute faulty code, the push was rejected and needs to be fixed prior to arrival at main **develop** branch.

4.4 Code repository

The project uses GitHub's public repositories and free accounts[11] to host symbIoTe's open-source software project. GitHub is a Git repository-hosting environment. It includes all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. GitHub provides a web-based graphical interface and desktop integration. It also provides access control and several collaboration features such as bug tracking, feature requests and task management for the symbIoTe development process.

The main features of GitHub are:

- Documentation, including automatically-rendered README files in a variety of Markdown-like file formats;
- Issue tracking (including feature requests) with labels, milestones, assignees and a search engine;
- Pull requests with code review and comments;
- Commits history;
- Graphs: pulse, contributors, commits, code frequency, punch card, network, members;
- Integrations Directory;
- Unified and split diffs;
- Email notifications;
- Option to subscribe someone to notifications by @ mentioning them;
- Indication of build status and test coverage deriving from Travis CI.

4.5 Continuous Integration

Currently, the most popular continuous integration (CI) tools are Travis CI and Jenkins. symbIoTe will use Travis CI because of the low configuration and maintenance overheads. Travis CI is easy to setup and maintain, only by including a configuration file (*travis.yml*) in the correct code branch. There can be a separate configuration file for each branch. It is a cloud-hosted solution, so there is no maintenance overhead. Jenkins, on the other side, has to be deployed, set up and maintained at the project side, adding maintenance/management overheads.

Travis, as the solution selected for symbIoTe, will be used to automatically and frequently build the symbIoTe software. It has a smooth integration with GitHub in order to support the instant automatic software build and test. The Travis CI server is notified by GitHub, whenever new commits are pushed to that repository or a pull request is submitted from the development team for a particular repository.

Bellow we list some of Travis' main features

- Cloud instance offering in order to support open source community. All built history is provided with no restriction at Travis public place. Commit changes, timestamp, author of commit, built indication (success / failure) are provided in a nice interface;
- Raw log txt file is provided for every build;

[11] https://github.com

- Integration with GitHub: All artifacts needed to build the system are placed in the code repository and the system will be built upon every new commit that is pushed to that repository or any new pull request that is submitted;
- Build automation: Source code compiling, binary code packaging, deployment and documentation generation can be automatically performed by calling single commands in existing build scripts;
- Can be configured to only run for specific branches, or branches whose names match a specific pattern;
- Automated Unit tests whenever developers commit code to the repository;
- E-mail notification and feedback to responsible developers whenever new commits are pushed to that repository or a pull request is submitted;
- Continuous integration and testing statistics;
- Support various programming languages like: C, C#, C++, F#, Groovy, Java, JavaScript (with Node.js), Perl, Perl6, PHP, Python, R, Ruby, Scala. This is particular important for the unified management of various components of the symbIoTe architecture across compliance levels;
- Integration with Coverity tool: run static analysis on each and every commit of symbIoTe project.

4.5.1 Artifacts repository - Bintray

An integral part of a CI environment is a repository serving build artifacts. In a modular system such as symbIoTe we expect to produce at least a few dependencies on each other modules (Java jars). Those are the result of each successful build done on Travis and are published to the repository so that there is always a built module available to serve as a dependency for other modules using maven/ivy dependency model which Gradle ingests without problems. A viable cloud hosting service, which works well with Travis, is the Bintray.com repository.

4.6 Building Tools

In order to facilitate the implementation process and improve the quality of the delivered products, an automated build tool is required. Apart from being a prerequisite for continuous integration and testing, automating the build process can also alleviate programming concerns (e.g. dependency handling), enhance the performance of the build process (e.g. incremental builds, build caching) and provide useful feedback. There are a variety of build tools available, however, since our project will be based mainly on Java, we only consider Ant, Maven and Gradle, which are the most prominent Java-automated build tools.

4.6.1 Ant + Ivy

Ant was released in 2000 and became the pioneer of modern automated build tools. It used to be the most popular build tool for Java projects, until it was later surpassed by more sophisticated tools like *Maven* and *Gradle*. It uses XML format for build scripts, which is not an ideal match for the procedural programming idea it is based on and may lead to extremely large build scripts. Moreover, it adopted *Ivy* to provide dependency management. Its main strengths are the low learning curve and the great support for developer tools like IDEs, Application and Continuous Integration Servers. However,

creating and maintaining build scripts might get tricky, especially when the XML scripts grow large in size and customizations are needed. Furthermore, it also lacks in build speed, documentation and community support compared to more modern tools, like *Maven* and *Gradle*.

4.6.2 Maven

Maven was introduced in 2004, aiming to tackle some drawbacks of Ant, and is currently the most popular choice for automating the Java build process. Maven also uses XML for writing scripts, but in contrast to *Ant* it relies on conventions and provides targets available for invocation. Furthermore, it offers full support for all the popular developer tools and a wide variety of available plugins. It is also fairly simple to write your own plugin to extend the framework. Another significant improvement is its build speed, which greatly surpasses *Ant*. However, although it also has a low learning curve and provides fairly good documentation, customizing build scripts and dealing with multi-module projects might become challenging and often even more complex than in Ant.

4.6.3 Gradle

Gradle emerged in 2012, built on *Ant* and *Maven*, but introduced a Groovy-based Domain Specific Language (DSL) to replace the XML format used by its ancestors. Gradle has the best learning curve, it is supported by a highly active community and it offers excellent documentation. In terms of build speed, it is well matched with *Maven*. Due to its recent emergence, it falls a little behind in terms of plugins availability and integration with developer tools compared to *Ant* and *Maven*. However, it supports a fair variety of developer tools and features a simple and well-documented process for creating plugins. Furthermore, creating and maintaining build scripts is much simpler and the scripts are significantly shorter in size, due to the use of DSL.

5 Technology Scouting

In this section, technologies relevant to the development of the symbIoTe project are listed. Participating partners are very familiar with some of the technology components, while others were identified in the context of this project. This list serves as a basis from which the technologies relevant to be integrated into the project will be chosen.

5.1 Django 1.9.5

Django[12] is a free, open-source web framework written in Python. It provides a set of components common for this type of job that allow fast and easy website development. For the particular case of symbIoTe, Django might function as a general framework for creating and managing the Resource Access Proxy (RAP) and the Authentication and Authorization Manager (AAM). Django shall allow Ubiwhere's Mobility Backend as a Service (MBaaS) platform to be symbIoTe-compliant, which enables the resources provided by the platform to be used by the Smart Mobility and Ecological Routing Enabler through symbIoTe's semantics.

5.2 OpenWSN 1.9.0

OpenWSN[13] is a free and open-source operating system for the IoT, implementing a complete protocol suite based on Institute of Electrical and Electronics Engineers (IEEE) and Internet Engineering Task Force (IETF) open standards. Specifically, it integrates PHY and Media Access Control (MAC) layers of the IEEE802.15.4-2015 technology, the IETF 6TOP adaptation layer recently standardized by the IETF 6tisch working group, as well as 6LoWPAN adaptation layer, RPL routing protocol, and CoAP application layer. It also supports MAC-layer security functionalities as described by IEEE 802.15.4-2015 standard [2][3]. OpenWSN runs on a variety of hardware devices, such as Memsic TelosB, Zolertia Z1 and OpenMote-CC2538, and supports both Windows and Linux OS. Moreover, it also offers Gateway functions between the external network and the smart devices.

Thanks to all these features, it represents one of the most attractive solutions for Smart Devices in the context of the Industrial IoT (IIoT).

5.3 Charm_Crypto 0.42

Charm-Crypto[14] is a framework for the quick and easy prototyping of advanced cryptosystems. Based on the Python language, it provides a wide set of cryptographic primitives ready for use or to be further customized. Charm-Crypto intrinsically supports specific libraries for handling complex cryptography techniques and solutions, like Attribute Based Encryption (ABE) and Attribute Based Access Control (ABAC) techniques.

For the specific case of symbIoTe, Charm-Crypto can be used by AAM and RAP modules for generating, validating, and processing access tokens.

[12] https://www.djangoproject.com
[13] https://github.com/openwsn-berkeley/openwsn-fw
[14] http://jhuisi.github.io/charm/install_source.html

5.4 Macaroons

Macaroons [4] are flexible authorization credentials, tailored for Cloud Services that support decentralized delegation between principals. Macaroons are based on nested chained Message Authentication Codes (as keyed-Hash Message Authentication Codes (HMAC)), in a way that they are highly efficient, easy to deploy and widely applicable. They embed an inherent support for delegation, attenuation with contextual caveats and asynchronous token revocation. In the context of symbIoTe, Macaroons can be successfully used for token generation, revocation and handling both at the AAM and at the RAP.

5.5 JSON Web Tokens

JSON Web Token (JWT) [5] represents a compact instrument for transferring trusted information between two parties, already standardized in RFC7519. Information (also namely claim) is encoded as a JSON object. JWT includes a Message Authentication Code or a Signature that allows verifying its authenticity and integrity through symmetric or asymmetric cryptography techniques. The reference RFC defines a wide set of standardized claims, that includes the “issuer” of the token, the “subject” of the token, details on the time validity (as “issued at” and “expiration time” claims). Further customized caveats for specific needs are allowed as well.

Within the symbIoTe project, JWT is an interesting alternative with respect to Macaroons. In fact, they can be used for carrying trusted access permissions (i.e., attributes), useful to support ABAC techniques. JWT can be generated and revoked by AAM, as well as validated and processed by AAM and RAP modules.

5.6 Netflix OSS

As the symbIoTe software components are designed and developed to feature easy distribution, resilience and horizontal scalability, the targeted runtime environment also has to deal with such features. Furthermore, advanced and easy to use deployment and management tools should be provided out of the box. The Netflix Open Source Software (OSS) group provides a set of open source projects that deal with these runtime requirements.

Amongst others, the following projects may be relevant for symbIoTe:

- The Hystrix[15] middleware abstracts tight service dependencies within a distributed environment by isolating the (remote) access points. The external service calls are wrapped in a Hystrix command object where service failures can be detected and managed easily by avoiding cascading failures and fallback logic options;
- Eureka[16] is a RESTful service for managing and locating services within a distributed environment. Eureka provides advanced load balancing (basic round-robin strategy) and failover features for middle tier servers and therefore could ease the deployment and management within the symbIoTe ecosystem;

[15] https://github.com/Netflix/Hystrix
[16] https://github.com/Netflix/eureka

- Feign[17] eases the creation and handling of text-based HTTP API calls by introducing templates based on code annotations. The JAVA library seamlessly integrates the Netflix Denominator[18] library for managing DNS cloud environments and additionally reduces the complexity within a distributed environment. In symbIoTe this library could reduce the effort of handling service discovery and lookup of multiple service instances;
- Ribbon[19] abstracts the creation of REST calls by adding template builder functionalities with built-in client-side fallback hooks and response validation schemes. Ribbon also supports multiple endpoints and protocols such as HTTP, TCP or UDP. Ribbon could be used in symbIoTe to facilitate scaling and resilient communication behaviour without any other dedicated load balancing service;
- Zuul[20] provides a gateway service with dynamic routing and monitoring capabilities. symbIoTe could use Zuul as central gate keeper for all requests where several cross cutting concerns such as monitoring, logging or security aspects (e.g. anomaly detection, authentication) could be handled.

5.7 Spring Cloud

Given the possibility of symbIoTe having a microservices architecture, the systems need support for management, configuration and devops analysis tools e.g. provided as Netflix OSS projects (see Section 5.6). Spring Cloud offers seamless integration of its own solutions from the Spring stack (e.g. Framework & Boot) with those tools. This is a complete solution, which provides features of modern software on all layers:

- starting with the lowest level of e.g. inversion-of-control principle for objects lifecycle management;
- building web services upon that core with Spring Framework;
- simplifying development, configuration and deployment of one's code with Spring Boot;
- and finally by adapting those solutions to work in distributed cloud environments.

As such, most of the components released by Netflix under OSS project are already integrated with Spring Boot to provide a scalable and configurable microservices-powered system and offered as Spring Cloud.

Spring framework and Spring Boot have large and active communities that contribute through blogs, videos, conferences, tutorials, MOOCs, tweets, stack overflow and other means.

All this factors are relevant to symbIoTe, facilitating the development process with a tool that also has great community support.

We will now mention a few features from Spring Cloud which empower this cloud-apps building suite.

[17] https://github.com/OpenFeign/feign
[18] https://github.com/Netflix/Denominator
[19] https://github.com/Netflix/ribbon
[20] https://github.com/Netflix/zuul

5.7.1 Spring framework

The Spring Framework[21] is the leading Open Source Java platform for the development of enterprise ready Java applications. Among others, it provides the tools and helpers needed to:

- Isolate deployment issues between application servers, making the application deployable in a wide array of vendor solutions both proprietary and Open Source;
- Provide tools to implement different programming patterns like, e.g. Singleton or Publish-Subscribe;
- Isolate and standardize access to different data sources, be it local or remote;
- Provide a powerful framework for bean registration and dependency injection;
- Implement mechanisms for Aspect Oriented Programming.

As a basic framework, it's not tied to any concrete component of symbIoTe in particular. It's role will be to be the foundation upon which the different components implemented in Java will be built, providing the tools and helpers needed to forget about deployment and wiring issues and concentrate in business logic instead.

5.7.2 Spring Boot

Spring Boot[22] allows developers to create Spring-based applications easily. It does so by providing a default configuration, from which the project can be expanded. It requires no XML configuration, one of the biggest burdens of the Spring framework. Spring Boot also provides non-functional features that are common in this kind of large projects, such as embedded servers, metrics, logging, security, health checks and others.

5.8 SLA-Framework

The SLA-Framework is a Java based tool, developed by Atos, able to manage templates, agreements & violations with all documents based on WS_Agreement[23] standard. Usually the workflow of an SLA is the following:

1. A Provider offers a Service: a software service, hardware resources, etc;
2. The service is described by ServiceDescriptionTerms (WS-Agreement concept) with a Domain Specific Language. The ServiceDescriptionTerms are intended to define a service that has to be provisioned;
3. The service offering is represented by a Template "document", which can be used to generate an Agreement;
4. An Agreement is a "document" that associates a Service and a Consumer. It is a Contract;
5. A Template and an Agreement can describe some restrictions (Guarantee Terms) to fulfilled by the Service Provider;
6. The not fulfilment of any restriction generates a Violation.

[21] https://projects.spring.io/spring-framework/
[22] http://projects.spring.io/spring-boot/
[23] https://www.ogf.org/documents/GFD.107.pdf

The SLA-Framework is able to manage template and agreement documents in WS-Agreement format. Plugins can be programmed to customize the input format of a template/agreement that internally will be converted to WS-Agreement standard.

The enforcement is the process by which it is evaluated if the conditions of an agreement are fulfilled, i.e. the measured metrics for the variables in guarantee terms fulfil the constraints. The SLA-Framework is able to pull the metrics from 3rd party systems, evaluate the constraints, and raise the appropriate violations.

We propose to use the SLA-Framework as Federation Manager (SLA) for symbIoTe. A platform owner is able to specify its offerings. When an enabler/application or other platform wants to use the service it is possible to create and agreement based on the offering. The SLA-Framework will be able to evaluate if the agreement is fulfilled or not. If the evaluation is very complex a plugin can be programmed and integrated into the SLA-Framework. A plugin must be programmed in order to retrieve the data from the external monitoring system. The SLA-Framework is already prepared to integrate easily the Java based plugins.

Some other data regarding SLA-Framework:

- Java Based;
- RESTful interface;
- Internally data is recorded in WS-Agreement format;
- Data can be sent/received in JSON/XML;
- Stores data in MySQL database;
- Runs with Windows & Linux;
- Features can be customized via Java plugins;
- Runs behind a Tomcat or any other application server.

5.9 OpenIoT

The OpenIoT[24] (Open Source blueprint for large scale self-organizing cloud environments) middleware is an open source IoT platform for the cloud developed within the framework of the FP7 project OpenIoT. The middleware supports dynamical composition of IoT services driven by the data gathered and processed from integrated heterogeneous data sources. It is written in Java and the middleware flexibility is supported by Semantic Web technologies. In the symbIoTe project, the OpenIoT middleware is involved in two use cases (Smart Mobility and Ecological Routing and Smart Campus), which serve as underlying IoT platforms used by cross-platform applications. It will also serve as a proof-of-concept of the implementation of the symbIoTe Level 1 compliance, i.e. the symbIoTe extension of the middleware will be published as an open source code so that other IoT platforms willing to join the symbIoTe ecosystem can reuse the same approach. Additionally, it will participate in the symbIoTe platform federation mainly through the symbIoTe Level 2 compliance.

[24] https://github.com/OpenIotOrg/openiot

5.10 Eclipse OM2M

Eclipse OM2M[25] is an open source project that provides a horizontal M2M service platform that allows the development of services independently of the underlying network.

Developed at the Eclipse Foundation, it is compliant with the oneM2M and the ETSI SmartM2M standard. Its generic RESTful interface eases the deployment of vertical applications. Its OSGI-based architecture can also be extended with pulgins.

Eclipse OM2M is relevant to symbIoTe since the symbIoTe architecture is motivated by the oneM2M functional architecture. Thus, some of the OM2M design decisions can be useful for the symbIoTe design, while further evaluation is needed whether symbIoTe can reuse OM2M components. Furthermore, OM2M is also a candidate IoT platform to be integrated within the symbIoTe ecosystem.

5.11 OData

OData[26] is a REST-based communication protocol built on the Hypertext transfer protocol (HTTP) defining a standard for data-centric Application Programming Interfaces (API). Originally developed by Microsoft, it provides standards and best practices for performing Create-Read-Update-Delete (CRUD) operations on REST-based resources as well as querying collections of resources. OData is currently in version 4.0 and has been approved as standard by the Organization for the Advancement of Structured Information Standards (OASIS). The OData specification has been implemented by numerous open-source libraries in C#, SAP, PHP and Java. There is also a library for Python, though it does not implement the full standard. Apache OLingo and odata4j are both Java-based implementations for OData. Both libraries offer implementations for server and client (thus JavaScript)-side. In symbIoTe OData could be used for APIs when accessing and discovering resources (thus sensors) provided by symbIoTe-enabled platforms.

5.12 UniversAAL

UniversAAL[27] is an open source IoT platform distributed under the Apache Software License version 2.0 developed within EU project #247950 from 2010 to 2014. It's RDF and OWL ontology based middleware featuring a service oriented bus architecture hiding the heterogeneity of actors and sensors from different vendors, thus reducing the complexity of the integration task when facing different components in one IoT smart space.

Despite UniversAAL is being deployed on 13 different pilot sites since 2015 within the ReAAL project[28], it does not seem to be developed very active lately. Latest official releases for different operating systems like Windows, Linux or Android date back to late 2015.

In symbIoTe the UniversAAL platform could ease the integration task of different smart objects within one smart space like a user's smart residence bringing together different

[25] http://www.eclipse.org/om2m/
[26] http://www.odata.org
[27] http://www.universaal.info/
[28] http://www.cip-real.eu/home/

domains like home automation and healthcare. Concrete service and resource discovery could be abstracted by calling out to the UniversAAL middleware enabled smart space.

5.13 SensorThings API

The OGC *SensorThings API*[29] is a HTTP/REST-based communication API and data model for transferring sensor data and metadata designed by the *Open Geospatial Consortium*[30] and accepted as a standard in 2016. It is based on the *OData* protocol and designed to be lightweight so resource-constrained sensors can still use it. The data model defines the entities Thing, Location, HistoricalLocation, Sensor, ObservedProperty, Datastream, Observation and FeatureOfInterest while the API defines how to Create, Read, Update and Delete these entities. There are several open and closed-source server implementations and client implementations for many programming languages.

In symbIoTe the API could be a candidate API for accessing sensor data, while the data model could be used as a core data model.

5.14 Apache Jena

Apache Jena[31] is an open source Java framework for building Semantic Web and Linked Data applications. It offers tools for storing and manipulating RDF data as well as an HTTP interface or triple store called *Fuseki*. It is very similar to *RDF4j* but in addition provides support for the *Web Ontology Language (OWL)* and is published under the Apache License 2.0. Apache Jena supports in-memory as well as persistent storage of RDF data and offers support for execution and manipulation of SPARQL 1.1 queries. The Inference API of Apache Jena allows the inclusion of rule-based reasoning and comes already with multiple reasoners included.

In the context of symbIoTe the Apache Jena framework could be used to store, process and query data in RDF format which will be needed at multiple components, e.g. the symbIoTe Core.

5.15 Alignment API

The *Alignment API*[32] is an API and implementation for expressing and sharing ontology alignments. It includes a data format in the form of an ontology for expressing alignments, i.e. correspondences between entities and ontologies in a uniform way. The format is expressed in RDF and therefore is freely extensible. The extension *Expressive and Declarative Ontology Alignment Language (EDOAL)*[33] allows the representation of complex correspondences between entities, including transformation and value conversion, and is therefore very well suited to express sophisticated ontology mappings.

[29] https://github.com/opengeospatial/sensorthings
[30] http://www.opengeospatial.org/
[31] https://jena.apache.org/
[32] http://alignapi.gforge.inria.fr/
[33] http://alignapi.gforge.inria.fr/edoal.html

In symbIoTe *EDOAL* could be used to express mappings between two different platform-specific extensions of the core information model. This mapping definitions could be used to enable querying of available sensors, actuators and services offered by different platforms in a unified way.

5.16 Search Tools

One of the basic functionalities of symbIoTe is the support for discovery of IoT resources registered by individual IoT platforms, enablers and *prosumers*. This functionality will rely on the capability of the symbIoTe Core to search within the meta-data of existing registrations. As the envisioned symbIoTe ecosystem includes a multitude of registered resources, and a corresponding volume of meta-data, the search capabilities of symbIoTe are subsequently required to provide high scalability with respect to the response times on submitted queries, as well as the accuracy of the results.

In the following, we survey a series of well-known search tools/frameworks and assess their fit to symbIoTe. In our survey we pay particular attention to the support for search in data represented in XML/RDF, which is the model selected by symbIoTe with the purpose of enabling a flexible and expressive representation of the registration meta-data.

5.16.1 Elasticsearch

Elasticsearch[34] is a powerful open source search and analytics engine based on *Apache Lucene*[35], which is widely regarded as the most advanced search engine library today. *Elasticsearch* abstracts away some of the complexity introduced by *Apache Lucene* and combines *full-text search* with distributed and scalable real-time document store and analytics. Among its advanced features are the dealing with human language, geolocation and relationships, as well as the ranking of documents by relevance. It is implemented in Java and it is available under the *Apache 2.0* license. The *symbIoTe Search Engine* could leverage some of the advanced features offered by *Elasticsearch* to polish the matching procedure of the client/application needs, to available virtual resources. For example, it could be used to empower the *Search Engine* with full-text search capabilities or to rank the search results by proximity to our desired location. Unfortunately, *Elasticsearch* uses JSON format to represent the data, in contrast with the RDF format proposed by *symbioTe*. Previous versions of *Elasticsearch* supported RDF format via the use of River plugins. However, River plugins have been deprecated in the most recent versions for the sake of cluster stability.

5.16.2 Rdf4j

Rdf4j[36], previously known as *Sesame*, is a framework used to store and analyse RDF data. It provides both memory-based and disk-based storage as well as two distinct Servlet packages allowing the access and management of triplestores on a remote server. *Rdf4j* is implemented in Java and is available under a *BSD-style* license. Furthermore, it includes Sesame Rio, which contains a set for Java-based writers and parsers, and it supports two query languages; *SPARQL*, an RDF query language which has been

[34] https://www.elastic.co/
[35] https://lucene.apache.org/
[36] http://rdf4j.org/

elevated to one of the key technologies of semantic web and *SeRQL*, which is Sesame's proposal for querying RDF data. Moreover, the stackable interface, which can be used to add functionality, and the decoupling of its storage engine from the query interface sets it apart from other competitive solutions. *Rdf4j* can be leveraged by *symbIoTe Registry* and *Search Engine* to facilitate the storage, analysis and querying of data in RDF format, which is currently the prevailing proposal for *symbioTe*. It can also provide advanced features, like *full-text search* through *LuceseSail* and *Geospatial search* by using the *USeekM*[37] library. On the downside, *Rdf4j* lacks scalability capabilities, like partitioning and replicating data on different nodes.

5.16.3 Virtuoso

Virtuoso[38] is a multi-model hybrid database engine, which combines a variety of different functionalities, such as a *Relational Database Management System* (RDBMS), an *Object Relational Database Management System* (ORDBMS), RDF stores, file and web application servers in a single system. Instead of using distinct servers for providing the above services, it acts like a *"Universal Server"* with a multithreaded process implementing multiple protocols. It supports both physical and in-memory storage and uses SPARQL for querying triplestores. Although it is implemented in C, it supports many programming languages like Java, PHP, Ruby and Python and it is available under the *GPLv2* license. *Virtuoso* can be adopted by *symbIoTe* to provide storage and querying functionalities of RDF data, needed by *symbIoTe Registry* and *Search Engine* respectively. Furthermore, it can also provide the necessary infrastructure to host any web application implemented during the course of the symbIoTe project. *Virtuoso* could also enhance the scalability of *symbIoTe*, as it offers partitioning and replication techniques. Advanced features like *full-text* and *geospatial* search are also supported by *Virtuoso*, making it capable of further enhancing the capabilities of the *Search Engine*.

5.17 Anomaly detection tools

An additional security functionality of symbIoTe could be an anomaly detection implementation. This functionality would rely on the capability of analyzing registered meta-data about connections and other events in the system. Such analysis outside the main system minimizes utilization of symbIoTe resources and ensures transparency of this process. Mechanisms to analyze these data and discover anomalies could include advanced machine learning techniques like artificial neural networks, and fast pre-processing technologies like used in the complex event processing scenarios to prevent unnecessary overhead.

Anomaly detection is currently the only available method that allows the detection of unknown threats (that do not have any signature that could allow their identification)[39]. Moreover, in the IoT environment, due to the diversity of the used devices as well as their heterogeneous interfaces and potential difficulties in management, updating these devices may be significantly difficult. Therefore, it seems reasonable to introduce another security

[37] https://dev.opensahara.com/projects/useekm
[38] http://virtuoso.openlinksw.com/
[39] A. Kliarsky, A. Atlasis, "Responding to Zero Day Threats", SANS Institute, 2011, s. 7-8, http://www.sans.org/reading-room/whitepapers/incident/respondingzero-day-threats-33709

solution, most likely in the network layer, able to detect malicious traffic or behaviour, originating from or to the devices, generally.

5.17.1 Encog

Encog[40] is a Java-based neural network framework with plenty of algorithms implemented, GUI and rich documentation (including lessons about artificial intelligence).

It can be trained to remember the normal state, like the structure or statistics of events, and later be used to discover anomalies.

5.17.2 Siddhi CEP

Siddhi[41] is a Complex Event Processing engine with SQL-like query language. It supports the definition of complex rules applicable to event streams.

It can be used to discover specific situations declared in the query language.

5.17.3 STIX

Structured Threat Information eXpression[42] is a structured language and a set of tools for "cyber threat intelligence". The language is XML-based and supports the description of lots of assets, events and situations.

It can be used to report anomalies in a both human- and machine-readable format.

5.17.4 WSO2 Platform

WSO2[43] is a multi-purpose platform containing complete Complex Event Processing module (based on Siddhi). Other modules include complete GUI, user management, visualization etc. It can communicate with many protocols through adapters and has impressive throughput.

It can be used to quickly deploy CEP system for anomaly detection and provide visualization out-of-the-box.

[40] http://www.heatonresearch.com/encog/
[41] https://github.com/wso2/siddhi
[42] https://stixproject.github.io
[43] http://wso2.com

6 Conclusions

This Deliverable describes the implementation framework to be adopted by the symbIoTe consortium for the implementation of the symbIoTe prototype in a distributed and efficient manner, resulting in a high quality and close to market prototype.

Initially, D5.1 defines the various roles within the project, inspired by the Agile approach for project development. The considered roles within the symbIoTe project are the Product Owner, the Scrum Master, the Component Owners and the Development Team.

The system release iteration, which describes the main implementation workflow, is defined. It follows a cycle of four steps:

- **Feature Planning**, where new features are specified and where system tests that test the components that implement the features are also defined. The various features are then assigned to groups of developers;
- **Development**, which itself is a cycle of various steps: the developers pull the latest stable code, work on their features, test the code locally, push the features to a remote repository in order to have a backup of their work, push their code to a temporal staging branch and, if it passes the unit tests, the developers can create a pull request. The continuous integration server runs all the relevant tests and, if successful, forwards the changes to the develop branch;
- **System testing**, where the entire integrated system is tested;
- **System release**, following successful system testing, the system can be released and a new release cycle can commence.

It is decided that the project will follow a release cycle of three months, four per year, which allows sufficient time for large development teams to communicate and coordinate. To monitor the release cycle, there will be weekly conference calls to ensure team alignment and to report achieved progress and emerging problems. There will also be a monthly conference call, which will track the development progress and identify major obstacles.

Three types of reports are defined that support the system release and are to be used internally: the release specification report, which describes the feature planning phase; the component release report, which supports the consistent documentation of the implementation process by providing information on the implemented components; the system release report, which captures the results of the system release cycle.

The software quality assurance section defines standards and procedures that must be followed with the aim of producing quality software. These include coding style, unit testing, integration testing, system testing, performance testing and documentation.

Finally, coding guidelines are proposed with the aim of preventing bugs and reducing the time developers spend fixing the code. OWASP's Application Security Verification Standard is proposed to ensure proper security measures are used and Java Coding Guidelines from Software Engineering Institute at Carnegie Mellon University is suggested for Java code development.

Various tools are listed that will support the implementation framework: Atlassian JIRA will be used for feature planning and issue tracking; Git will provide version control; github.com

will be used as a code repository; Travis will be used for Continuous Integration; Bintray will be used as an artifact repository; and Gradle will be used as a building tool.

The document ends with the results of the technology scouting, which are presented as a list of potential technologies that could be relevant for the symbIoTe project. They will serve as a basis from which the technologies to be integrated into to the project will be chosen.

7 References

[1] symbIoTe Project Deliverable D9.1 – POPD Requirement No.2; February 2016.

[2] S. Sciancalepore, G. Piro, G. Boggia, and L. A. Grieco**,"** Application of IEEE 802.15.4 security procedures in OpenWSN protocol stack**",** *IEEE Standards Education e-Magazine*, 2, vol. 4, 4th Quarter, 2014.

[3] S. Sciancalepore, M. Vucinic, G. Piro, G. Boggia, and T. Watteyne**,"** Link-layer Security in TSCH networks: effect on slot duration**",** *Transactions on Emerging Telecommunications Technologies (ETT)*, Jun., 2016, to be published.

[4] A. Birgisson, J. Gibbs, M. Vrable, M. Lentczer, “Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud”, Network and Distributed Systems Security Symposium, 2014.

[5] M. Jones, J. Bradley, N. Sakimura, “JSON Web Token (JWT)”, RFC 7519, Proposed Standard, May 2012.

Abbreviations

AAM	Authentication and Authorization Manager
ASVS	Application Security Verification Standard
CI	Continuous Integration
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
JWT	JSON Web Token
MAC	Media Access Control
OWASP	Open Web Application Security Project
RAP	Resource Access Proxy
SQA	Software Quality Assurance

Appendix A: Agile

Agile Project Management Roles

Agile project teams include the following five roles:

Development team: Programmers, testers, designers, writers, and anyone else who has a hands-on role in product development is a member of the development team.

Product owner: The person responsible for bridging the gap between the customer, business stakeholders, and the development team. The product owner is an expert on the product and the customer's needs and priorities. So, this is the one person responsible on a team who is responsible for the prioritized work item list (called a product backlog in Scrum), for making decisions and for providing information in a timely manner. The product owner works with the development team daily to help clarify requirements. The product owner is sometimes called a customer representative.

Team lead (Scrum master): The person responsible for supporting the development team, clearing organizational roadblocks, and keeping the agile process consistent. A scrum master is sometimes called a project facilitator.

Stakeholders: Anyone with an interest in the project. Stakeholders are not ultimately responsible for the product, but they provide input and are affected by the project's outcome. The group of stakeholders is diverse and can include people from different departments, or even different companies.

Agile mentor: Someone who has experience implementing agile projects and can share that experience with a project team. The agile mentor can provide valuable feedback and advice to new project teams and to project teams that want to perform at a higher level.

Agile Project Management Events

Most projects have stages. Agile projects include seven events for product development. These events are meetings and stages and are described in the following list:

Project planning: The initial planning for your project. Project planning includes creating a product vision statement and a product roadmap, and can take place in as little time as one day.

Release planning: Planning the next set of product features to release and identifying an imminent product launch date around which the team can mobilize. On agile projects, you plan one release at a time.

Sprint: A short cycle of development, in which the team creates potentially shippable product functionality. Sprints, sometimes called iterations, typically last between one and four weeks. Sprints can last as little as one day, but should not be longer than four weeks. Sprints should remain the same length throughout the entire projects.

Sprint planning: A meeting at the beginning of each sprint where the scrum team commits to a sprint goal. They also identify the requirements that support this goal and will be part of the sprint, and the individual tasks it will take to complete each requirement.

Daily scrum: A 15-minute meeting held each day in a sprint, where development team members state what they completed the day before, what they will complete on the current day, and whether they have any roadblocks.

Sprint review: A meeting at the end of each sprint, introduced by the product owner, where the development team demonstrates the working product functionality has been completed during the sprint.

Sprint retrospective: A meeting at the end of each sprint where the scrum team discusses what went well, what could change, and how to make any changes.

Agile Project Management Artefacts

Project progress needs to be measurable. Agile project teams often use six main artefacts, or deliverables, to develop products and track progress, as listed here:

Product vision statement: An elevator pitch, or a quick summary, to communicate how your product supports the company's or organization's strategies. The vision statement must articulate the goals for the product.

Product backlog: The full list of what is in the scope for your project, ordered by priority. Once you have your first requirement, you have a product backlog.

Product roadmap: The product roadmap is a high-level view of the product requirements, with a loose time frame for when you will develop those requirements.

Release plan: A high-level timetable for the release of working software.

Sprint backlog: The goal, user stories, and tasks associated with the current sprint.

Increment: The working product functionality at the end of each sprint.