# DELIVERABLE

**Project title:** **LetItFlow: Active Distributed Workflow System for elderly**

**Project reference number:** **AAL-2013-6-128**

## D3.1 – System interfaces definition

**Revision: 1.0**

**Main Authors:**

**Pedro A. Ruiz, Miguel Baizán and Francisco Fornés from INTEGRASYS**

**Ben Loke and Leon Wiertz from Noldus**

**Simona Bica, Victor Carmocanu and Otilia Bularca (reviewer) from SIVECO**

**Georg Regal and Valentin Gattol (reviewer) from AIT**

| Date: [29/02/2016] |
| --- |
| Dissemination Level: Public |

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES



# ABBREVIATIONS

BPM – Business Process Management

BPMN – Business Process Model and Notation

MoSCoW - **M**ust have, **S**hould have, **C**ould have and **W**ould like but won't get

# 1. INTRODUCTION

The present document, deliverable **D.3.1 – System interfaces definition** is the first deliverable of WP3, which is related to the development of the LetItFlow system components. In the previous deliverables, the consortium has focused on the requirements analysis and system design aspects of the project. D3.1 represents the link between these initial aspects and the implementation tasks.

According to the Description of Work of the LetItFlow project, D3.1 reports the activity of *Task 3.1 System Interfaces*. This will represent the main input to deliverable D3.2 and to the integration tasks of WP4.

## 1.1 Scope and objectives of the deliverable

The main outputs of WP2 are the functional specification of the LetItFlow project and a general architecture design, further including the identification of the different modules that will compose the final system. After this work and following the typical software development cycle, the next logical step prior to the implementation tasks is to create more detailed designs. However, when it comes to development activities in collaborative projects it is crucial to define responsibilities and to pay special attention to the joint points among partners, since it is usually more critical to define the interfaces of the different modules than offering detailed designs of them. This is the better way to assure that the final integration of the works performed in parallel will be carried out smoothly.

The deliverable D3.1 reports this effort to establish a common understanding among partners and serves as a reference document for defining the interfaces of the LetItFlow modules. Specifically, the main goals of this document are:

- Define the software and hardware architectures to be deployed in the final scenarios
- Identify the interfaces of the different LetItFlow modules, which at this stage will be considered as black boxes
- Define a common data model
- Identify the development environment and tools to be used during the implementation stage
- Define the test environments for performing atomic tests on modules.

## 1.2 Structure of the deliverable

The deliverable D3.1 is structured in sections as follows:

- Section 2 - Modules definition: presents a refinement on the definition of the LetItFlow modules
- Section 3 - User interface mockups: presents a refinement on the user interfaces
- Section 4 - Infrastructure: includes the description of the software and hardware infrastructure that will be deployed in the final LetItFlow scenarios
- Section 5 - Development environment and repository: evaluates different development environments and chooses for each module the most adequate
- Section 6 - Testing environment: refers the most relevant aspects regarding the process of testing the major modules of the LetItFlow solution
- Section 7 - Interfaces definition: defines the base technologies to develop the services for the interfaces of the LetItFlow modules, describes templates for defining such interfaces and reports all the specific interfaces and services.

- Section 8 - Data models: defines the data model that the modules will use for communicating among them through the interfaces.

# 2. MODULES DEFINITION

After some further analysis on the results of WP2, the Consortium has seen the necessity of refining the definition of the LetItFlow modules. In this section of the document we update (and in some cases extend) such definitions, so that the Consortium establishes a common understanding of the different functionalities.

The LetItFlow modules are:

- **Workflow engine:** This is the core component that manages all the business processes logic. It allows users to define, execute and monitor business processes. It also allows managing and monitoring the status of activities and the transition between different tasks inside a process. It belongs to the server side or backend part of the LetItFlow system.
- **LETAPP:** The smart device application (tablet, smartphone, smartwatch) to support technicians, nurses and operators in their daily work tasks. It represents the main front-end for the workflow engine. It guides the workers throughout the business processes implemented in the workflow engine.
- **LETALARM:** alarm system used for notifying the workers about high priority tasks and about some of the tasks triggered by timers (e.g. in the form of reminders).
- **LETCRITICAL:** alarm/warning system for notifying the workers about critical situations. In this case, the alarm is triggered by humans or monitoring devices (e.g. vital signal monitoring system). Most of the functionalities of this module will be implemented in the smartwatch interface.
- **LETTRAIN I:** training application based on LETAPP (task related content, recording status of the task…). Similar to a read-only LETAPP. This module allows staff to be trained and get familiar with the different business processes.
- **Monitor Manager**: module for monitoring the status of workers' during their daily activities.
- **Communication module (N-Linx based):** Communication layer that provides support to the communication between modules based on the AMQP protocol.

Apart from these modules, the Consortium has identified two additional modules. Aiming at focusing on the important parts of the LetItFlow system, the Consortium has decided to assign them with a low priority and it has been decided to postpone its implementation for now. The modules are the following:

- LETTRAIN II: related to the analysis system (e.g. based on the ObserverXT)
- LETSIM: simulation tool that focuses on the laboratory workflow.

# 3. USER INTERFACE MOCKUPS

In the following section we will describe the first version of User Interfaces for the proposed LetITFlow system that interacts with different modules that build the overall workflow support system. The workflow support system is the core of the LetITFlow project. The modules are the workflow engine (section 3.1), the LetITFlow application deployed on a smartphone (section 3.2) and a smartwatch (section 3.3) and the monitoring manager (section 3.4).

The interfaces design is based on existing guidelines and emphasizes on the special needs of the active older adults, considering the specific objectives of the LetItFlow project. Following a personalization approach, we will design the interfaces to be adaptable (e.g. in terms of speaker volume, size of the text, brightness, content, etc.) to fulfil various user needs.

## 3.1 Workflow Engine

The User Interface of the workflow engine is a PC based interface for the laboratory manager and is based on web technologies.

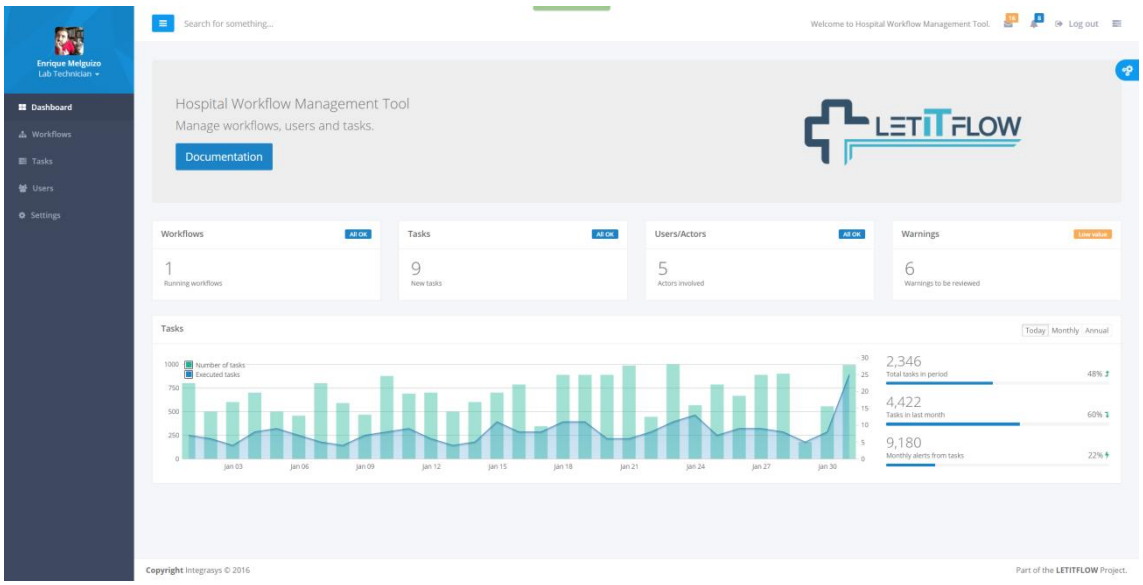


**Figure 1. Workflow engine UI interface (dashboard)**

The Figure 1 represents the main page of the workflow engine. In this page, the manager can visualise some statistics related to different performance indicators for the different business processes and access to other managerial functions. Specifically, by using the left panel, the manager is able to access workflows installed in the engine, modify some of their properties (e.g. user assignment, time schedules, etc.), visualise the tasks and the users assigned to those tasks, or manage the users. All these features are represented in the next figures.

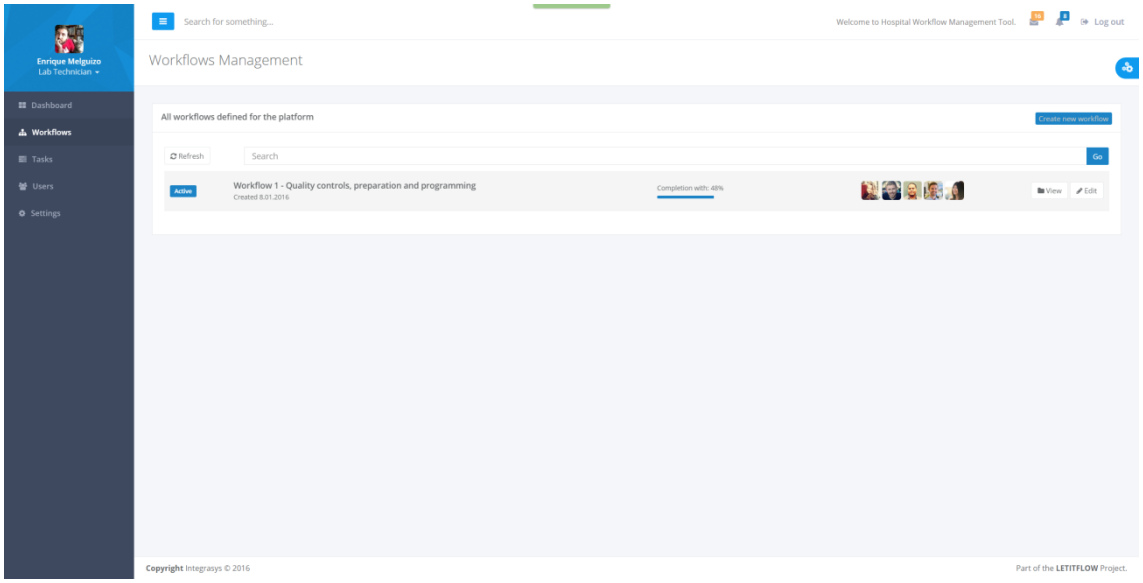


**Figure 2. Workflow engine UI interface (list workflows)**

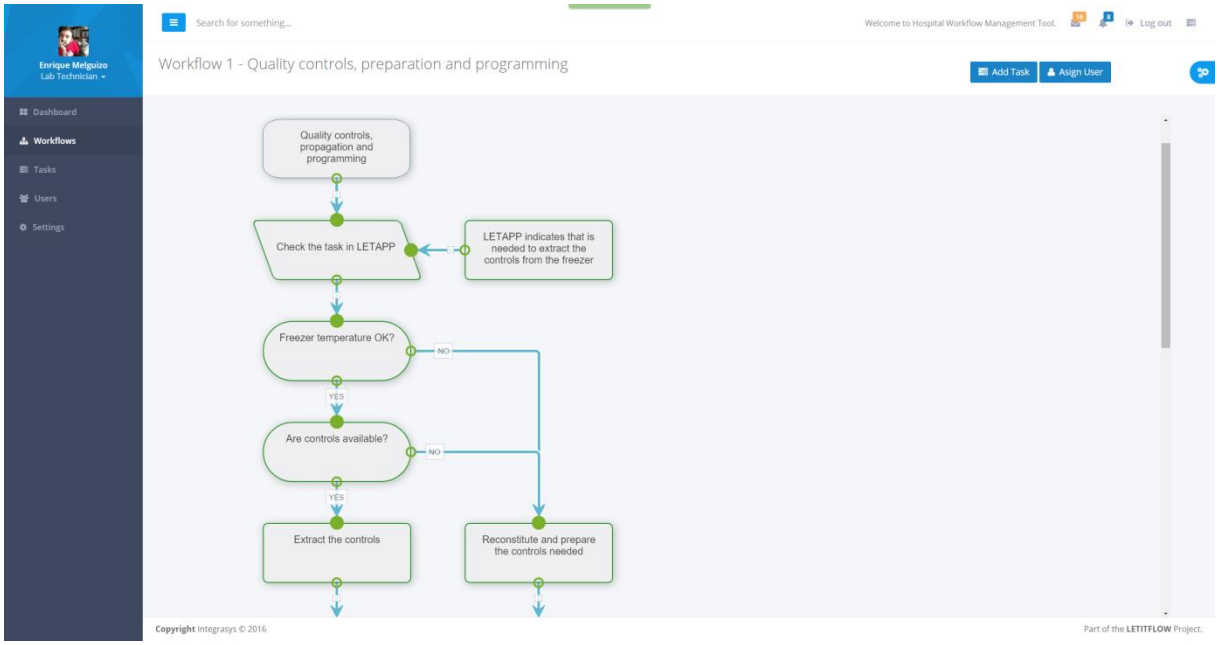


**Figure 3. Workflow engine UI interface (visualise workflows)**

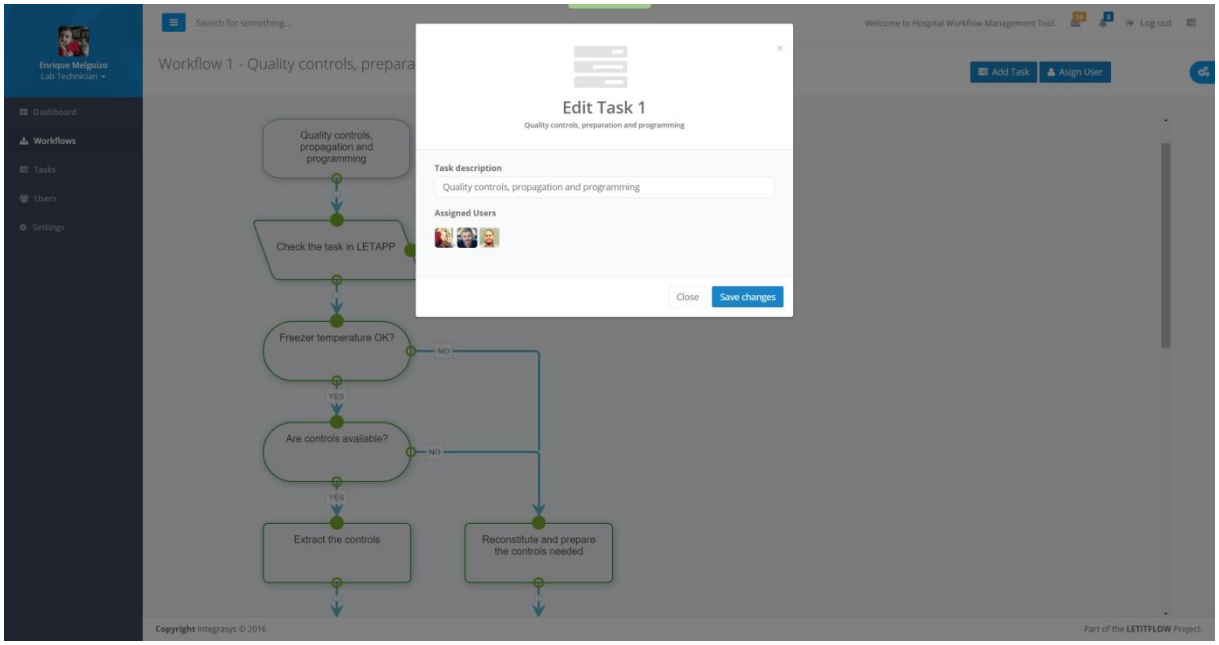


**Figure 4. Workflow engine UI interface (edit workflow properties)**

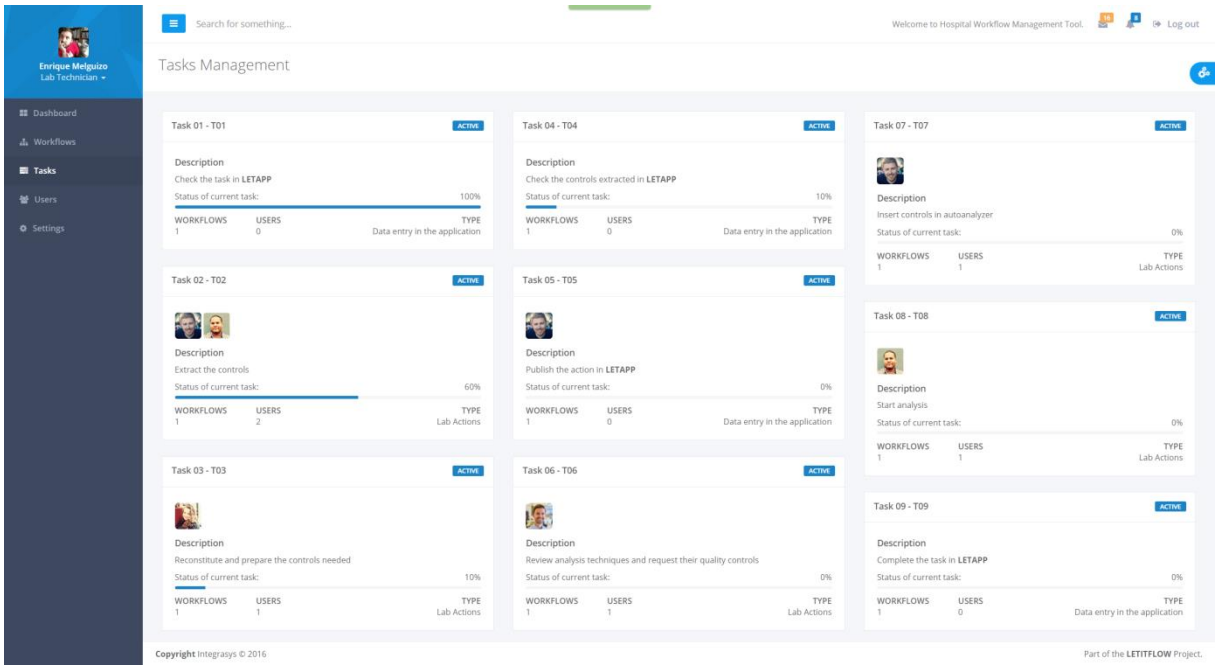


**Figure 5. Workflow engine UI interface (visualise tasks)**

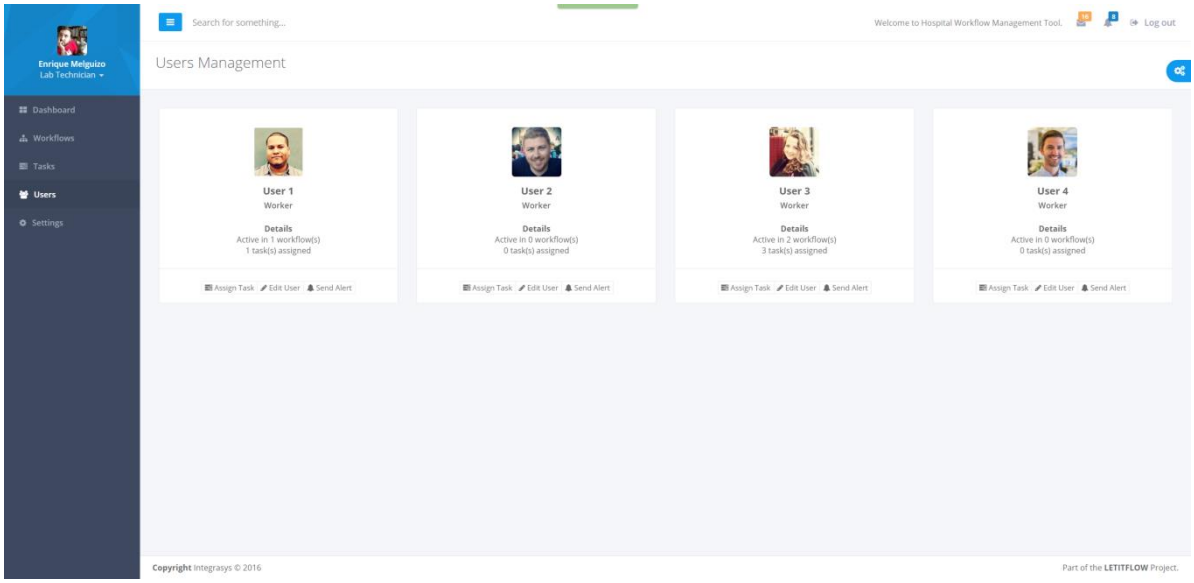


**Figure 6. Workflow engine UI interface (manage users)**

## 3.2 LetItFlow Smartphone Interface

In this section the user interface for the smartphone is described. The smartphone application allows access to the functionalities of LetApp, LetAlarm, LetTrain and LetCritical.

### 3.2.1 LetApp

The LetApp interface comprises three main functionalities that can be accessed by tapping on the icons at the bottom: (1) List of tasks, (2) Alarm functionality, (3) Messaging functionality.

| Status | Colour | Symbol |
|---|---|---|
| **Completed** | Green |  |
| **In Progress** | Yellow |  |
| **Open** | Grey |  |
| **On Hold** | Orange |  |

**Table 1 - Status of Task, Color and Symbol**

Figure 7 shows the interface for the task list. It consists of a list of tasks in a logical order. Each task has an assigned subtask that one may access by tapping on that task. Table 1 gives an overview of task status through a colour-coded background and a dedicated icon for the four states. If a subtask is displayed, the main task (e.g. "(1) Startup biochemistry…") will be displayed at the top of the screen and always remain visible, so that the user knows which task he is currently working on. For a quick switch between tasks in the subtask screen, the user may tap on the main task icon to get back to the list of available tasks.

The status of a task may be changed either by changing the status of all subtasks (when all subtasks are completed, the status of the main task changes automatically), or by changing the status of the task itself. This is possible by tapping on the task status icon.

By tapping on a task, additional training and information material is displayed (see Section 3.2.2 LetTrain).


**Figure 7. LetApp: (a) List of tasks, (b) Subtasks**

Figure 8 shows the messaging functionality of LetApp. The goal of the messaging functionality is to allow for secure internal communication that helps the medical staff in internal coordination for better collaboration. Using the messaging interface, they can send messages to colleagues. Messages may represent text, pictures or audio recordings. It allows individual and group communication.

Frequently used contacts are displayed in the list, for adding a new chat the user has to tap on the new message button.

Messages are shown as bubbles in the order of receiving and sending, a pattern used by most of today's messaging clients. For consistency, we rely on the same presentation mode.


**Figure 8. Messaging Screens in LetApp**

### 3.2.2 LetTrain

Figure 9 shows the smartphone interface for LetTrain. The goal of LetTrain is to provide additional training material (e.g. detailed task descriptions, additional documents for further reading, learning materials, and tutorial videos). User may access additional information by clicking on a task that has information assigned to it. If multiple pages are available (e.g. text description and video), users may change between the pages by either clicking on the “>” and “<” arrows or by swiping to the left / right.

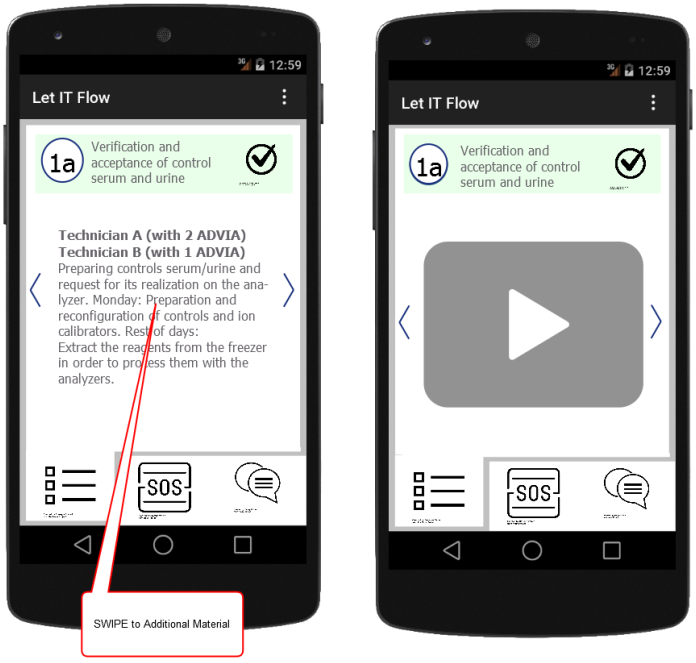


**Figure 9. Additional Training Material in LetTrain**

### 3.2.3 LetAlarm and LetCritical

Figure 10 shows the smartphone interface for LetAlarm. It provides users with the possibility to send an alarm to colleagues.

The main idea proposed in the interface design is to provide the needed functionality quick by hand, so that users may ask for help quickly in an emergency situation. The user may send an alarm message along with additional descriptions in the form of text, voice recordings, or photos to one or more colleagues. A list of colleagues and groups is displayed on the screen: by swiping left, users are shown the colleagues that have already been notified; by tapping on the plus icon, users can add additional colleagues that should receive the alarm.


**Figure 10. Send Alarm Interface in LetAlarm**

Figure 11a shows the smartphone notification for LetAlarm. If an alarm is sent this dialogue is displayed to the user, to inform about the broadcast and allow for cancelation of the call. Figure 11b: An alarm notification is shown if a user receives an alarm broadcast. The user can agree or decline to help. Figure 11c: is displayed to the user after a colleague agreed to help.

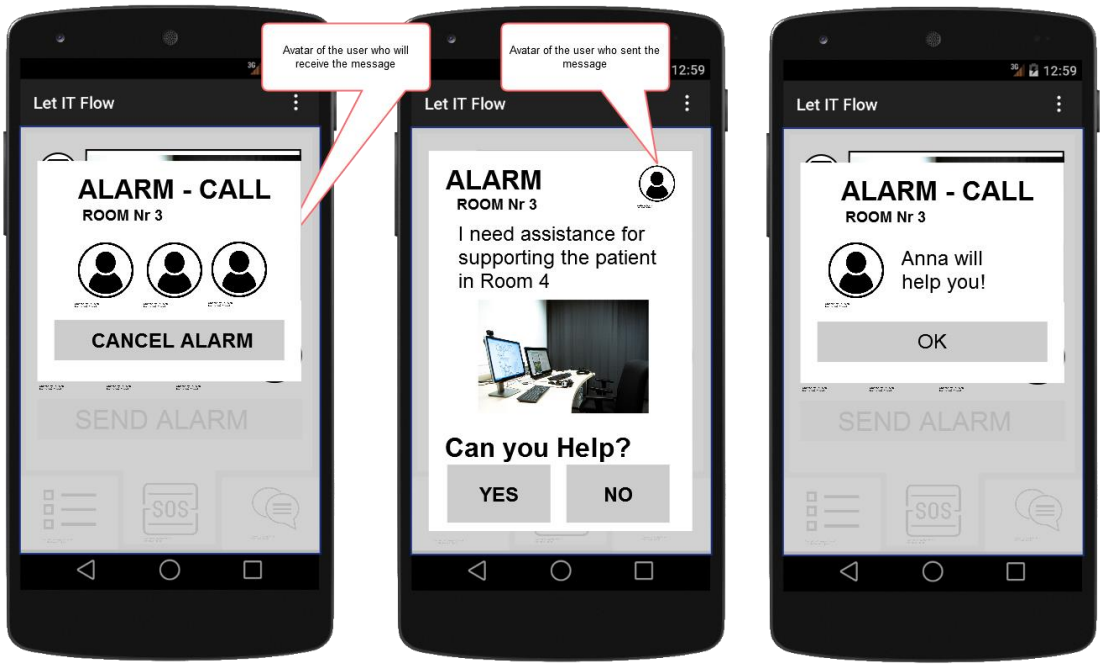


**Figure 11. LetAlarm: (a) Alarm Call and (b) Alarm-Notifications (c) Confirmation**

## 3.3 LetItFlow Smartwatch Interface

As mentioned before, the LetItFlow application is deployed on devices like smartphones, smartwatches. In this section the user interface for the smartwacth is described. The smartphone application allows access to the functionalities of LetApp, LetAlarm, and LetCritical.

### 3.3.1 LetApp

Figure 12 shows the smartwatch interface for LetApp. It provides an overview of open tasks and the subtasks related to this task. The LetItFlow application on the smartwatch follows the design-guidelines for Android wear[1]. Android wear applications basically consist of a set of so-called “cards” that provide only one piece of information at a time, following the principle: “Do one thing, really fast”. Also, information must be visible within a glance, following the principle: “Design for the corner of the eye”.

Figure 12 describes the design principle for one card. It displays the time, the upcoming task and the status of the task (in this example, “on hold”), by setting the background colour and the icon according to the status, as defined in Table 1.


**Figure 12. Smartwatch Design Principle realized in LetApp**

The basic interaction principle is in Figure 13. By swiping up and down the user may switch between tasks. By swiping left and right the user may access subtasks. Figure 14 shows how tasks and subtasks are connected and also provide visualizations for all the four states (cf. Table 1).

[1] https://developer.android.com/design/wear/index.html

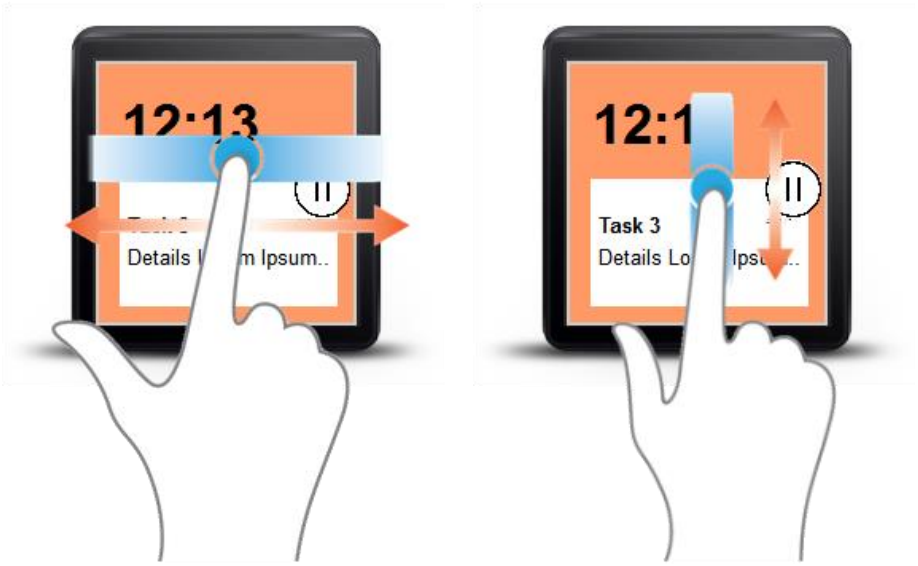


**Figure 13. Smartwatch – Gesture Control**


**Figure 14. Smartwatch – Task Overview and Control**

Figure 15 shows the smartwatch interface for the messaging interface of LetApp. It displays a notification for an incoming message to the user. To reply to this message, the user has to use the the messaging functionality of LetApp. The user may delete the notification by swiping to the left or right.


**Figure 15. Message Notification in LetApp on the smartwatch**

### 3.3.1 LetAlarm and LetCritical

Figure 16 shows the smartwatch interface for LetAlarm and LetCritical. It shows an alert message to the user that was received. The user may delete the notification by swiping to the left or right.


**Figure 16. Alarm Notification in LetAlarm and LetCritical**

## 3.4 Monitoring manager

The monitor manager is the backend component for the LetCritical component and monitors and warns the nurses/elderly whenever a potential critical situation arises. The monitoring manager does not have a real user interface.

# 4. INFRASTRUCTURE

This section includes the description of the software and hardware infrastructure that will be implemented in the LetItFlow project.

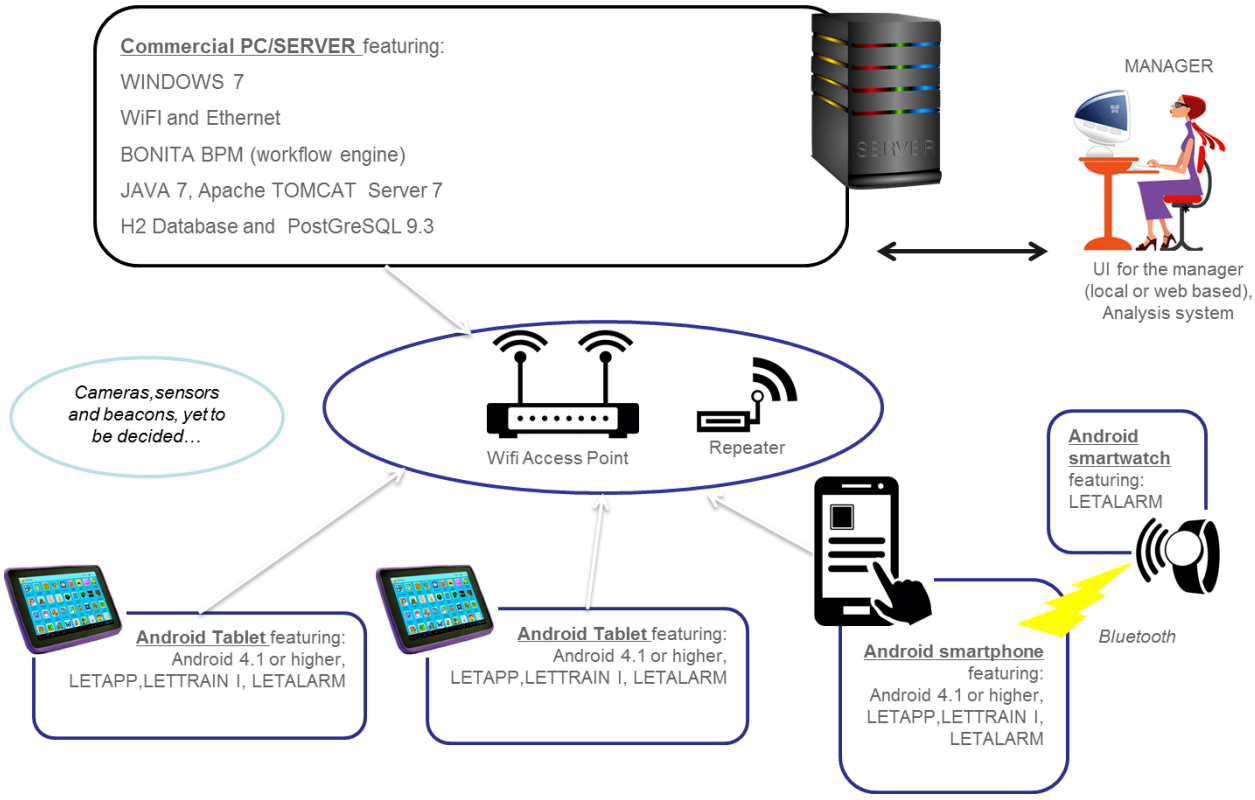


**Figure 17. HW and SW infrastructure for HUVM and UHB**

The preliminary software and hardware architecture devised for the HUVM and UHB scenarios is depicted in Figure 17. As it may be observed, we plan to deploy a new server at the facilities, whose main role is to run the workflow engine module. This will be based on the Bonita BPM 7[2] platform and Java 7 technologies. Such platform will make use of the Apache Tomcat 7 web server and the PostgreSQL database technologies for respectively providing web-based user interfaces and persistence to the business process logic.

The connection between the server and the different mobile devices will be established via a Wi-Fi access point. Some repeater might be required for extending the coverage of the Wi-Fi signal throughout the laboratory of HUVM.

For each of the two medical institutions – HUVM and UHB, the infrastructure is represented by mobile devices including two Android tablets, which will be deployed in fixed positions, one smartphone and one smartwatch associated to it. The tablets and smartphone will access the LETAPP, LETALARM and LETTRAIN I applications. The Android smartwatch will offer some of the LETALARM functionalities.

[2] http://www.bonitasoft.com/for-you-to-read/bpm-library/bonita-bpm-7

# 5. DEVELOPMENT ENVIRONMENT AND REPOSITORY

The deliverable D2.3, Section 7 (i.e. Technology approach), includes some preliminary analysis on the development frameworks, tools and programming languages that the LetItFlow consortium has considered as the most suitable for building the whole LetItFlow system. Based on this evaluation, we identify here the development environment to be used for each module.

Selecting the specific technologies involves balancing the adequacy to the project requirements and to the technical expertise of the LetItFlow partners; normally, when two or more technologies seem adequate at the requirements level, it is chosen the one which is most familiar to the technical staff of the project.

On the other side, the different LetItFlow modules impose different requirements, and therefore different technologies might be used for the achievement of the same goals. It is worth mentioning that this fact is not considered as critical as long as the definition of interfaces allows cross-module communication and a seamless integration. Therefore, the next subsections present the development environment used for creating each of the LetItFlow modules.

## 5.1 Workflow engine

The workflow engine in LetItFlow will be based on the Bonita BPM software version 7.1.

Bonita BPM is a powerful BPM-based application platform for building highly engaging, personalized, process-based business applications. It is based on the BPMN 2.0 standard and it is mainly composed of two parts: the development environment (Bonita BPM Studio) and the runtime environment (Bonita BPM Platform).

Bonita BPM Studio is a graphical environment for creating processes and application pages. Therefore, in LetItFlow we will use this framework for designing the different processes that define the daily activity of hospitals. It will establish a meeting point for the technical and non-technical partners where both can graphically and easily define or review the workflows that will come up from the use cases defined in WP2.

The Bonita BPM Platform is suitable for testing a process that is in development. Once a process is correctly defined in the Bonita BPM Studio, it can be built and deployed on the Bonita BPM production platform. By default, the Bonita BPM platform uses the following technological components for running the processes: Tomcat, the Bonita BPM Portal (part of Bonita BPM visible to process users and managers who use it to manage/execute tasks and processes), the Bonita BPM Engine (workflow engine) and an h2 database. This is the configuration normally used for testing. However, different technologies can be used in production. In LetItFlow we will use the following technologies for running the workflow engine:

- Tomcat server 7
- The Bonita BPM engine
- PostgreSQL 9.3

Regarding the development frameworks we have two options:

- **Eclipse**[3] for building the workflow engine user interface. This uses JavaScript and web based technologies.

[3] https://ro.wikipedia.org/wiki/Eclipse_(software)

- **Bonita BPM Studio**[4]. It has an Eclipse-based graphical interface. It will be used for modelling the business processes and also for implementing them, since the tool itself is a complete framework based on Java technologies. Once the processes are designed, the Bonita BPM Studio offers graphical interfaces for inserting the logic to the processes.

Summarizing, Bonita BPM studio will ensure creating and building the processes and the Bonita BPM engine, Tomcat and PostGreSQL running the processes. The workflow engine will use REST APIs for allowing the access to all the data. You may read Sections 7 and 8 for further details about this.

## 5.2 LETAPP

The LetApp component has two parts, one that runs on a smartphone and a central part that runs on a PC/Server. The PC/Server component will manage the daily medical staff activity by interacting with their daily routine. The mobile component will aid the medical staff in their daily routine by displaying useful information (tasks, activities etc). The displayed information will be available to interact with presenting the user with detailed information or updates on their current activity.

This will be possible by developing specific business processes in Bonita BPM Engine (Workflow engine) that will act as functional requirements, thus covering the medical staff specific requirements.

In order to facilitate communication and integration with different LetItFlow applications such as LetCritical, a REST API is developed to allow the access to the data (for details, see Section 7).

## 5.3 LETCRITICAL

The LetCritical component has a part that runs on a smartphone and a part that runs on a central PC/Server. The smartphone part is responsible for communicating with the sensors, sending the sensor data to the central machine, and communication with the user. The communication from the smartphone to the central machine use the Noldus N-Linx library, a communication library using the AMQP protocol, build on top of RabbitMQ client libraries (for details, see 7.1.2).

The development platform for the smartphone app will be Android Studio, whereas the server side part will be developed in C++ using Visual Studio 2013.

## 5.4 LETALARM

The same development technologies used for the LETAPP (Section 5.2) are applicable to the LETALARM module.

## 5.5 LETTRAIN I

The same development technologies used for the LETAPP (Section 5.2) are applicable to the LETTRAIN I module. The workflows for LETTRAIN I are similar to those implemented in LETAPP. The most relevant functionalities of LETTRAIN I facilitate the training of the staff and are related to specific content that can be added to the tasks and also to the procedures to allow the users access

[4] https://en.wikipedia.org/wiki/Bonita_BPM

such training content. This content will be stored in the database and it will consist of self-explicative data sources distributed in different formats (pdf, word, mp4 etc.).

## 5.6 Monitoring manager

See Section 5.3 for details. The development techniques applicable for the server side part of the LetCritical component apply to the Monitoring Manager as well.

# 6. TESTING ENVIRONMENT

This section refers the most relevant aspects regarding the process of testing the major modules of LetItFlow solution: testing procedure (specific activities/tasks), test objectives, specific testing approaches, testing methods, test design specifications, test case specifications, testing deliverables and test results.

The goal of this section is to define a first draft of the testing environment which will be used for validating each module independently. The test environment usually entails the hardware, network and supporting software/tools infrastructure required to execute the tests.

Testing strategy presumes the following aspects:

- Agreement on goals and objectives among stakeholders
- Expectation management
- Test identification
- Timeframe.

The following essentials being addressed: “What to test”, “How to plan and execute the testing?” and “Goals of testing”.

***Testing approach***

As presented in the deliverable D2.2, during the implementation of the LetItFlow system we will apply an innovative approach – the use of *a mix of methods and principles from Agile methodology and UCD design, specific for the software development*. In the stage of testing we are going to use Agile development processes (testing in an Agile perspective). Agile development process implies continuous improvement and the testing is based on rapid feed-back, following the next path: Task→Task Developing→Task Done→Task Testing→Task Ok→Task Publish.

We consider that the Agile development process is very suitable for any future requirements (either new requirements have to be added or the existing ones have to be changed).

The appropriate Agile testing techniques are specific to the following Agile testing areas (quadrants Qi):

- Q1: Technology Facing tests that support the team
- Q2: Business Facing tests that support the team
- Q3: Business Facing tests that critique the product
- Q4: Technology Facing tests that critique the product.

The specific testing techniques (Unit testing) of Q1 aim to support internal quality of the development process and cover:

- Unit tests
- Component tests.

The specific testing techniques (Functional testing) of Q2 aim to support the collection of better requirements and conditions for software completeness and cover:

- Functional tests
- Story tests
- Simulations.

The specific testing techniques (Exploratory testing) of Q3 ensure that working product meets expectations and cover:

- Scenario tests
- Usability tests
- User Acceptance Tests (UAT).

The specific testing techniques (Performance & Load Testing) of Q4 ensure quality of non-functional requirements and cover:

- Performance tests
- Load tests
- Security tests.

***Test objectives***

Testing Environment is focused on execution of Unit Tests (UT). Accordingly, the objectives of this stage are:

- Unit tests / Component tests will find out whether the units / components behave as expected (software code level).
- Regression tests. The regression test will ensure that the system remains functionally the same after changes in the source code have been made during the development process. The regression test must be 100% automated since it has to be run during every build in order to ensure that the functionality of the application remains the same.

***Testing the features***

This stage of testing procedure means setting up the features which have to be implemented/tested and their level of priority (High, Medium and Low). “High” means that all features with High priority should be implemented first. The features with Low priority can not be the subject of the development and testing processes, they are not mandatory for the system to be operational.

*High* – the feature has to be implemented and tested as soon as possible

*Medium* – the feature could be implemented and tested after the features with high priority

*Low* – the feature is not mandatory for the system to be operational; it should be implemented and tested when the development team hasn’t other high and medium priorities to solve.

For each of the major modules of LetItFlow will be entailed the “Features To Be tested” and the “Features Not To Be Tested”, based on “Prioritization (MoSCoW)” criteria, as presented hereinafter.

***Regression test rules***

The regression test will fully cover the component tests. The purpose of the test is to ensure that every time when a release is made, this release will be tested against the regression test. The main goal of the regression test is to identify changes in the behavior of the application from a previously established date. Regression testing will be performed whenever the software or its environment is changed.

The regression test gives the developers the ability to implement easily and safely major changes within the code. Applying the TDD (Test Driven Development) approach, the development process begins by creating the tests before the code. Thus, before the development of the particular features is started, we have to design and implement the respective test cases for them. If any future

requirements are demanded, new corresponding tests will be drafted and added into the existing regression test and only then will start the requirements development.

***Coverage requirements***

Test coverage belongs to the “White box testing” techniques which addresses the developer/tester perspective of application testing.

White box testing techniques explore the following aspects:

- Examine paths in implementation
- Each statement, decision branch or path has at least one test case
- Coverage definition varies
- Coverage should be calculated and tracked automatically.

Test coverage is defined as “how many potential paths were performed” and is focused on:

- Code coverage
- Decision coverage
- Condition coverage
- Path coverage.

In this stage of testing the major functionalities of LetItFlow solution, the requirements for the Unit/Component test are 100% based on decision coverage.

***Test deliverables***

The following test deliverables (Scripted testing) will be required during the implementation of LetItFlow project:

- Test plan – overall plan for testing
- Test design specifications – define what needs (features / functionalities) to be tested
- Test case specifications (TC – Taste Case) – define tests to run
- Test procedure – define how test should be run
- Test logs – record details of test in timely order
- Test summary reports – present results and evaluation of tests.

***Environmental Needs*** of testing will be depicted hereinafter for each major module/component of LetItFlow solution (workflow engine, LETAPP, LETALARM, LETCRITICAL, LETTRAIN, Monitoring manager).

***Test Design Specification***

Test Design Specification addresses the *features to be tested* (represented by the “must have” requirements of each module). This stage of testing (business facing tests) is focused on business requirement fulfillment.

*Test Design Specification* will be depicted hereinafter for each major module/component of LetItFlow solution (workflow engine, LETAPP, LETALARM, LETCRITICAL, LETTRAIN, Monitoring manager).

***Test Case Specification (TC – Test Case)***

Test Case Specification addresses the features tested with Unit/Component test and requires test scripts or a procedure.

Test case is characterized by:

- Objective
- Test actions
- Expected results
- Execution preconditions.

A possible structure of TC could be:

**Identifier : TCi**

**Description:**

**Preconditions:**

**Input:**

**Steps:**

1.

2.

…

k.

**Expected result:**

*Test Case Specification* will be depicted hereinafter for each major module/component of LetItFlow solution (workflow engine, LETAPP, LETALARM, LETCRITICAL, LETTRAIN, Monitoring manager).

## 6.1 Workflow engine

***Testing approach***

Testing approach will be user focused, applying the UCD (User Centred Design) and Participative Design principles. These principles will be much more critical during the business processes definition stage, where the design of the specific processes will be done using the BPMN standard and the Bonita BPM Studio framework. Thus the approach is to define and refine iteratively the business processes identified for each of the LetItFlow Scenarios (at HUVM and UHB). Specifically, the technical partners will define the processes according to the user stories and use cases defined in WP2 and the final users (i.e. hospitals) will review those processes to validate if they describe

realistically their daily workflows. Several feedback cycles might be required to consolidate the definition of each workflow.

When it comes to development properly said, we are going to use the following techniques for testing:

- Unit tests
- Component tests.

When we talk about Unit tests for this LetItFlow module we refer to testing a task. This will be the smallest testable part. Component tests refer to a whole process.

***Features to be tested (Test design specification)***

For each task contained in the processes developed with the Bonita BPM studio, we will create a basic web interface that represents the forms that will need to be filled by the users when performing each step of a case. These web interfaces will eventually have an equivalent interface in the mobile devices (majorly for the LETAPP module), so they will serve to test that the LETAPP features will integrate smoothly with the workflow engine.

In those web interfaces we will only pay attention to the functionality (i.e. not usability) and include debugging messages for checking the REST API of the workflow engine.

At this point, the processes and tasks are not yet designed, so we cannot define now the features that will be tested. We can only state that a test will be created for each task that composes a business process.

***Tools (Environmental needs)***

The Bonita BPM Studio contains several features that you can use to verify your processes and perform tests. For instance, you can automatically validate a process diagram by just pressing a V*alidation* button. Additionally, there are test connectors for attaching external test cases for any of the components built within the process if necessary.

Also, when you run a process from Bonita BPM Studio it runs on the local Bonita BPM Engine and launches a local Bonita BPM Portal. The process is not deployed, but behaves as it would after deployment. Running a process from Bonita BPM Studio is intended for testing during process development.

Therefore, for testing the workflow engine and the processes developed over it, we will majorly use the built-in testing features of the Bonita BPM Tool suite. As stated before, with that aim, we will develop some basic web forms to validate each task and step of the processes.

***Test cases (Features tested with Unit Test)***

Since the test cases will be defined according to the tasks that compose the business processes, and these need yet to be designed during the implementation phase, we will define the test cases in the next deliverable of WP3.

## 6.2 LETAPP

***Testing approach***

The development/testing process is mainly centred upon Agile Development Methodologies, most specifically TDD (Test Driven Development), taking into account the specific methodological requirements of the project.

The end-users will be involved in the testing process and their feed-back will be included during the development of new releases.

Testing process will be user focused, applying the UCD (User Centred Design) and Participative Design principles.

We are going to use specific techniques of test environment:

- Unit tests
- Component tests.

***Features to be tested (Test design specification)***

As mentioned before, for each of the major modules of LetItFlow will be entailed the “Features To Be tested” and the “Features Not To Be Tested”, based on “Prioritization (MoSCoW)” criteria specified in the deliverable D2.3.

*Features to be tested* (features that are the object of testing design specification) are represented by the “must have” requirements of LETAPP module as follows:

1. The user should be able to login into LETAPP through a mechanism of profiles registered with credentials. The user will access LETAPP through his/her credentials (UserName and Password).
    a. If the user introduces correctly the credentials, he is gaining access to the LETAPP Main Menu.

    b. If the credentials of the user were incorrect, then an error/fail message will be prompted informing the user that he/she has provided inappropriate credentials.

2. The user/system administrator should be able to register new users in the database.

3. The user should be able to create new tasks in the database.

4. The user should be able to able to start, cancel, resume or complete a task.

5. The user/system administrator/manager should be able to assign/add tasks for a certain user.

6. The user should be able to send a message to another user or group of users registered in the application.

7. The user should be able to receive messages that are addressed to him.

8. The user should be able to access the task list, the subtasks assigned to him and also the status of each of these tasks. LETAPP must display in case of more complex tasks, subtasks/steps that need to be fulfilled in order to complete the specific tasks. The user should be able to modify the status of his/her tasks in order to keep track of his/her activities.

***Features not to be tested***

*Features not to be tested* are represented by the “should have” and “would have” requirements of LETAPP module, as specified in the deliverable D2.3. These features are not the subject of the development and testing processes; their implementation level of priority is “Low”.

Features not to be tested of LETAPP are the following:

- Modify an existent user
- Accessing the Main Menu functions: User Profile details, Tasks List assigned to a user etc.
- Access to task related content
- Add task related content
- Notify the user about upcoming events.

***Tools (Environmental needs)***

We are going to use the following testing tools:

- Testing web interface: Apache JMetter
- Testing web services: JUnit
- Implementing test automation for the designated Test Cases: Maven Framework.

***Test cases (Features tested with Unit Test)***

Test cases will be performed for the “must have” features of LETAPP module as follows:

1. **Identifier: TC 01 (LETAPP)**

**Description**: Users Authentication based on profiles and credentials.

**Preconditions**: User and account are created and associated.

**Input**:

**Steps**:

Success Scenario

1. User enters user name and password;
2. System finds associated account;
3. Credentials are verified
4. User gets access to the system.

**Expected Result:** Accept access to the system based on user credentials

Fail Scenario

1. User enters random user name and password;
2. System checks associated account;
3. Error message is prompted;
4. User does not get access to the system.

**Expected Result**: Deny access to the system based on user credentials.

## 2. Identifier: TC 02 (LETAPP)

**Description**: Register/Create new users.

**Preconditions**:

**Input**:

**Steps**:

Success Scenario

1. User credentials are inserted
2. User credentials are verified in order to make sure that they are not already registered into the system
3. New user is created
4. New user is displayed
5. New user account is displayed.

**Expected Result:** User is created

Fail Scenario

1. User credentials are inserted
2. User credentials are verified not to be existent
3. User is not created
4. Error message is prompted.

**Expected Result**: User is not created (error message is displayed).

## 3. Identifier: TC 03 (LETAPP)

**Description**: Create new task.

**Preconditions**:

**Input**:

**Steps**:

Success Scenario

1. Task details are inserted
2. Task details are verified
3. Task is created
4. New task is displayed.

**Expected Result:** Task is created.

Fail Scenario

1. Task details are inserted
2. Task details are verified
3. Task is not created.

**Expected Result**: Task is not created (error message is displayed).

### 4. Identifier: TC 04 (LETAPP)

**Description**: Modify the status of a task.

**Preconditions**: Task must be created.

**Input**:

**Steps**:

Success Scenario

1. Task status is inserted
2. Task status is modified
3. Task details are displayed.

**Expected Result:** Task status is updated.

Fail Scenario

1. Task status is inserted
2. Task status is not modified
3. Error message is prompted

**Expected Result**: Task status is not updated.

### 5. Identifier: TC 05 (LETAPP)

**Description**: Assign task to a user.

**Preconditions**:

1. Task exists
2. User exists.

**Input**:

**Steps**:

Success Scenario

1. Insert task name
2. Verify if task name exists
3. Insert user name
4. Verify if user name exists
5. Task is assigned to the concerned/related user
6. Task details and the assigned user are displayed.

**Expected Result:** Task is assigned.

Fail Scenario

1. Insert task name
2. Verify if task name exists
3. Insert user name
4. Verify if user name exists
5. Task is not assigned to the concerned/related user
6. Error message is prompted.

**Expected Result**: Task is not assigned.

6. **Identifier: TC 06 (LETAPP)**

**Description**: Send a message.

**Preconditions**:

1. User must be logged in the system.

**Input**:

**Steps**:

Success Scenario

1. Message details and destination are inserted/registered
2. Message is sent
3. Message received confirmation is displayed.

**Expected Result:** Message is sent.

Fail Scenario

1. Message details and destination are inserted/registered
2. Message is not sent
3. Error message is displayed.

**Expected Result**: Message is not sent.

7. **Identifier: TC 07 (LETAPP)**

**Description**: Receive a message.

**Preconditions**:

1. User must be logged in the system.

**Input**:

**Steps**:

Success Scenario

1. Message is received
2. Message content is displayed.

**Expected Result:** Message is received.

Fail Scenario 1

1. Message is not received.

Fail Scenario 2

1. Message is received
2. Message content is not displayed.

**Expected Result**: Message is not received.

8. **Identifier: TC 08 (LETAPP)**

**Description**: Access to task and subtask list.

**Preconditions**:

1. User must be logged in the system
2. User must have assigned tasks.

**Input**:

**Steps**:

Success Scenario

1. Task is selected
2. Task details and sub tasks are displayed
3. Sub task is selected
4. Sub task details are displayed.

**Expected Result:** Task and subtasks are displayed.

Fail Scenario 1

1. Task is selected
2. Task details and sub tasks are not displayed.

Fail Scenario 2

1. Task is selected
2. Task details and sub tasks are displayed
3. Sub task details are not displayed.

**Expected Result**: Tasks and sub tasks are not displayed.

## 6.3 LETCRITICAL

For the LetCritical component there is one scenario to test: Critical Event Handling.

The normal scenario is as follows:

1 A Critical Event is sent to the smartwatch

2a The user sees the event and acknowledges it (there is a critical event).

3a The LetAlarm component will send an alarm message

2b The user sees the event and overrules it (there is no critical event)

3b No further action is taken

2c The user does not see the critical event (because is unable to react)

3c After a predefined amount of time (say 30s) the LetAlarm component will send an alarm message

Expected result is that an alarm event is raised (or not). Fail scenarios in case (a) and (c) is that no alarm message is sent. Fail scenario in case (b) is that an alarm message is sent.

## 6.4 LETALARM

***Testing approach***

The development/testing process is mainly centred upon Agile Development Methodologies, most specifically TDD (Test Driven Development), taking into account the specific methodological requirements of the project.

The end-users will be involved in the testing process and their feed-back will be included during the development of new releases.

Testing process will be user focused, applying the UCD (User Centred Design) and Participative Design principles.

We are going to use specific techniques of test environment:

- Unit tests
- Component tests.

***Features to be tested (Test design specification)***

As mentioned before, for each of the major modules of LetItFlow will be entailed the “Features To Be tested” and the “Features Not To Be Tested”, based on “Prioritization (MoSCoW)” criteria specified in the deliverable D2.3.

*Features to be tested* are represented by the “must have” requirements of LETALARM module as follows:

1. The user should be able to send an alarm to another user or group of users registered in the application.
2. The user should be able to receive alarms that are addressed to him.
3. The user should be able to attend an alarm notify that he is working on/is participating in the incident.
4. It should be possible (LETALARM should be capable) to identify/provide the location of the user.

***Features not to be tested***

*Features not to be tested* are represented by the “should have” and “would have” requirements of LETALARM module, as specified in the deliverable D2.3. These features are not the subject of the development and testing processes; their implementation level of priority is “Low”.

Features not to be tested of LETALARM are the following:

- Notify the user about upcoming events.
- Notify the user for unusual results of vital signs (Interpreting the user’ vital signs).

***Tools (Environmental needs)***

We are going to use the following testing tools:

- Testing web interface: Apache JMetter
- Testing web services: Junit
- Implementing test automation for the designated Test Cases: Maven Framework.

***Test cases (Features tested with Unit Test)***

1. **Identifier: TC 01 (LETALARM)**

**Description**: Send an alarm.

**Preconditions**:

**Input**:

**Steps**:

Success Scenario

1. Alarm details are inserted
2. Alarm is created
3. Alarm is sent
4. Created alarm is displayed.

**Expected Result:** Alarm is sent.

Fail Scenario 1

1. Alarm details are inserted
2. Alarm is not created
3. Error message is displayed.

Fail Scenario 2

4. Alarm details are inserted
5. Alarm is created
6. Alarm is not sent
7. Error message is displayed.

**Expected Result**: Alarm is not sent.

2. **Identifier: TC 02 (LETALARM)**

**Description**: Receive an alarm.

**Preconditions**:

1. Alarm is created
2. Alarm is sent.

**Input**:

**Steps**:

Success Scenario

1. Alarm is received
2. Alarm details are displayed.

**Expected Result:** Alarm is received.

Fail Scenario

1. Alarm details are inserted
2. Alarm is created
3. Alarm is not sent
4. Error message is displayed.

**Expected Result**: Alarm is not sent.

3. **Identifier: TC 03 (LETALARM)**

**Description**: Attend an alarm.

**Preconditions**:

1. Alarm is created
2. User has received the alarm
3. User is working in the incident.

**Input**:

**Steps**:

Success Scenario

1. Alarm is received
2. User notify that is involved in the incident
3. User details are displayed.

**Expected Result:** Notification of attending the alarm is sent and the user details are displayed.

Fail Scenario 1

5. Alarm is received
6. User is involved in the incident
7. Notification is not sent
8. Error message is displayed.

**Expected Result**: Notification is not sent.

Fail Scenario 2

1. Alarm is received
2. User is involved in the incident
3. Notification is sent
4. User details are not displayed.

**Expected Result**: User details are not displayed.

4. **Identifier: TC 04 (LETALARM)**

**Description**: Identify the location of the user.

**Preconditions**:

1. User must be logged in the system

**Input**:

**Steps**:

Success Scenario

1. User location is identified
2. User location is displayed.

**Expected Result:** User location is identified.

Fail Scenario

1. User location is not identified.

**Expected Result**: User location is not identified.

## 6.5 LETTRAIN I

The same testing approach and infrastructure used for the LETAPP (Section 6.2) are applicable to the LETTRAIN I module.

## 6.6 Monitoring manager

The monitoring manager uses a different test approach. It needs annotated data to be trained and that annotated data can be used to access the accuracy and precision of the monitoring manager. Or in other words: Given specific input we know what its output should be. Thus if there is a good set of training data then it should always be possible to feed the module with the training dataset and validate its outcome.

# 7. INTERFACES DEFINITION

After some preliminary analysis realized in previous deliverables and the consequent discussions between partners, it has been agreed to use RESTful web services and AMQP based communication middleware (specifically N-LINX) as the base technologies to develop the services for the LetItFlow modules interfaces.

After performing the analysis of the requirements for the services and modules to be developed in the project, the consortium concluded that both technologies sufficiently cover such requirements; RESTful web services are specified for those cases where a request-reply (data centred) communication paradigm fits the better, whereas N-LINX are used for services demanding publish-subscribe (message centred) communication paradigm.

In the case of the RESTful services we use JSON format for transmitting the data.

## 7.1 Interfaces definition templates

The LetItFlow consortium has used the templates and rules described in the following subsections to define the interfaces and services required by the modules that will compose the final solution. The main objective of such templates is to standardize the definition of all the interfaces and ease the integration process to be performed under WP4.

### 7.1.1 REST based interfaces

REST interfaces are classified in different *categories* (e.g. user management, task management, business data model, etc.). Each *category* represents a subsection inside each module section.

For each *category* we define a number of *resources* (e.g. for the user management category, we can define as resources user, group, role, etc.). Each *resource* is represented by a subsection inside the *category* section. The representation of the *resource* is provided as a JSON object that is specified at the start of the *resource* subsection.

For each *resource* we define the *services* that will operate on it. In most of the cases, these operations fall into the CRUD categories: Create, Read, Update, Delete, which respectively correspond to the POST, GET, PUT and DELETE methods used in REST. Alternatively, some more specific operation can be defined (e.g. Search). The Table 2 is used for describing the *services*.

| ***Service*** | **NameOfTheService** |
|---|---|
| Description | Description of the service |
| Request URL | API URL |
| Request method | Method |
| Request payload | Payload |
| Response payload | Payload |
| Response codes | Code |

**Table 2 - Template for defining RESTful services**

### 7.1.2 N-LINX-based interfaces

N-Linx is an interprocess and cross platform communication library which has been built on top of the RabbitMQ [Ref. 1] and TCP network protocols. It is a small layer but the main purpose of N-Linx is standardization. First it offers out-of-the-box an easy accessible and standardized API in C, C++, C# and Java and with the potential to support numerous other programming languages. Second, NLinx standardizes the messages and data which are exchanged between applications by means of contracts.

The N-Linx library is composed of a base module and a Thrift module. The Thrift module extends the base module with a full RPClike framework. The main difference between both modules is that the Thrift module adds type safety, versioning of interfaces and a boilerplate server framework.

**N-Linx Base**

The N-Linx Base library enables you to send a string or a byte array from a sender application to a receiver application. There is no type safety and the parsing of the string or byte array must be done manually, for example by using JSON string parsing.

The base library offers two communication methods:

a. Produce – consume

b. Request – reply

A producer can send data to one or more consumers in the network. It is a one-way communication pattern. A typical use case is a publish-subscribe design pattern where there are several applications who are interested and listening to events that are emitted by one producer.

The request-reply pattern is ideal for unicast routing of a request message to one receiver that replies with an answer. A typical example is in the more traditional client-server setup.

N-Linx Thrift combines the strength of a type safe RPC framework of Thrift with the flexibility of a RabbitMQ message broker. This opens great possibilities in how programs can communicate with one another and still have type safety, versioning of interfaces and a single site of data definition in term of the Thrift IDL contracts. N-Linx Thrift implements a RabbitMQ transport layer and uses the binary TBinaryProtocol layer and the TThreadPoolServer server layer.

The N-Linx Thrift library offers one communication method:

c. Call – receive

Because the call – receive method is built on top of the N-Linx base library it also supports the communication patterns fanout and direct.

**Contracts**

The contracts of the N-Linx library form the core of the system. They define exactly what information can be exchanged with an application and how the information is exchanged. It is the dictionary of the inter-process communication. The contracts are described in the Thrift IDL language ([Ref. 2]) which allows for a type safe and version-proof definition. The code for each programming language is automatically generated from the IDL files by the Thrift compiler. This powerful concept guarantees that both sides of the communication channel use the exact same data types and interface functions. Furthermore, changes in the interface definition are supported by the Thrift versioning system.

This SDK comes with a few predefined contracts which you can find in the folder SDK\Contracts. The contracts are divided by product and within each product you can find definitions for entities, interfaces and types.

| What | Example[5] | Description |
|---|---|---|
| Entities | Room.Thrift | a compositional struct that defines a domain object. A Room contains e.g. a Name and a Status |
| Interfaces | SessionService.Thrift | Defines the interface with methods. Also defines the Exchange and optional routing key of the communication channel. |
| Types | VisoTypes.Thrift | Defines enumeration and other types of the Viso system. |

**Table 3 - The three parts that make up an N-Linx contract.**

[5] Examples taken from the Viso contract located in folder SDK\Contracts\Viso.

```
namespace cpp Noldus.NLinx.Contracts.Viso

namespace java Noldus.NLinx.Contracts.Viso

namespace csharp Noldus.NLinx.Contracts.Viso



// Settings for this service

const i32         interfaceVersion  = 0;

const   string                  exchange                                =
"Noldus.NLinx.Viso.SessionService"

const  string         routingKey           = "EAF6A7F6-6787-40B0-918B-
17FBA58919DB"


/**  The  session  service  is  responsible  for  creating  new  live
recordings and for retrieving recorded sessions. */

service SessionService extends CommonService.CommonService

{

    /** Get a list of finished sessions from all rooms. A session is
considered as finished if it has a valid stop time. */

    list<Session.Session> GetSessions();
```

**Figure 18. Example of an N-Linx contract for a request – reply type of exchange.**

### N-Linx Base Data Types

The base library of N-Linx supports two data types: you can send a message as a string or as a byte sequence (see the Publish or Call methods in the API reference manuals). The semantics of the message, so what it represents, is a responsibility of the client applications, the N-Linx base library does not make any assumptions about this. Most commonly you can use JSON writing and parsing to compose a message.

| | Description | Values |
|---|---|---|
| Base types | | |
| string | A UTF-8 encoded variable byte sequence. | a unicode string |
| byte array | a sequence of unencoded bytes | 0x72A339… |

**Table 4 - Data types of the N-Linx base library.**

### N-Linx Thrift Data Types

N-Linx Thrift uses the Thrift interface definition language and thus inherits also its types. In addition to the Thrift base types the N-Linx library defines a few extended types (Table 5). First there are definitions for date and time and for duration to guarantee that the exchange of time information is unambiguous, even for communication across time zones. Second, there are definitions for color and a GUID. The extended types are defined in file .\Contracts\Common\Types\ CommonTypes.Thrift.

Note the absence of unsigned integer types. This is due to the fact that there are no native unsigned integer types in many programming languages.

| | Description | Values |
|---|---|---|
| *Thrift base types* | | |
| bool | a boolean value | true, false |
| byte | An 8-bit signed integer | –128 to 127 |
| i16 | A 16-bit signed integer | –32,768 to 32,767 |
| i32 | A 32-bit signed integer | –2,147,483,648 to 2,147,483,647 |
| i64 | A 64-bit signed integer | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| double | A 64-bit floating point number | 1.7E +/- 308 (15 digits) |
| string | A text string with UTF-8 encoding | a Unicode string |

| | | |
|---|---|---|
| binary | a sequence of unencoded bytes | 0x72A339… |
| *N-Linx extended types* | | |
| IsoDateTime | An ISO 8601[6] date and time as string value | “2014-05-23T10:55:00.000000+02:00” |
| UnixTime | A UTC Unix time[7] as double with the fractional part defining the number of milliseconds | 1422863491.675 which is UTC 2015-02-02T07:51:31.675 |
| Duration | The number of microseconds as a 64-bit signed integer | e.g. 2286349 µs =2.286349 seconds |
| Color | An ARGB struct | e.g. {255, 128, 128, 128} defining a grey color |
| GUID | A 16-byte number | e.g. C9 8B 91 35 6D 19 EA 40 97 79 88 9D 79 B7 53 F0 |

**Table 5 - Data types of the N-Linx Thrift library.**

### JSON

JSON (JavaScript Object Notation) is a lightweight structured data format. JSON is easy to read for both humans and computers and has less overhead than XML. You can use the JSON format when you choose to work with the N-Linx base library Publish and Call methods. For N-Linx Thrift you do not need to write your own parsing because this is handled internally by the library.

For the message parsing you need to install a JSON library, for. part of the r example for C++ the ubuntu JSON boost Spirit parser (https://apps.ubuntu.com/cat/applications/precise/libjson-spirit-dev/) or the Newtonsoft Json.NET for C# (http://www.newtonsoft.com/json). More information about JSON can be found here: http://www.json.org/.

### Recommend usage in LetItFlow

The Thrift Library is at this moment **NOT** available for the JAVA library. There for we recommend to use the base N-Linx modules with JSON messages.

An example JSON message to send location data:

```
{
    “nlx_time”: 1450708850804,
```

[6] http://www.iso.org/iso/iso8601

[7] http://en.wikipedia.org/wiki/Unix_time

“nlx_track_id”: “Anne”,

“x”: 12.604,

“y”: 24.400,

“z”: 2.04

}

See Paragraph 4.2 of the SDK documentation for a simple Producer Consumer code example with the Base Library.

## 7.2 Workflow engine interfaces

**The workflow engine used in LetItFlow is based on the Bonita BMP Platform. The interfaces that it uses are based on the REST API that the platform offers. The *resources* defined in the following subsections are directly related to the entities of the**

Figure 19, which represents the entity-relationship diagram for the components that integrates the LetItFlow system.

### 7.2.1 User management

#### *7.2.1.1 user*

JSON representation of the *resource:*

```
{
    "id":"user ID",
    "title":"string",
    "lastname":"string",
    "firstname":"string",
    "userName":"string",
    "password":"",
   "enabled":"true | false"
}
```

*Services* to operate on the *resource*:

| *Service* | **createUser** |
|---|---|
| Description | Create a new user |
| Request URL | http://../API/identity/user |
| Request method | POST |
| Request payload | A partial representation of a user in JSON (“id” not included) |
| Response payload | The full JSON representation of the user |
| Response codes | - |

| *Service* | **readUser** |
|---|---|
| Description | Read a user details |
| Request URL | http://../API/identity/user/<userId> |
| Request method | GET |
| Request payload | - |
| Response payload | The full JSON representation of the user |
| Response codes | - |

| *Service* | **searchUsers** |
|---|---|
| Description | Search for a group of users |
| Request URL | http://../API/identity/user |
| Request method | GET |
| Request payload | - |
| Response payload | A JSON array of users |
| Response codes | - |

| Service | updateUser |
|---|---|
| Description | Update user details |
| Request URL | http://../API/identity/user/<userId> |
| Request method | PUT |
| Request payload | A JSON representation of the user, with the new information |
| Response payload | - |
| Response codes | - |

| Service | deleteUser |
|---|---|
| Description | Remove a user |
| Request URL | http://../API/identity/user/<userId> |
| Request method | DELETE |
| Request payload | - |
| Response payload | - |
| Response codes | - |

#### *7.2.1.2 group*

JSON representation of the *resource:*

```
{
      "id":"group ID",
      "name":"name",
      "parent_path":"the path of the parent group of this group (empty
      if the group has no parent)",
      "path":"the full path of the group (including its parent path)",
      "description":"description"
}
```

*Services* to operate on the *resource*:

| ***Service*** | **createGroup** |
|---|---|
| Description | Create a new group |
| Request URL | http://../API/identity/group |
| Request method | POST |
| Request payload | A partial representation of a group in JSON (“name” is mandatory) |
| Response payload | The full JSON representation of the group |
| Response codes | 403 if a group with the same name and parent already exists |
| *Service* | readGroup |
| Description | Read a group details |
| Request URL | http://../API/identity/group/<group_id> |
| Request method | GET |
| Request payload | - |
| Response payload | The full JSON representation of the group |
| Response codes | 404 if no group with this group ID is found |

| ***Service*** | **searchGroups** |
|---|---|
| Description | Search for groups |
| Request URL | http://../API/identity/group |
| Request method | GET |
| Request payload | - |
| Response payload | A list of groups in JSON |
| Response codes | - |

| Service | updateGroup |
|---|---|
| Description | Update group details |
| Request URL | http://../API/identity/group/<group_id> |
| Request method | PUT |
| Request payload | A partial representation of a group in JSON (“name” is mandatory) |
| Response payload | - |
| Response codes | 403 if another group with the same name and parent already exists<br>404 if no group with this group ID is found |

| Service | deleteGroup |
|---|---|
| Description | Remove a group |
| Request URL | http://../API/identity/group/<group_id> |
| Request method | DELETE |
| Request payload | - |
| Response payload | - |
| Response codes | 404 if no group with this group ID is found |

#### *7.2.1.3 role*

JSON representation of the *resource:*

```
{
      "id":"role ID",
      "name":" name",
      "description":"description",
}
```

*Services* to operate on the *resource*:

| ***Service*** | **createRole** |
|---|---|
| Description | Create a new role |
| Request URL | http://../API/identity/role |
| Request method | POST |
| Request payload | A partial representation of a role in JSON (“name” is mandatory) |
| Response payload | The full JSON representation of the role |
| Response codes | 403 if a role with the same name already exists |

| ***Service*** | **readRole** |
|---|---|
| Description | Read a role details |
| Request URL | http://../API/identity/role/<role_id> |
| Request method | GET |
| Request payload | - |
| Response payload | The full JSON representation of the role |
| Response codes | 404 if no role with this role ID is found |

| ***Service*** | **searchRoles** |
|---|---|
| Description | Search for roles |
| Request URL | http://../API/identity/role |
| Request method | GET |
| Request payload | - |
| Response payload | A list of roles in JSON |
| Response codes | - |

| Service | updateRole |
|---|---|
| Description | Update role details |
| Request URL | http://../API/identity/role/<role_id> |
| Request method | PUT |
| Request payload | A partial representation of a role in JSON (“name” is mandatory) |
| Response payload | - |
| Response codes | 403 if another role with the same name already exists<br>404 if no role with this role ID is found |

| Service | deleteRole |
|---|---|
| Description | Remove a role |
| Request URL | http://../API/identity/role/<role_id> |
| Request method | DELETE |
| Request payload | - |
| Response payload | - |
| Response codes | 404 if no role with this Role ID is found |

#### *7.2.1.4 membership*

JSON representation of the *resource:*

```
{
      "role_id":"id of the role of this membership",
      "group_id":"id of the group of this membership",
      "user_id":"id of the user in this membership"
}
```

*Services* to operate on the *resource*:

| ***Service*** | **createMembership** |
|---|---|
| Description | Create a new membership |
| Request URL | http://../API/identity/membership |
| Request method | POST |
| Request payload | A partial representation of a membership in JSON ("user_id", "group_id" and "role_id" are mandatory) |
| Response payload | The full JSON representation of the membership created |
| Response codes | 403 if a membership with the same attributes already exists |

| ***Service*** | **searchMemberships** |
|---|---|
| Description | Search memberships of a user |
| Request URL | http://../API/identity/membership?p=0&c=10&f=user_id%3d<the id of the user> |
| Request method | POST |
| Request payload | - |
| Response payload | The full JSON representation of the memberships of the user |
| Response codes | - |

| ***Service*** | **deleteMembership** |
|---|---|
| Description | Delete a membership of a user using the group id and role id. |
| Request URL | http://../API/identity/membership/<the id of the user>/<the id of the group>/<the id of the role> |
| Request method | DELETE |
| Request payload | - |
| Response payload | - |
| Response codes | - |

### 7.2.2 BPM management

#### *7.2.2.1 actor*

JSON representation of the *resource:*

```
{
      "id":"actor id",
      "process_id":"process id",
      "description":"a description of the actor",
      "name":"name of the actor
}
```

*Services* to operate on the *resource*:

| ***Service*** | **readActor** |
|---|---|
| Description | Retrieve information about an actor |
| Request URL | http://../API/bpm/actor/<actor ID> |
| Request method | GET |
| Request payload | - |
| Response payload | The full JSON representation of the actor |
| Response codes | - |

| ***Service*** | **searchActors** |
|---|---|
| Description | Search actors for a given process id |
| Request URL | http://../API/bpm/actor?p=0&c=10&f=process_id%3d <process ID> |
| Request method | GET |
| Request payload | - |
| Response payload | The full JSON representation of the actors found |
| Response codes | - |

| Service | updateActor |
|---|---|
| Description | Update an actor information |
| Request URL | http://../API/bpm/actor?p=0&c=10&f=process_id%3d <process ID> |
| Request method | PUT |
| Request payload | A partial representation of an actor in JSON (only “description” can be updated) |
| Response payload | - |
| Response codes | - |

#### 7.2.2.2 *actorMember*

JSON representation of the *resource:*

```
{
      "id":"actor member id",
      "actor_id":"id of the actor for this mapping",
      "role_id":"id of role, or -1 if the member type is not role",
      "group_id":"id of group, or -1 if the member type is not group",
      "user_id":"id of user, or -1 if the member type is not user"
}
```

*Services* to operate on the *resource*:

| Service | createActorMember |
|---|---|
| Description | Add a new actorMember |
| Request URL | http://../API/bpm/bpm/actorMember |
| Request method | POST |
| Request payload | A partial representation of a membership in JSON ("actor_id" is mandatory) |
| Response payload | The full JSON representation of the actorMember created |
| Response codes | - |

| Service | searchActorMembers |
|---|---|
| Description | Search for actor members |
| Request URL | http://../API/ bpm/actorMember?p=0&c=10&f=member_type%3d<member type> |
| Request method | GET |
| Request payload | - |
| Response payload | The full JSON representation of the actorMember found |
| Response codes | - |

| **Service** | **deleteActorMember** |
|---|---|
| Description | Delete an existing actorMember. |
| Request URL | http://../API/bpm/actorMember/<actorMember ID> |
| Request method | DELETE |
| Request payload | - |
| Response payload | - |
| Response codes | - |

#### *7.2.2.3 process*

JSON representation of the *resource:*

```
{
      "id":"the identifier of the process definition (long)",
      "name":"the process name (string)",
      "description":"the process description (string)",
      "activationState":"the state of the process definition (ENABLED or
      DISABLED)",
      "actorinitiatorid":"the id of the actor that can initiate cases of
      the process"
}
```

*Services* to operate on the *resource*:

| ***Service*** | **readProcess** |
|---|---|
| Description | Read a process definition |
| Request URL | http://../API/bpm/process/<processId> |
| Request method | GET |
| Request payload | - |
| Response payload | The full JSON representation of the process |
| Response codes | - |

| ***Service*** | **readProcessContract** |
|---|---|
| Description | Retrieve the process instantiation contract elements. These are required to start a process |
| Request URL | http://../API/bpm/process/<processId>/contract |
| Request method | GET |
| Request payload | - |
| Response payload | A JSON object representing the process contract elements |
| Response codes | - |

| ***Service*** | **startProcess** |
|---|---|
| Description | Start a process providing correct contract values. This will automatically create a new case. |
| Request URL | http://../API/bpm/process/<processId>/instantiation |
| Request method | POST |
| Request payload | A JSON object representing the contract element values. |
| Response payload | The created case ID in JSON format |
| Response codes | 201 OK or a contract violation explanation in case of a 400 Bad request |

#### 7.2.2.4 *case*

JSON representation of the *resource:*

```
{
      "id":"the identifier of the case",
      "start":"the starting date of the case",
      "state":"an enum that represent the state of the case. It can be
      INITIALIZING, STARTED, SUSPENDED, CANCELLED, ABORTED, COMPLETING,
      COMPLETED, ERROR, ABORTING",
      "rootCaseId":"the identifier of the container of the case",
      "started_by":"the identifier of the user who started the case",
      "processDefinitionId":"the identifier of the process related of
      the case"
}
```

*Services* to operate on the *resource*:

| ***Service*** | **createCase** |
|---|---|
| Description | Create a case |
| Request URL | http://../API/bpm/case |
| Request method | POST |
| Request payload | The process definition id, in JSON. |
| Response payload | The JSON representation of a case resource |
| Response codes | - |

| *Service* | **readCase** |
|---|---|
| Description | Get a case by using its identifier |
| Request URL | http://../API/bpm/case/<case ID> |
| Request method | GET |
| Request payload | - |
| Response payload | A JSON representation of the case |
| Response codes | - |

| *Service* | **searchCases** |
|---|---|
| Description | Search for a case |
| Request URL | http://../API/bpm/case/ |
| Request method | GET |
| Request payload | - |
| Response payload | JSON representations of matching cases |
| Response codes | - |

| *Service* | **deleteCase** |
|---|---|
| Description | Delete a case |
| Request URL | http://../API/bpm/case/ |
| Request method | DELETE |
| Request payload | The case id, in JSON |
| Response payload | - |
| Response codes | - |

#### 7.2.2.5 task

JSON representation of the *resource:*

```
{
      "id":"the task id (long)",
      "name":"the task name (string)",
      "description":"the task description (string)",
      "processId":"the process id (long) that is associated with this
      task",
      "state":"the current state of the task (string, for example,
      ready, completed, failed)",
      "assigned_id":"the user id (long) that this task is assigned to,
      or 0 if it is unassigned",
      "executedBy":"the id (long) of the user who executed the task, or
      0 if the task has not been executed",
      "caseId":"the case id (long) that is associated with this task",
      "actorId":"the id (long) of the actor that can execute this task,
      null otherwise"
}
```

*Services* to operate on the *resource*:

| ***Service*** | **readTask** |
|---|---|
| Description | Read a task |
| Request URL | http://../API/bpm/task/<task ID> |
| Request method | GET |
| Request payload | - |
| Response payload | JSON representation of a task |
| Response codes | - |

| *Service* | **searchTasks** |
|---|---|
| Description | Search for a task |
| Request URL | http://../API/bpm/task/ |
| Request method | GET |
| Request payload | - |
| Response payload | JSON representation of an array of tasks |
| Response codes | - |

| *Service* | **updateTask** |
|---|---|
| Description | Update a task field(s) |
| Request URL | http://../API/bpm/task/<task ID> |
| Request method | PUT |
| Request payload | Task fields to update (forbidden fields are : caseId, processId, name, executedBy, id) |
| Response payload | - |
| Response codes | - |

#### *7.2.2.6 timerEventTrigger*

JSON representation of the *resource:*

```
{
      "id": the ID of the timer returned,
      "eventInstanceId": the ID of the event instance to which this
      trigger is related,
      "executionDate": the long value of the next execution date (number
```

```
      of milliseconds from January 1st, 1970 00:00:00),

      "eventInstanceName": the name of the event instance to which this
      trigger is related

}
```

*Services* to operate on the *resource*:

| ***Service*** | **searchTimerEventTriggers** |
|---|---|
| Description | Search for timer event triggers related to a case |
| Request URL | http://../API/bpm/timerEventTrigger<br>http://../API/bpm/timerEventTrigger?caseID=<case ID > |
| Request method | GET |
| Request payload | - |
| Response payload | JSON representation of a list of timer event triggers |
| Response codes | - |

| ***Service*** | **updateTimerEventTrigger** |
|---|---|
| Description | Specify the next execution date of a timer event trigger. |
| Request URL | http://../API/bpm/timerEventTrigger/<timerEventTriggerID> |
| Request method | PUT |
| Request payload | A JSON representation of a long value with attribute name "executionDate" |
| Response payload | The actual long value corresponding to the next execution date of the timer event trigger, as a long value |
| Response codes | - |

#### 7.2.2.7 caseVariable

JSON representation of the *resource:*

```
{
      "name":"name of the variable in the case",
      "description":"Detailed description of the case variable",
      "value":"the current value of the case variable",
      "case_id":"ID of the case this variable belongs to",
      "type":the Java type of the variable"
}
```

*Services* to operate on the *resource*:

| ***Service*** | **readCaseVariable** |
|---|---|
| Description | Search for case variables from its case ID, or find a single case variable from case ID and variable name |
| Request URL | http://../API/bpm/caseVariable/<caseID>/<variableName> |
| Request method | GET |
| Request payload | - |
| Response payload | JSON representation of a case variable |
| Response codes | 500, if an exception occurs during the find |

| ***Service*** | **searchCaseVariables** |
|---|---|
| Description | Search for a list of case variables |
| Request URL | http://../API/bpm/caseVariable?p=<firstPageNumber>&c=<pageSize> |
| Request method | GET |
| Request payload | - |

| | |
|---|---|
| Response payload | JSON representation of a list of case variables |
| Response codes | 500, if an exception occurs during the find |

| ***Service*** | **updatecaseVariable** |
|---|---|
| Description | Update a case variable value. |
| Request URL | http://../API/bpm/caseVariable/<caseID>/<variableName> |
| Request method | PUT |
| Request payload | A JSON representation of a case variable (“type” and “value” are mandatory attributes) |
| Response payload | - |
| Response codes | 200, if Ok.<br>500, if an exception occurs during the find. |

### 7.2.3 BDM management

#### *7.2.3.1 businessData*

JSON representation of the *resource:*

```
{
      "persistenceId": number,
      "attributeName":attributeType
      ...
}
```

*Services* to operate on the *resource*:

| ***Service*** | **readBusinessData** |
|---|---|
| Description | Get the business data specified by its identifier: “businessDataType” (e.g. com.company.model.Client) and “persistenceId” |
| Request URL | http://../API/bdm/businessData/<businessDataType>/<persistenceId> |
| Request method | GET |

| | |
|---|---|
| Request payload | - |
| Response payload | A business data in JSON format |
| Response codes | 500, when business data identifier is not valid |

| ***Service*** | **readBussinessDataAttribute** |
|---|---|
| Description | Get the business data attribute of business data according to its identifier and attribute name. |
| Request URL | http://../API/bdm/businessData/<businessDataType>/<persistenceId>/<attributeName> |
| Request method | GET |
| Request payload | - |
| Response payload | A business data in JSON format |
| Response codes | 500, if an exception occurs during the find |

#### *7.2.3.2 businessDataQuery*

There is no JSON representation relevant for this *resource*.

*Services* to operate:

| ***Service*** | **executeBusinessDataQuery** |
|---|---|
| Description | Execute a query on a business data resource. It can be a default or custom query, previously defined in the Bonita BPM Studio. |
| Request URL | http://../API/bdm/businessData/businessDataType?q=queryName&p=0&c=10&f=param=value<br><br>Where:<br>businessDataType - the fully-qualified business data type name<br>q=queryName - the query name<br>p=0 - the page number |

| | |
|---|---|
| | c=10 - the maximum number of results in the page<br>f=parameter=value - sets the parameter value according to business data query parameters defined in Bonita BPM Studio |
| Request method | GET |
| Request payload | - |
| Response payload | JSON representation of query result |
| Response codes | - |

## 7.3 LETAPP interfaces

| ***Service*** | **LOGIN** |
|---|---|
| Description | Access the system |
| Request URL | http://API/LogIn/?{userName}/{password} |
| Request method | GET |
| Request payload | - |
| Response payload | Verify if the UserName and Password are correct |
| Response codes | - |

| ***Service*** | **sendChatMessage** |
|---|---|
| Description | Send a message |
| Request URL | http://API/sendChatMessage/ |
| Request method | PUT |
| Request payload | The full JSON representation of the message |
| Response payload | |
| Response codes | - |

| *Service* | **ReceiveChatMessage** |
|---|---|
| Description | Receive a message |
| Request URL | http://API/recievedChatMessage/ |
| Request method | GET |
| Request payload | The full JSON representation of the message |
| Response payload | The full JSON representation of the message |
| Response codes | - |

| *Service* | **View task related content** |
|---|---|
| Description | View task related content |
| Request URL | http://API/taskDetails/?taskId={taskId} |
| Request method | GET |
| Request payload | The full JSON representation of the Task content |
| Response payload | The full JSON representation of the Task content |
| Response codes | - |

| *Service* | **View Tasks** |
|---|---|
| Description | View all user tasks |
| Request URL | http://API/getAllTasks/ |
| Request method | GET |
| Request payload | The full JSON representation of the Task |
| Response payload | The full JSON representation of the Task |
| Response codes | - |

| *Service* | **View SubTasks** |
|---|---|
| Description | View all user tasks |
| Request URL | http://API/getAllSubTasks/?taskId={taskId} |
| Request method | GET |
| Request payload | The full JSON representation of the SubTasks |
| Response payload | The full JSON representation of the SubTasks |
| Response codes | - |

| *Service* | **Add task related content** |
|---|---|
| Description | Add task related content |
| Request URL | http://API/addTaskContent/?taskId={taskId} |
| Request method | PUT |
| Request payload | The full JSON representation of the Task |
| Response payload | The full JSON representation of the Task |
| Response codes | - |

| *Service* | **Get statistical and historical data** |
|---|---|
| Description | Display statistical and historical data |
| Request URL | http://API/getData/ |
| Request method | GET |
| Request payload | The JSON list of historical and statistical data |
| Response payload | The JSON list of historical and statistical data |
| Response codes | - |

| *Service* | **Notify upcoming events** |
|---|---|
| Description | Get notified about upcoming events |
| Request URL | http://API/getUpcomingEvents/ |
| Request method | GET |
| Request payload | The JSON list of upcoming events |
| Response payload | The JSON list of upcoming events |
| Response codes | - |

| *Service* | **General Status View** |
|---|---|
| Description | Get informed about the general task status for users |
| Request URL | http://API/getUserStatus/{userName} |
| Request method | GET |
| Request payload | The JSON list of all the task with their status related to the user |
| Response payload | The JSON list of all the task with their status related to the user |
| Response codes | - |

## 7.4 LETCRITICAL interfaces

Depending on the sensors that will be selected to be used by the LetCritical component the exact format of the messages to be sent might vary. From a General perspective there are two types of messages:

1) Push Data messages. These are the message that will be sent from the smartphone to the server for further processing.

This is a JSON message with at least the following fields:

```
{
      "id": the ID of the user,
```

```
      "sensorName": descriptive name of the sensor,

      "sensorType": from a fixed enumeration (eg. HeartRate,
      BloodPressure, etc.),

      "time": the long value of the data generation (number of
      milliseconds from January 1st, 1970 00:00:00),

      "value": Sensor value (type determined by SensorType)

      ...

}
```

2) Request/ReplyCritical Events. These are messages sent from the central server to the phone when a critical event is detected. The server expects an answer on these messages, if no answer is given within a predefined timeframe an alarm will be raised (see LetAlarm in the following paragraph).

```
{

      "id": the ID of the critical event,

      "type": the type of critical event,

      "description": the description of the specific critical event,
      will be displayed to the user

      "userId": the ID of the user,

      "time": the long value of the detection time of the critical event
      (number of milliseconds from January 1st, 1970 00:00:00),

      ...

}
```

The reply (if any) consist of a following JSON String

```
{

      "id": the ID of the critical event,

      "userId": the ID of the user,
```

```
      "response": true | false (real critical event or false alarm)
      ...
}
```

## 7.5 LETALARM interfaces

| ***Service*** | **Send alarm** |
|---|---|
| Description | Send critical alarm |
| Request URL | http://LETALARM/SendAlarm/ |
| Request method | PUT |
| Request payload | The JSON structure of an critical alarm |
| Response payload | |
| Response codes | - |

| ***Service*** | **Receive alarm** |
|---|---|
| Description | Get critical alarm |
| Request URL | http://LETALARM/RecieveAlarm/ |
| Request method | GET |
| Request payload | The JSON structure of an critical alarm |
| Response payload | The JSON structure of an critical alarm |
| Response codes | - |

| ***Service*** | **Attend alarm** |
|---|---|
| Description | Inform that the user will be attending the alarm |
| Request URL | http://LETALARM/AttendAlarm/?alarmId={alarmId} |

| | |
|---|---|
| Request method | PUT |
| Request payload | - |
| Response payload | |
| Response codes | - |

| ***Service*** | Identify the location of user |
|---|---|
| Description | **Identify the location of user** |
| Request URL | http://LETALARM/emergencyLocation/?userId={userId} |
| Request method | GET |
| Request payload | The JSON structure of the user location |
| Response payload | The JSON structure of the user location |
| Response codes | - |

## 7.6 LETTRAIN I interfaces

The LetTrain I module is designed as a read-only application based on LetApp. Therefore, this module will use a subset of the interfaces presented for the workflow engine. Specifically, those classified as *Read* or *Search* operations in Section 7.2.

## 7.7 Monitoring manager

Since the monitoring manager is the backend for the LetCritical component its interfaces are described in Section 7.4

# 8. DATA MODELS

In this Section, the data model that corresponds to the access to the main components of the LetItFlow system (e.g. tasks, processes, users, etc.) is presented. The conceptual ER diagram in

Figure 19 represents such data model.

Aiming at better understanding the data model, we present some of the key concepts that the entities in the diagram represent:

- Process: A process is a sequence of tasks. The sequence has a start and an end, and can contain decision points, known as gateways. A process definition is constructed as a flow diagram (based on BPMN 2.0 standard).
- Task: A task is an activity in a process. Tasks may be performed by people (human tasks, which normally involve using a form to enter data or to receive information) or automatically (service tasks, invisible to users during normal operation).
- Case: A case is an instance of a process. This means that a process might be executed in parallel by different users, and the workflow engine maintains independent information for each instance.
- Organization: The set of all the users who play a part in all the processes that model the business uses. Typically, the organization corresponds to the hierarchy of teams within the business, so an organization is characterised by groups, users and the role of users in the groups.
- Actor: An actor is a placeholder specified in the process definition for the users. Using an actor instead of specifying real people directly makes the process definition more flexible: when the list of users changes, only the process configuration need to be changed. Making the connection between the actor definition and the set of users is called actor mapping.

**Now that some of the key concepts are clearer, we shortly describe the relationships between the different entities in the diagram of**

Figure 19. The entity “role” represents the role of a user in a group, and “group” represents the group a user belongs to. There is a membership when a user belongs to both (a group and a role). A user may have more than 1 membership defined (i.e. different roles in different groups). An example of a membership might be: Mary (the “user”) is a member (the “role”) of the Laboratory department (the “group”). Therefore, using the “membership” entity, the developer may characterise any organisation by distributing the staff (i.e. user) in different departments (i.e. group) and with different functions (role).

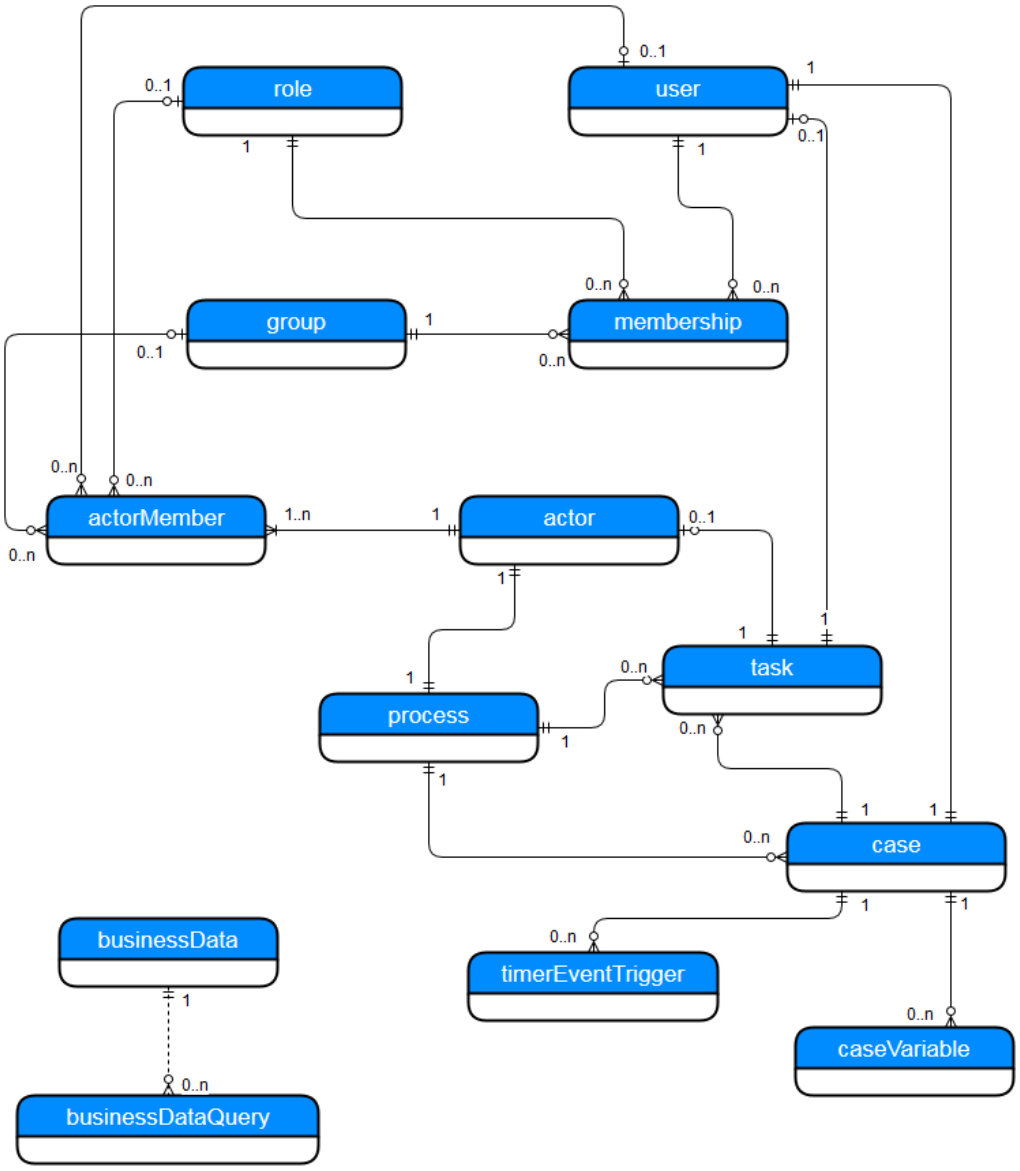


**Figure 19. BPDM for the workflow engine**

The “actor” entity has a similar meaning as the actor of a use case definition; it is a generic categorization of the person who can perform a process. Examples of actors in LetItFlow might be a nurse, a laboratory technician, etc. By using the “actorMember” entity, the developer will be able to establish the mapping between a generic actor of a process and a specific user, group or role.

The “process” entity models a process, which is a sequence of tasks represented as a flow diagram. A “case” entity represents a process instance. The “case” entity is related to a “process” and also to a “user” entity, which represents the user that started the case.

Regarding the “task” entity, there is a relation to the process and case which contain the task, and also to the user who executes the task (only in case it has been already executed). Optionally, it can be defined the actor that can execute this task.

A “case” may contain a timer trigger which serves to execute events based on timers. Timer events are used: to start a process on a repeating schedule, at a fixed date and time, or after a specified

interval (start timer), to delay a process for a set amount of time or until a specific date and time (intermediate timer) or to count down during a task until a deadline is reached (boundary timer or non-interrupting boundary timer).

Finally, regarding variables, we establish a difference between variables with Process scope ("caseVariable") and variables with Business scope ("businessData"). The "caseVariable" entity refers to Process variables which can be used globally within a Process instance, or limited to a specific Task. In this case, the data will not exist after the Process instance is completed. On the other side, business variables are used to represent business concepts (defined in a Business data model that can be created with the Bonita BPM Studio) which will be persisted automatically. These data can exist outside Process instances and be accessible to multiple Processes. We represent this kind of data with the "businessData" entity. Apart from this, the Bonita BPM Studio allows to define customized queries for retrieving the business data. Such queries are represented by the "businessDataQuery" entity.

# 9. BIBLIOGRAPHY


1. https://developer.android.com
2. http://www.iso.org/iso/iso8601
3. http://www.bonitasoft.com/
4. http://www.json.org/
5. http://www.slideshare.net/kmstechnology/introduction-to-agile-software-testing.
6. https://www.gov.uk/service-manual/making-software/testing-in-agile.html.
7. Antony Colfelt (2010). ***Bringing User Centered Design to the Agile Environment*** (http://boxesandarrows.com/bringing-user-centered-design-to-the-agile-environment/).