



INTEGRATION GUIDELINES

Deliverable nº: D6.1



EC-GA Number: 723737
Project full title: HUMAN MANUFACTURING



Work Package: **WP6**
Type of document: **Deliverable**
Date: **15/04/2018**

Grant Agreement No 723737

Partners: **Holonix**

Responsible: **Holonix**

Title: **D6.1. Integration Guidelines** Version: 4 Page: 0 / 20

Deliverable D6.1 **Integration Guidelines**

DUE DELIVERY DATE: 12/01/2017

ACTUAL DELIVERY DATE: 15/04/2018

Document History

Vers.	Issue Date	Content and changes	Author
0.0.1	13/12/2017	Document structure	Djennadi Rabah
0.1	14/01/2018	First version for peer review	Djennadi Rabah, Stefano Borgia, Eva Coscia, Alessandra Gulotta
0.2	09/02/2018	Version after review.	Djennadi Rabah, Stefano Borgia, Eva Coscia, Alessandra Gulotta
0.3	09/03/2018	New version after the second review	Djennadi Rabah, Stefano Borgia, Eva Coscia, Alessandra Gulotta
0.4	06/04/2018	New version after the third feedback	Stefano Borgia

Document Authors

Partners	Contributors
Holonix	Djennadi Rabah, Stefano Borgia, Eva Coscia, Alessandra Gulotta

Dissemination level: CO

Document Approvers

Partners	Approvers
LMS	Anna Karvouniari
SINTEF	Felix Mannhardt

Executive Summary

This document reports the achievements from task T6.1 and presents principles and guidelines that drive the integration of all the HUMAN components developed during the activities in WP2-WP5. The concept of integration can be further specified based on various levels: with specific reference to the HUMAN platform, the focus is on data-level and application-level integration and interoperability, as the integration process described in this document aims at connecting different components and sub-systems, developed by different IT partners of HUMAN, to finally release the HUMAN system that behaves accordingly with the requirements expressed by the end users (D1.1) and implements the architecture described in D1.4. Integration with other systems (business-level integration) or at UI level are out of scope.

The proposed methodological solutions allow the integration process to be a continuous one, able to include new versions of the existing components, which will be available during the execution of the project, as well as integrate new future components.

The objective of D1.4 is to provide a document representing a simple user guide for HUMAN IT partners in the development of their components, to ensure easy integration with the rest of the system, but also for those who will develop a new service or want to integrate a new sensor device.

The architectural solutions which enables integration are explained and examples for integration of components in the HUMAN architecture are provided. Tools and technologies used to implement the integration solutions between the HUMAN components are introduced, with motivations for their choice. Moreover, the deliverable presents the testing procedures to check that the integrated solution behaves as expected, in terms of data flow and functionality, together with test plans, test beds and a bug reporting tool.

TABLE OF CONTENTS

1. Introduction	7
1.1 SCOPE	7
1.2 CONTEXT	8
1.3 STRUCTURE OF THIS DELIVERABLE	9
2. Features supporting integration and interoperability	10
2.1 INTRODUCTION	10
2.2 COMMUNICATION LAYERS	11
2.2.1 MQTT BROKER	12
2.2.2 EVENT BROKER (KAFKA)	12
2.2.3 MESSAGE SCHEMA	12
2.3 DATA MODELS AND APIS	15
3. Continuous integration: principles and guidelines	20
3.1 INTRODUCTION	20
3.2 USERS INVOLVED INTO THE INTEGRATION PROCESS	20
3.3 ADDITION OF A NEW SERVICE WITHIN HUMAN PLATFORM	21
3.3.1 SPECIFIC REQUIREMENTS	21
3.3.2 WEB ADDRESSES / ACCESS POINTS OF THE CURRENT DEPLOYMENT	21
3.3.3 INTEGRATION GUIDELINES	21
3.4 ADDITION OF A NEW IOT DEVICE WITHIN HUMAN PLATFORM VIA MQTT	26
3.4.1 SPECIFIC REQUIREMENTS	26
3.4.2 WEB ADDRESSES / ACCESS POINTS OF THE CURRENT DEPLOYMENT	26
3.4.3 INTEGRATION GUIDELINES	27
3.5 ADDITION OF A NEW IT-COMPONENT WITHIN THE CORE OF HUMAN PLATFORM	29
3.5.1 SPECIFIC REQUIREMENTS	29
3.5.2 WEB ADDRESSES / ACCESS POINTS OF THE CURRENT DEPLOYMENT	29
3.5.3 INTEGRATION GUIDELINES	29
4. Technologies supporting the integration	31
4.1 TECHNOLOGIES FOR DATA INTEGRATION	31
4.2 TECHNOLOGIES FOR APPLICATION INTEGRATION	32
5. Integration testing plan	34
5.1 TESTING PLAN OVERVIEW	35
5.2 PHASE 1	36
5.3 PHASE 2	37
5.4 PHASE 3	37
5.4.1 CORE MODULES – KAFKA INTEGRATION	38
5.4.2 SERVICES – KAFKA INTEGRATION	38
5.5 PHASE 4	39
6. Bug reporting	41
6.1 REDMINE OVERVIEW	41
7. Conclusions	43
Appendix A. Current access points and URLs	44
Appendix B. Current HUMAN repositories on Gitlab	44
Appendix C. First proposal of testing procedures	45
APPLICATION INTEGRATION TEST	45
DATA INTEGRATION TEST	46
EVENT, INTERVENTIONS AND FACTORY MODEL	51
WORKER MODEL	56
Appendix D. References	59

Acronyms

Acronym	Explanation
ActiveMQ	Apache ActiveMQ TM (messaging and Integration Patterns server)
AMQP	Advanced Message Queuing Protocol
API	Application programming interface
BLE	Bluetooth Low Energy
BVP	Blood Volume Pulse
CRUD	Create-Read-Use-Delete functionalities for data management
GSR	Galvanic Skin Response
HTTP	HyperText Transfer Protocol
HUMAN	HUMAN MANUFACTURING
IoT	Internet of Things
ISO	International Organization for Standardization
KIT	Knowledge In Time (HUMAN service)
LWM2M	Low machine to machine
M2M	Machine to Machine
MCP	message communication protocol
MQTT	Transport or Message Queue Telemetry Transport
REST	REpresentational State Transfer
SII	Shopfloor Insight Intelligence (HUMAN service)
SOAP	Simple Object Access Protocol
STOMP	the Simple (or Streaming) Text Orientated Messaging Protocol
TCP	Transmission Control Protocol
UUID	Universally unique identifier
WOS	Workplace Optimisation Service (HUMAN service)
WP	Work Package

1. Introduction

In this chapter, the purpose and context of this document are outlined, relations with activities in the project are presented and the organisation of the document is explained.

1.1 SCOPE

This deliverable reports the outcomes of activities executed in Task T6.1 of the HUMAN Project, whose objective is: *"define the integration principles and guidelines to be followed by the development activities in WP2-WP5 to ensure that the final results of the specific WPs will be easily integrated. In tight connection with T1.5, data formats ensuring interoperability as well as the infrastructure and environments for the service delivery are defined, following an iterative and incremental approach. The task will define methodology and technologies for continuous integration, test beds and bug reporting. The testing procedures to check the correctness of the integrated solution are defined, to be executed in T6.2. This task will set the scene for T6.2."*

The purpose of D6.1 is to provide the guidelines to ensure the successful integration of the WP2-WP5 results into the industrial scenarios.

These guidelines consists of:

1. Clarification of integration levels to be achieved for the delivery and customisation of the HUMAN system (chapter 2)
2. Integration solutions implemented in the HUMAN architecture: adoption of communication brokers and APIS (chapter 2)
3. Methodology for the continuous integration, with operative guidelines (chapter 3)
4. Technologies adopted to implement the integration solutions (chapter 4)
5. Testing phases and guidelines for testing (chapter, to be implemented and reported in D6.2)
6. Information on where the integration solutions are deployed and how to access them (Appendix A)

With reference to point 2), it has to be clarified that these solutions are currently under development in WP3 and they will be available for the integration to be completed by M20. This document presents the draft status of API, data models, broker implementation that are in the scope of tasks T3.1, T3.2 and T3.3. Results of the integration activities will be reported in D6.2. Details for accessing the different components of the architecture are shared by the consortium partners using a web tool pointing to technical documentation, and will be reported in a web page.

Since HUMAN adopted an agile approach, aimed at providing different releases of components, which reflect a better understanding of users' needs and expectations and to solve technical issues, it is

important to remark that the deliverable explains how the presented solutions allow the integration process to be a continuous one, able to include new versions of the WP2-WP5 results, which will be available during the execution of the project, as well as new future IT components.

1.2 CONTEXT

This deliverable provides a description of the technical solutions developed to ensure the integration of the various HUMAN components that have been identified in the architecture described in D1.4, together with guidelines to drive the implementation of IT components in the four main technical WPs (WP2, WP3, WP4 and WP5); the guidelines, integrated with more technical details from T3.1, will support their successful integration in the unique HUMAN system that will be deployed in the industrial scenarios, as will be planned and reported in D6.2.

In the first phase of HUMAN, the expected audience of D6.1 are IT partners that are in charge of developing the technical results in WP2-WP5, but, afterwards, the integration guidelines are expected to be adopted also by other external developers that may wish provide new services or new hardware devices (e.g. the ones for sensing ambient conditions) to be integrated in the HUMAN system.

This deliverable sets the scene for task T6.2 and contents of deliverable D6.2, aimed at reporting the results of the deployment. The outcomes of the adoption of the integration solution and of the integration testing will be also redirected to T1.4 to support the refinement and consolidation of the final HUMAN architecture that will be presented in D1.5.

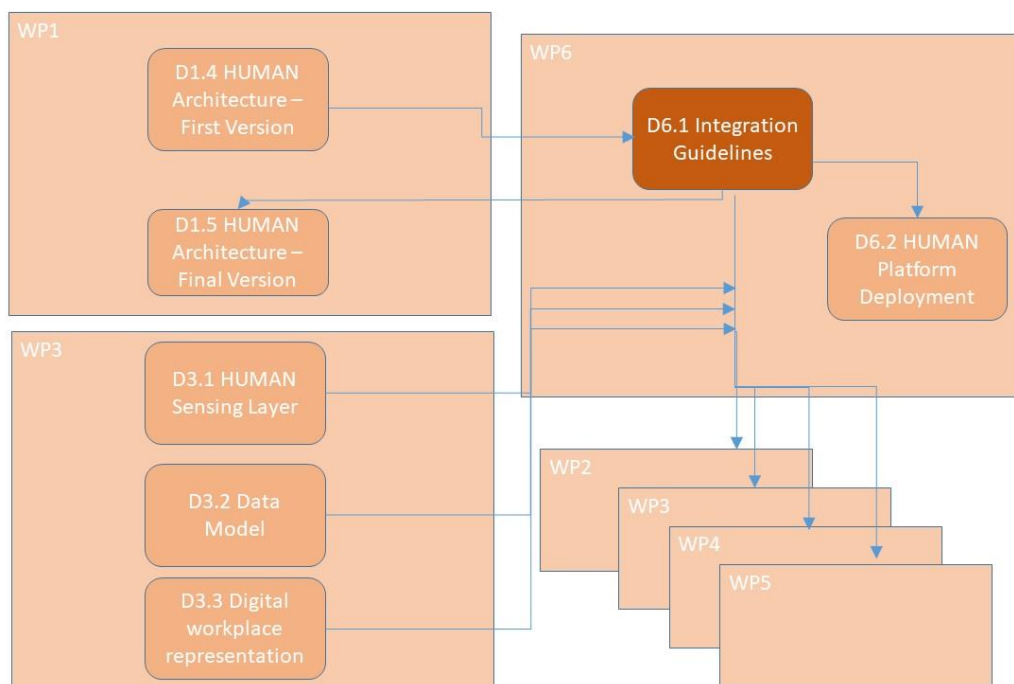


Figure 1: D6.1 positioning in the overall project

1.3 STRUCTURE OF THIS DELIVERABLE

The deliverable consists of the following sections:

- Section 1: introduces the objectives of the document and its relation with other tasks and work packages.
- Section 2: describes the selected features to ensure integration and interoperability at the level of data and application.
- Section 3: summarizes the adopted principles supporting continuous integration and defines the integration guidelines.
- Section 4: provides an overview of the technologies used for implementing the HUMAN solution, underling the aspects enabling the integration.
- Section 5: proposes strategy and planning of integration tests.
- Section 6: delivers a brief introduction to the bug reporting tool Redmine.
- Section 7: offers conclusions on the work done so far and plans for the next further actions to refine and finalise the integration.

2. Features supporting integration and interoperability

2.1 INTRODUCTION

The term integration refers to the process of linking together different IT hardware and software systems through the use of software features and architectural solutions, so that the resulting system works together as whole, offering the expected behaviour. In general, four most common levels of integration that an IT-based system are the following:

1. Data-level integration. It refers to solutions enabling the flow of data among different applications that need to share them and possibly to make available to other applications the results of their elaborations on these data. It is usually defined at the level of database and it is composed of data batch transfer, data merging and replication, ETL solutions (Extract, Transform, Load). The most common solutions for addressing the data-level integration consists of the definition of data formats and services that have control on the shared data, so to avoid the risks of having multiple independent and difficult to control accesses to the data.
2. Application-level integration (referred to different applications at a functional level). It is usually obtained through the adoption of request / response paradigm or middleware tools that enable communication and management of data, consolidating and federating integration architecture.
3. Business process-level integration. It refers to the integration of different IT assets (single applications or systems) which resides in different locations of an enterprise and now also on the Cloud, to implement the logical business process of a Company or of an ecosystem. This integration identifies how steps in a workflow are supported by IT assets and defined how to orchestrate their interactions.
4. Presentation-level integration. It consists of the standardization of the user interfaces inside a whole single common model, usually web-based portal. It was previously used to integrate applications that could not otherwise be connected, but applications integration technology has since evolved and become more sophisticated, making this approach less prevalent.

The objective of D6.1 is to cover only the first two levels: proposed data formats, communication services, integration methodology and technologies used to implement the integration are described in this document. In particular, the main features supporting integration and interoperability in the Human platform consists of: the modularity of the whole platform, the adoption of a unique database repository for data, the use of the communication brokers. All these elements have

been taken into account during both the concept and the design of the HUMAN architecture, presented in details in deliverable D1.4. A simplified version of this architecture is reported below, in Figure 2, with the specific and restricted goal to underline which the parts offers solutions for the integration.

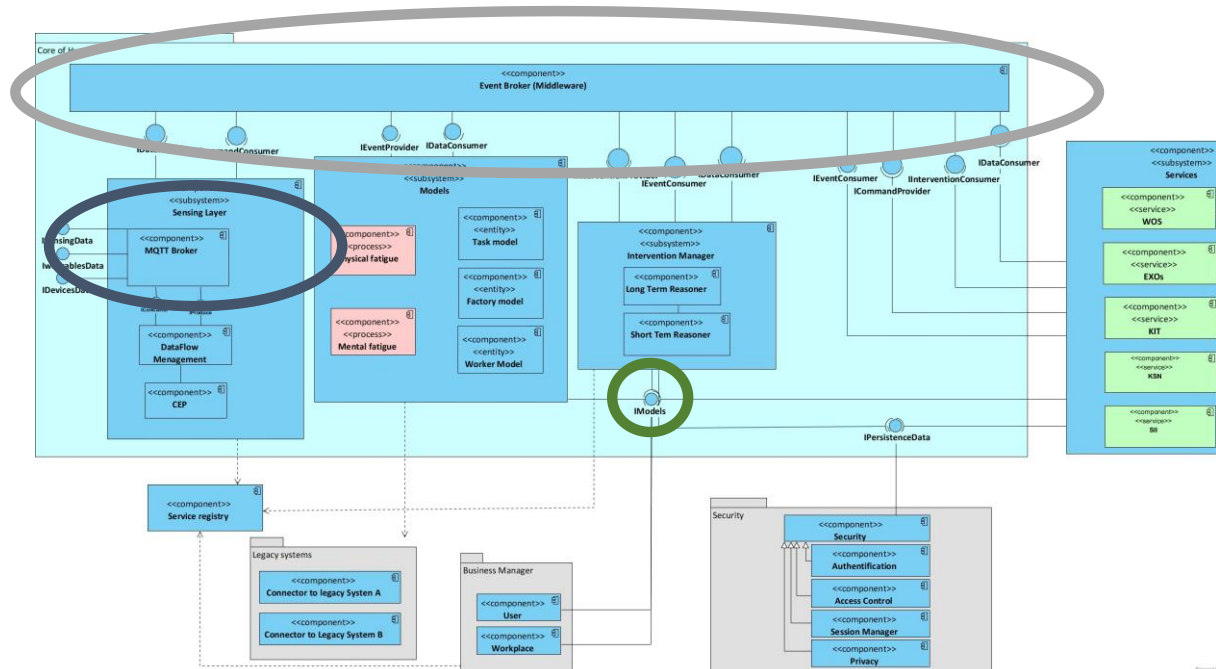


Figure 2: HUMAN architecture with interoperability-enabling elements

The circles in Figure 2 highlight the three main integration elements: the iModels connector (in the green circle), providing the unique access point to the HUMAN models (data-level integration), and the two brokers that centralize the exchange of data between HUMAN core applications and external applications or IoT devices (application-level integration).

Note that the manual programming of point-to-point interactions between two IT components has been avoided: even if it represents the easiest way to integrate applications, it is of course a solution that cannot support scalability of the system, as the addition of new modules will require developing additional ad hoc integration solutions.

2.2 COMMUNICATION LAYERS

Considering the communication layers, HUMAN architecture presents two brokers that provide interface respectively towards the devices and the external application (i.e. the HUMAN services)

2.2.1 MQTT BROKER

In HUMAN context, IoT sensors provide data representing parameters of the physical environment and of the worker. This flow of field data, measured by physical sensor devices and coming into the HUMAN system, is managed by the use of a Machine to Machine (M2M) broker representing a low-level middleware (interface spotted by the blue circle in Figure 2). MQTT (Message Queue Telemetry Transport) is a lightweight protocol based on the publish/subscribe pattern over TCP/IP protocol. MQTT requires a broker (in this case the aforementioned M2M broker) to be present, distributing messages to the interested subscribers based on the topic of the message. It represents a communication protocol for use between client software on a M2M device and server software on a management / service platform.

2.2.2 EVENT BROKER (KAFKA)

A specific interface allows system core components to communicate with higher-level components: the high-level middleware (interface depicted inside the brown circle in Figure 2), mainly designed for exchanging of data between HUMAN applications, consists of Kafka event broker. Kafka is a publish-subscribe messaging system that maintains feeds of messages in topics. Producers write data to topics and consumers read from topics. The data exchange mechanism implemented by the event broker is based on definition and implementation of the message communication protocol (called HUMAN MCP). Different types of messages are used on the broker, with the goal of structuring different types of information. The structure of all the HUMAN messages is represented by a common schema.

2.2.3 MESSAGE SCHEMA

Each message of the HUMAN MCP consists of a header part and a payload part. MCP uses AVRO Schema [1] to define the message schema and validate its syntax. The messages are firstly encoded in AVRO, and then serialized in Binary. Each message consists of a header and a payload. The structure of the header is the same for every message: defining one single header implies that broker users share the same definition of payload and can recognize the type of the message. This choice appears useful for recording all events in a common format for later analysis and for easier filtering, avoiding clashes, etc. A versioned schema is defined for the header and each supported message type.

Message header provides information defining the payload: source of payload (key used for the filtering of the data), timestamp / position (info used for creating the key of each data record within the repository), type of payload and other optional fields. Instead, message payload data contain the relevant characteristics of the message.

The header part of a message contains metadata, which are common to all types of messages: (see Table 1)

Table 1: Message schema header

Key	Type	Description
schema	Integer	Version number of the schema used.
source	String	Identifier of the source system or source device sending the messages. For example, a serial number or a similar persistent identifier should be used.
position_longitude	decimal degrees	Angular geographic coordinate specifying north-south position of a point on Earth's surface. Optional field.
position_latitude	decimal degrees	Angular geographic coordinate specifying east-west position of a point on Earth's surface. . Optional field.
session	String	UUID identifying the session for which the message was send. Optional field that may be omitted if no session is active or if a session is not applicable (e.g., sensor).
instance	String	UUID identifying the case for which the message was send. Optional field that may be omitted if no instance is active or if the message is not related to a particular instance.
time	Long (UNIX)	Millisecond-timestamp of the source system (so, it represents the time when the event happened)
type	String	The type of the message expressed in a reverse domain name notation: "org.human.taskmodel". The type determines the schema used to en-/decode the payload of a message.
payload	String	The actual payload of the message. The schema used to parse the payload is determined by the field '\$type'.

Customized payloads contains specific information about jobs, task, physiological data, support level, stress, head position, etc. to be exchanged between Data Models and the Human components and services. Currently, the following types of payload have been defined

- “Session” (Table 2)
- “Job” (Table 3)
- “Task” (Table 4)
- “Physiological” (Table 5)
- “Support” (Table 6)
- “Stress” (Table 7)

“Part” (

- Table 8)

Table 2: “Session” message payload

Key	Type	Description
User	String	User
Operation	String	Operation that is executed

Table 3: “Jobs” message payload

Key	Type	Description
basaPathUrl	String	
sceneName	String	
sceneBundleName	String	

Table 4: “Task” message payload

Key	Type	Description
Operation	String	Operation that is executed: ["Start", "Complete"]
TaskId	String	Identifier of the task that the message refers to.

Table 5: “Physiological” message payload

Key	Type	Description
Operation	String	Operation that is executed: ["Start", "Complete"]
Task	String	Identifier of the step that the message refers to.
Id	String	Id of worker
TMP	String	Timestamp
HR	Int	Heart rate
BR	Int	Breath rate
ST	float	Skin temperature
BP	Type symbols [UPRIGHT, PRONE, SIDE, UNDEF]	Body position
AS	Type symbols [STATIONARY, MOVINGFAST, MOVINGSLOWLY]	Ambulation status

Table 6: “Support” message payload

Key	Type	Description
Level	Number	Level of detail requested for the augmented reality support from 0 to 10.

Table 7: “Stress” message payload (single measurement stress sensor)

Key	Type	Description
Stress	String	Level of stress.

Table 8: “Part” message payload

Key	Type	Description
Operation	String	Operation that is executed

The Human message schema is managed in Gitlab.

1. Each message payload is defined in a separated file, instead a “master” message contains the information about the header. Modification and updates can be made in the relative local files, which have to be committed and pushed into the web repository. Versioning is managed by Git distributed version control system: advanced features such as fully distributed operation, guarantees of content integrity using cryptographic checksums, foreword and backward compatibility control, etc. are provided. The developers can attach comments explaining the changes. In order to maintain a correct cloud / local synchronization between, checks and pulls of the message schema updates from remote repository have to be executed.
2. A Java application performs the merge of the single fragments into a whole message schema that represents the effective reference schema for the messages exchanged into the Human middleware.
3. The generated message schema is then automatically pushed into the software tool Schema Registry (integrated within the broker Kafka), which is used to manage the serialization of the messages.

As Kafka broker consists of a multi-channel system, each message type is assigned to a univocal topic on which the information can be published and consumed. Currently, the current list of the available topics is reported in the following, together with the relative exchanged message type:

- "EU_HUMANMANUFACTURING_JOB" topic for job message type
- "EU_HUMANMANUFACTURING_PART" topic for part message type
- "EU_HUMANMANUFACTURING_TASK" topic for task message type
- "EU_HUMANMANUFACTURING_SUPPORT" topic for support message type
- "EU_HUMANMANUFACTURING_STRESS" topic for stress message type
- "EU_HUMANMANUFACTURING_PHYSIOLOGICAL" topic for physiological message type
- "EU_HUMANMANUFACTURING_INTERVENTION" topic for intervention manager output

2.3 DATA MODELS AND APIS

WP3 is in charge of designing and implementing Data Models, which are a representation of context-related information shared by the HUMAN components. Such implementation does not consist only of a structure to represent these data, but also of the HTTP API to access them. APIs represent an

element for data-level interoperability, and are designed in order to provide to the core Component the access to the data of the Models (about worker, factory, task, jobs, etc.). According to the work package structure, T3.2 aims at formalizing the Data Models. Instead, T3.3 will focus on the definition and implementation of DB architecture and of APIs (called iModels APIs). As the scope of the deliverable is to provide integration guidelines and not the details of the implementation, in the current section an overview of the first release of the APIs are presented, together with their preliminary definition. Further extensions to APIs will be proposed in D3.3.

REST (REpresentational State Transfer) approach to implement APIs has been adopted. The REST is an architectural style, and an approach to communications that is often used in the development of Web services. Its decoupled architecture, and lighter weight communications between producer and consumer, make REST a popular building style for cloud-based APIs, such as those provided by the most common cloud provides. When web services use REST architecture, they are called RESTful APIs (Application Programming Interfaces) or REST APIs. HUMAN platform is also based on the REST approach, exposing the relevant entities of the data model and operating over them via HTTP verbs. The REST approach grants easy to use and portable API, leveraging on the HTTP protocol, well known by developers and widely supported on all platforms. So, HUMAN iModels APIs are implemented with web services and use JSON data encoding for information exchange, ensuring high interoperability. A client or user is able to invoke a web service by sending a message and then in turn gets back a response message.

As already explained in deliverables D1.4 and in D3.2, data that are of interest for several components are stored into a persistency solution and made available from a unique interface allowing the CRUD (create, retrieve, update, delete) operations. The interface provides a set of queries, implementing all the requirements of the HUMAN services. Firstly, an overview of the structure of the data shared by the HUMAN components, defined as part of T3.2 task, is depicted in Figure 3.

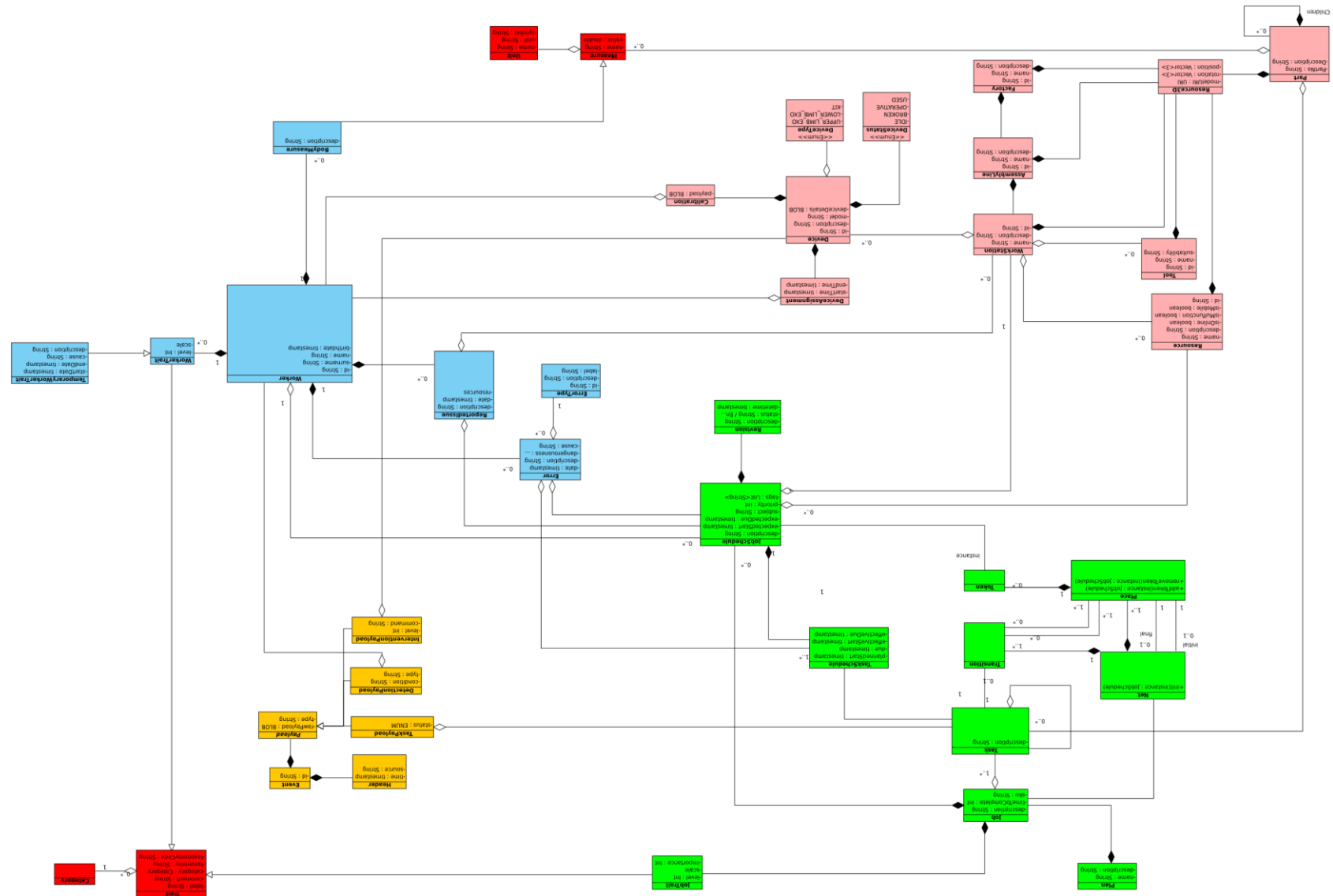


Figure 3: Shared Data model

Specifications of the first version of the APIs are provided, reflecting the queries and operations that the different components of the HUMAN system require on the shared data. The final one, including all the details, will be included in D3.3. The developed APIs are grouped in relation to the data of the considered model (worker, task, factory, event and intervention): main details are provided in the following tables.

General details of the APIs

- Format: JSON
- Security issue: in order to access to the data model APIs, login is required. The calling of the API for the login allows the user the authentication and the access to the APIs, with permissions depending on the user type.
- HTTP Request: POST, GET, PUT, DELETE
- Response types: HTTP status codes, HTTP body. (When the specific required entity is not found, empty JSON array is returned)

Table 9: List of APIs for Worker data

Method	HTTP request	Description
GET	/workers/workers	Gets the list of information related to all workers
GET	/workers/anthropometry	Gets the information regarding the anthropometry of workers
GET	/workers/anthropometry/{workerId}	Gets the anthropometry of a specific worker. Parameters: <i>workerId</i> (string)
GET	/workers/exoskeletonSettings/{workerId}	Gets the exoskeleton setting related to the worker. Parameters: <i>workerId</i> (string)
POST	/workers/startShift/worker/{workerId}/jobSchedule/{jobScheduleId}/workPlace/{workPlaceId}	Publishes information in the system that Worker (identified by <i>workerId</i>) started <i>Job</i> (identified by <i>jobId</i>) on <i>Workstation</i> (identified by <i>workstationId</i>)
POST	/workers/stopShift/worker/{workerId}/jobSchedule/{jobScheduleId}/workPlace/{workPlaceId}	Publishes information in the system that <i>Worker</i> (identified by <i>WorkerId</i>) ended his working shift.

Table 10: List of APIs for Task data

Method	HTTP request	Description
GET	/jobs/jobs	Gets all the jobs
GET	/jobs/jobsSchedule/{start}/{end}	Gets the list of scheduled jobs. Parameters: start date (string) and end date (string)
GET	/jobs/job/{jobId}	Gets the information regarding a specific job. Parameters: <i>jobId</i> (string)
GET	/jobs/job/{jobId}/petrinet/{petrinetId}	Gets a specific petrinet related to a specific job. Parameters: <i>jobId</i> (string) and <i>petrinetId</i> (string)
GET	/jobsSchedule/getJobSchedule/{jobScheduleId}	Gets the information related to specific jobSchedule. Parameters: <i>jobScheduleId</i> (string)

DELETE	/jobsSchedule/{jobScheduleId}	Removes the information related to specific jobSchedule. Parameters: <i>jobScheduleId</i> (string)
POST	/jobsSchedule/{jobScheduleId}/worker/{workerId}	Associates the Worker (identified by <i>workerId</i>) to a jobSchedule (identified by <i>jobScheduleId</i>). Parameters: <i>jobScheduleId</i> (string) and <i>workerId</i> (string)
POST	/jobsSchedule/{jobScheduleId}/workPlace/{workPlaceId}	Associates the Workplace (identified by <i>workPlaceId</i>) to a jobSchedule (identified by <i>jobScheduleId</i>). Parameters: <i>jobScheduleId</i> (string) and <i>workPlaceId</i> (string)
POST	/petrinet/upload	Loads a petrinet. Parameters: <i>key</i> =file, <i>value</i> =file path to petrinet.pnml (xml)
GET	/petrinet/{petrinetId}	Gets a specific petrinet
GET	/tasks/tasksScheduled/{start}/{end}	Gets a list of scheduled tasks. Parameters: <i>start</i> (long) and <i>end</i> (long)
GET	/tasks/task/{taskId}	Retrieves the information about a task. Parameters: <i>taskId</i> (string)
GET	/tasks/workingTask/{workerId}	Retrieves the information about a task executed by the specific worker. Parameters: <i>workerId</i> (string)
GET	/tasks/advanceInJob/worker/{workerId}/job/{jobId}	Returns the task on which the specified worker is currently working. Parameters: <i>workerId</i> (string), <i>jobId</i> (string)

Table 11: List of APIs for Factory and Device data

Method	HTTP request	Description
POST	/devices/initializeDevice	Initializes a device. Parameters: <i>description</i> (string), <i>deviceType</i> (int), <i>serialNumber</i> (long), <i>name</i> (string)
GET	/devices/getDevices	Gets all devices present In database
GET	/devices/getDevicesPendingRequest	Gets all the pending requests regarding the devices
POST	/devices/setDeviceData	Parameters: <i>id</i> (int), <i>topicStatus</i> (string), <i>topicIntervention</i> (string), <i>uuidDevice</i> (string), <i>userSSL</i> (string), <i>passwordSSL</i> (string, clear text)
GET	/devices/getDeviceParameters/serialNumber/{serialNumber}/deviceType/{deviceTypeId}	Gets all data regarding the specified device. Parameters: <i>name</i> (string), <i>uuidType</i> (String)
GET	/deviceTypes/getDeviceTypes	Gets the device types present within database
GET	/deviceTypes/getDeviceTypesPending	Gets all the pending requests regarding the device types
POST	/deviceTypes/setUuidDeviceType	Sets the uuid of the device type
POST	/workPlaces	Parameters: <i>name</i> (String), <i>description</i> (String)

3. Continuous integration: principles and guidelines

3.1 INTRODUCTION

The integration methodology proposed in this document is mainly based on the concepts of modularity and interoperability, but it is also thought and designed for allowing a continuous process of addition and integration of new components and delivery of functional and operational improvements, minimizing the risk associated to the change. In general, continuous integration can be viewed as a set of key features and practises, enabled by the choices at the level of architecture, technologies and tools, aiming at reducing this risk and compressing the development lifecycle: small integration batch sizes, comprehensive version control, modularity, bug reporting, etc.

At the application level, considering the HUMAN platform, integration and continuous integration can be exploited at two levels: the update or the introduction of an IT-component into the HUMAN core, of an IoT device (wearable device, sensor, etc.) or of a service.

Project partner software developers and, after the end of the project, external / new software developers can progressively contribute to make HUMAN platform grown, adding and integrating sensors, applications and services. The following guidelines are designed for authorized software developers. New applications can interact with the HUMAN core through the adopted brokers using the relative broker clients and query backend: so, a general requirement at the base of the integration process is to allow the communication on the middleware using Kafka clients. SW modules can also interact with database and other IT components using available HTTP Rest APIs.

3.2 USERS INVOLVED INTO THE INTEGRATION PROCESS

Given the continuous integration process defined and implemented in the Human platform, several stakeholders are involved, each one with specific tasks, access permissions and authorizations. Firstly, the actors assuming an active role in integration of devices, services and IT-components, are identified:

- “HUMAN Device provider” user = SW developer working at the integration of the IoT devices
- “HUMAN Service provider” user = SW developer working at the implementation and integration of services
- “HUMAN IT-component provider” user = SW developer working at the implementation and integration of a IT-Component (Models, Reasoners, etc.) into the Core of the HUMAN platform

- “HUMAN developer admin” user = SW developer responsible of the deployed HUMAN platform: this user is allowed to make specific implementation on the middleware, verifies the behaviour of the whole solution, and coordinates the deployment.

3.3 ADDITION OF A NEW SERVICE WITHIN HUMAN PLATFORM

The present section provides specifications and guidelines that drive a SW developer along the process of integrating a new service into the Human platform.

3.3.1 SPECIFIC REQUIREMENTS

The specific requirements related to the integration of new services are reported as follows:

- Kafka client library - current Kafka version 0.11.0.2 - related to the programming language used for SW developing.
- (MQTT client library related to the programming language used for SW developing)
- HTTP Request composer (e.g. Postman [2], etc.) or your own developed code to call HTTP API
- Wi-Fi internet network (in case the production deployment of the Human core is instantiated on a server)
- Access and use of the web-based Git-repository manager Gitlab and the project management web application Redmine (ref. chapter 4.1 for further detail of these technologies)

3.3.2 WEB ADDRESSES / ACCESS POINTS OF THE CURRENT DEPLOYMENT

Each Human end-user will have a specific production deployment, so the listed access assess points are referred to the current deployment on HOLONIX server.

- Middleware UI address: <http://ns3370643.ip-37-187-92.eu:22006/>
- Kafka broker address: tcp://ns3370643.ip-37-187-92.eu:22007
- MQTT broker: address tcp://mqtt1.holonix.biz ; port 8883
- APIs end point: <https://human.holonix.biz/models/api/> (this end point has to be concatenated with the specific API names in order to call the provided APIs)
- Gitlab repositories of Human SW developments (middleware, data model, core, intervention manager, whole platform): <https://gitlab.com/humanufacturing/integration>
- Gitlab repository of Human message schema: <https://gitlab.com/humanufacturing/messaging>

3.3.3 INTEGRATION GUIDELINES

The addition of a service in Human platform can require interoperability at three different level:

1. Integration with middleware (paragraph 3.3.3.1)
2. Integration with the data model (paragraph 3.3.3.2)
3. Integration with an IoT device (paragraph 3.3.3.3)

3.3.3.1 INTEGRATION WITH HIGH LEVEL BROKER (KAFKA)

The following steps must be executed for the integration with high-level broker (Kafka) in case a new component needs to produce/consume messages:

1. In case of new “HUMAN service provider” user, ask “HUMAN developer admin”
 - a. The access to HUMAN project repository on Gitlab
 - b. Authentication credential for Redmine
 - c. Authentication credential for the backend with role “HUMAN service provider”
2. Access the Gitlab repository with the files containing all the partitions of the message schema
3. Verify the existing partitions of message schema. If the new IT service requires a new type of event and message, create a new file defining the structure of the new payload associated to that event / message, then check its compatibility with the other schema partitions and with previous schema version: the Human whole message schema automatically evolves merging all the message schema partitions.
4. Download from Kafka confluence¹ the Kafka client related to the programming language used for developing your component.
5. Customize your Kafka client: create a specific “listener” class based on the message schema defined in HUMAN project, by importing the message schema in AVRO codec (an example of customized listener class created in Java development environment is reported in figure)
6. Upload your customized Kafka client on HUMAN code repository on Gitlab
7. Download / Clone the Kafka client from Gitlab repository in your local directory
8. Import the Kafka client into a “project” of your software development environment (e.g. in Eclipse, IntelliJ, etc.)
9. Download / Clone the message schema from Gitlab repository in your local directory
10. Import the message schema into a “project” of your software development environment (e.g. in Eclipse, IntelliJ, etc.)
11. Register your account of “HUMAN service provider” user calling the specific HTTP API POST for the registration

¹ <https://cwiki.apache.org/confluence/display/KAFKA/Clients>

12. Ask “HUMAN developer admin” user to update the middleware and to provide a new detection module into the HUMAN core (if it is required).
13. Ask “HUMAN developer admin” to set and configure the intervention manager in order to define and publish on the defined Kafka broker topic the intervention required by the service. The docker compose of the HUMAN core on Gitlab and the central deployment are updated.
14. If the choice is to work with a Human local deployment, download from Gitlab repository the Docker compose of the Human Core; otherwise, use the online central deployment.
15. Login as “HUMAN service provider” user into the HUMAN platform calling the specific HTTP API for the login
16. Run a unit test that opens a connection to Kafka, produces and consumes an event for testing
17. In your own software project implementing your service, create the dependencies to Kafka client project and to the message schema project and start the development of the new service that consumes data on the topic “EU_HUMANMANUFACTURING_INTERVENTION”
18. The “HUMAN developer admin” sets and configures the intervention manager in order to define and publish on the defined Kafka broker topic the intervention required by the service.
19. Make specific tests on your SW module implementing the new service.
20. Create a Docker image for your Human service and upload it on Gitlab repository so that the can be added to the Human platform Docker compose.

```
public abstract class Listener implements MessageListener<String, ByteBuffer> {}

private static final Logger LOGGER = LoggerFactory.getLogger(MessageListener.class);
private MessageFilter messageFilter = new MessageFilter();

abstract public void onMessage(Message message);

@Override
public void onMessage(ConsumerRecord<String, ByteBuffer> record) {
    Message message = new Message();

    try {
        message = Message.fromByteBuffer(record.value());
    } catch (IOException e1) {
        throw new RuntimeException(e1);
    }
    ConsumerRecord<String, Message> consumerRecord = new ConsumerRecord<String, Message>(record.topic(),
        record.partition(), record.offset(), record.key(), message);

    if (this.messageFilter.filter(consumerRecord)) {
        LOGGER.info("Received message " + consumerRecord);
        onMessage(message);
    } else {
        LOGGER.info("Filtered out message " + consumerRecord);
    }
}

/**
 * @return the messageFilter
 */
public MessageFilter getMessageFilter() {
    return messageFilter;
}

/**
 * @param messageFilter
 *      the messageFilter to set
 */
public void setMessageFilter(MessageFilter messageFilter) {
    this.messageFilter = messageFilter;
}
```

Figure 4: Screenshot of Java listener class customizing the Kafka client.

Notes

1. For some programming languages (e.g. Java), a customized version of the Kafka client is already available in HUMAN code repository on Gitlab
2. It is possible to change Kafka end point in the configuration file
3. The next evolution of the HUMAN platform will regards the issues of security and user authentication, with the expected adoption of SSL client certificates in order to establish the connection to Kafka broker instantiated on the central cloud deployment of the Human platform: it will be possible to download an SLL key from a link of the middleware front-end and use it to configure the used Kafka Client
4. Procedure for the implementation of push notification to the IoT devices is on going

3.3.3.2 INTEGRATION OF A NEW SERVICE WITH DATABASE, DATA MODEL OR OTHER IT COMPONENTS

1. Call the available HTTP Rest APIs provided by data model or other IT components in order to access information about Worker, Factory, Task, etc.

Notes

1. List of available APIs and the relative documentation will be available on Redmine. Deliverable D3.3 will describes data model implementation and APIs.

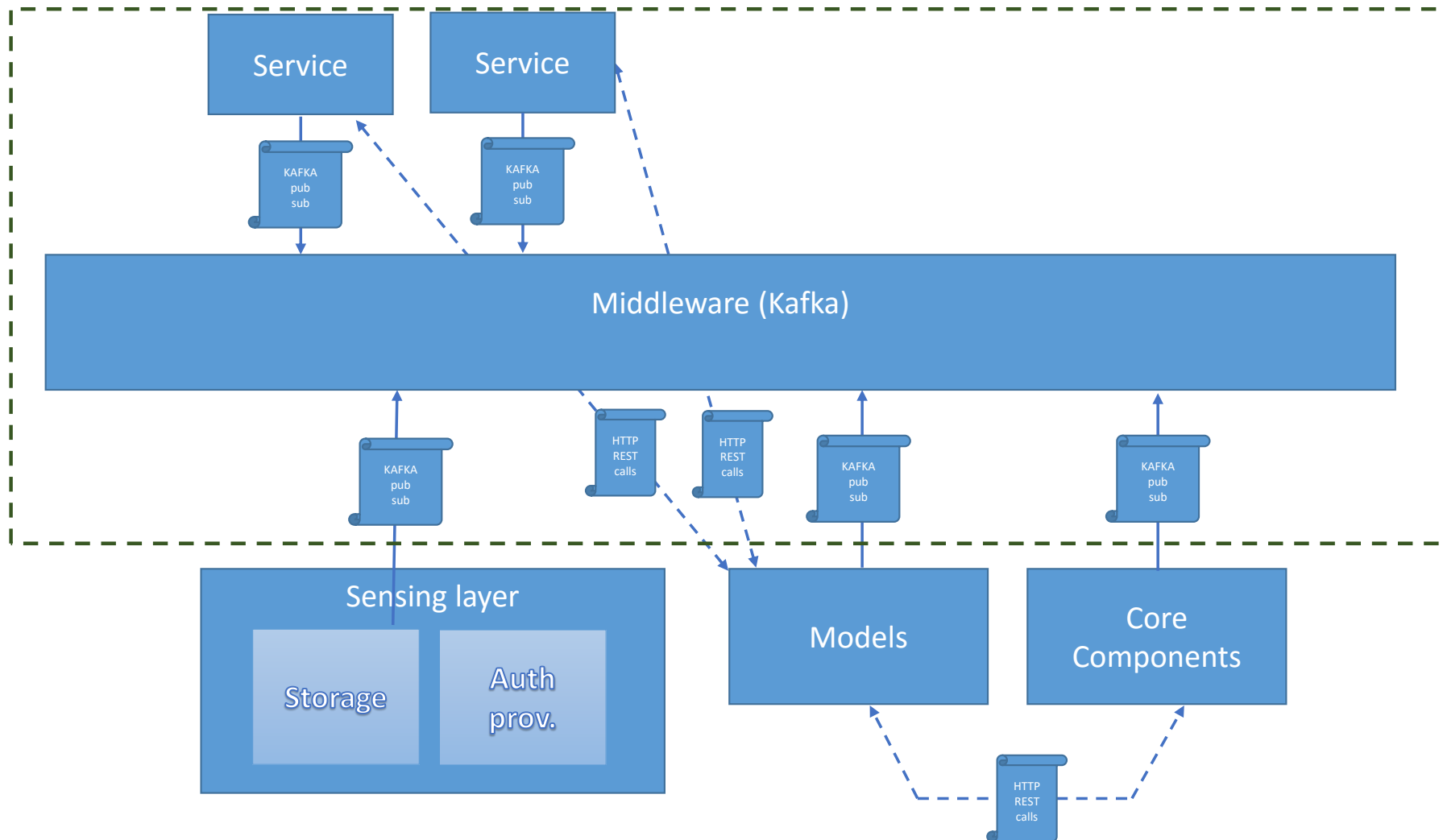


Figure 5: Integration of new services.

3.3.3.3 INTEGRATION OF A NEW SERVICE WITH THE IOT DEVICES (MQTT)

The following steps are to be followed in case a new service needs to communicate with the IoT devices through MQTT:

1. Download from the web the MQTT client library related to the programming language used for developing your service
2. Import the MQTT client library into a “project” of your software development environment (e.g. in Eclipse, IntelliJ, etc.)
3. In your own software project implementing your service, create the dependencies to MQTT client project and start the development
4. If central development is used, set the MQTT address. Otherwise, in case of local deployment of the HUMAN Core, set the address of the local host
5. Detect / select the MQTT topic on which the device is “listening” and send information with Json format to the device through the topic using MQTT library functions

3.4 ADDITION OF A NEW IOT DEVICE WITHIN HUMAN PLATFORM VIA MQTT

The present section provides specifications and guidelines that drive a SW developer along the process of integrating a new device into the Human platform. Considering Human architecture, the IoT devices have to measure and send sensor data to the sensing layer of the Human core using a MQTT broker.

3.4.1 SPECIFIC REQUIREMENTS

1. IoT device with firmware, host environment and wireless interface communication
2. MQTT client library compatible with the device
3. HTTP Request composer (e.g. Postman, etc.) or your own developed code to call HTTP API
4. Wi-Fi internet network (allowing sensor data communication)
5. Access and use of the web-based Git-repository manager Gitlab and the project management web application Redmine (ref. chapter 4.1 for further details of these technologies)

3.4.2 WEB ADDRESSES / ACCESS POINTS OF THE CURRENT DEPLOYMENT

Gitlab repository of Human message schema: <https://gitlab.com/humanufacturing/messaging>

Each HUMAN end-user will have a specific deployment, so the listed access points are referred to the current deployment on HOLONIX server.

- MQTT broker address: `tcp://mqtt1.holonix.biz`

- MQTT broker port: 8883
- APIs end point: <https://human.holonix.biz/models/api/> (this end point has to be concatenated with the specific API names in order to call the provided APIs)
- Gitlab repositories of Human SW developments (middleware, data model, core, intervention manager, whole platform): <https://gitlab.com/humanufacturing/integration>

3.4.3 INTEGRATION GUIDELINES

1. In case of new “HUMAN device provider”, ask “HUMAN developer admin”
 - a. The access to HUMAN project repository on Gitlab
 - b. Authentication credential for Redmine
 - c. Authentication credential for the backend with role “HUMAN service provider”
2. Configure the new IoT device setting the possible parameters concerning the MQTT
3. Download and install on the device host the related MQTT client (usually a library) for data exchange and management
4. Register your account of “HUMAN device provider” user calling the specific HTTP API POST for the registration
5. Login as “HUMAN device provider” user into the platform calling the specific HTTP API POST for the login
6. Register the new IoT device into the HUMAN platform and get the labels of the two topics that will allow respectively the publishing of data on MQTT (sensor data) and the consuming of data (intervention data, if required) calling the related HTTP API
7. Set the MQTT address in your device host environment
8. Define in your device host environment the specific methods for MQTT connection / disconnection, listening a topic, publishing and consuming data.
9. If possible, implement on your device an app with a basic front-end enabling the user to manage MQTT connection / disconnection of the device
10. Make specific tests on your IoT device.

Notes

1. Use a HTTP Request composer (e.g. Postman, etc.) or your own developed code to call the available HTTP Rest APIs

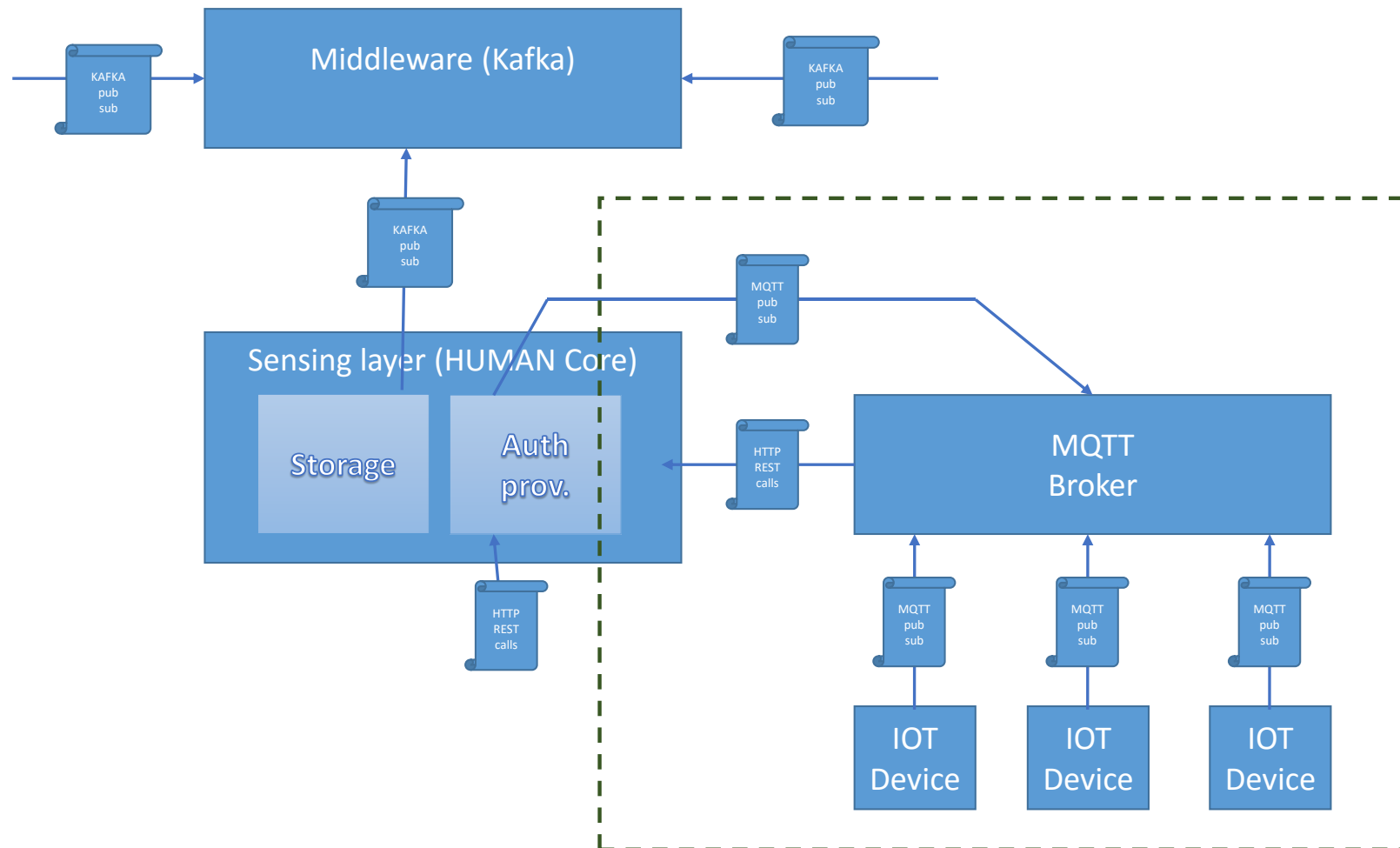


Figure 6: Integration of new devices.

3.5 ADDITION OF A NEW IT-COMPONENT WITHIN THE CORE OF HUMAN PLATFORM

3.5.1 SPECIFIC REQUIREMENTS

- The specific requirements related to the integration of new services are reported as follows:
- Kafka client library - current Kafka version 0.11.0.2 - related to the programming language used for SW developing.
- HTTP Request composer (e.g. Postman, etc.) or your own developed code to call HTTP API
- Wi-Fi internet network (in case the production deployment is instantiated on a server or in case of use of the central deployment of the Human Core for testing / development)
- Access and use of the web-based Git-repository manager Gitlab and the project management web application Redmine (ref. chapter 4.1 for further details of these technologies)

3.5.2 WEB ADDRESSES / ACCESS POINTS OF THE CURRENT DEPLOYMENT

Each Human end-user will have a specific production deployment, so the listed access assess points are referred to the current deployment on HOLONIX server.

1. Middleware UI address: <http://ns3370643.ip-37-187-92.eu:22006/>
2. Kafka broker address: <tcp://ns3370643.ip-37-187-92.eu:22007>
3. APIs end point: <https://human.holonix.biz/models/api/> (this end point has to be concatenated with the specific API names in order to call the provided APIs)
4. Gitlab repositories of Human SW developments (middleware, data model, core, intervention manager, whole platform): <https://gitlab.com/humanufacturing/integration>
5. Gitlab repository of Human message schema: <https://gitlab.com/humanufacturing/messaging>

3.5.3 INTEGRATION GUIDELINES

1. In case of new “HUMAN IT-component provider”, ask “HUMAN developer admin”
 - a. The access to HUMAN project repository on Gitlab
 - b. Authentication credential for Redmine
 - c. Authentication credential for the backend with role “HUMAN service provider”
2. Access the Gitlab repository with the files containing all the partitions of the message schema
3. Verify the existing partitions of message schema. If the new IT service requires a new type of event and message, create a new file defining the structure of the new payload associated to that event / message, then check its compatibility with the other schema partitions and with

previous schema version: the Human whole message schema automatically evolves merging all the message schema partitions.

21. Download from Kafka confluence² the Kafka client related to the programming language used for developing your component.
22. Customize your Kafka client: create a specific “listener” class based on the message schema defined in HUMAN project, by importing the message schema in AVRO codec (an example of customized listener class created in Java development environment is reported in figure)
23. Upload your customized Kafka client on HUMAN code repository on Gitlab
4. Download / Clone the Kafka client from Gitlab repository in your local directory
5. Import the Kafka client into a “project” of your software development environment (e.g. in Eclipse, IntelliJ, etc.)
6. Download / Clone the message schema from Gitlab repository in your local directory
7. Import the message schema into a “project” of your software development environment (e.g. in Eclipse, IntelliJ, etc.)
8. Register your account of “HUMAN It-component provider” user calling the specific HTTP API POST for the registration
9. If the choice is to work with a Human local deployment, download from Gitlab repository the Docker compose of the current Human Core; otherwise, use the online central deployment.
10. Login as “HUMAN IT-component provider” user into the HUMAN platform calling the specific HTTP API for the login
11. Run a unit test that opens a connection to Kafka, produces and consumes an event for testing
12. In your own software project implementing your service, create the dependencies to Kafka client project and to the message schema project and start the development of the new service that consumes data on the desired topic. Call the available HTTP Rest APIs provided by data model or other IT components in order to access information about Worker, Factory, Task, etc.
13. Make specific tests on your SW module implementing the new IT-component.
14. Push the software project of the new Human IT-component on Gitlab repository, so that the relative Docker image can be automatically generated, saved and, then, added to the Human Core Docker compose.

² <https://cwiki.apache.org/confluence/display/KAFKA/Clients>

4. Technologies supporting the integration

This section describes the tools used to implement the integration solutions between HUMAN components that have been introduced above.

Therefore, there are two groups of technologies:

- Technologies for data integration
- Technologies for application integration

4.1 TECHNOLOGIES FOR DATA INTEGRATION

The following table briefly summarizes the technologies used to support the implementation and deployment of APIs of applications of the HUMAN platform, in order to ensure the continuous integration when new versions are implemented and deployed.

Table 12: Technologies supporting APIs deployment

Technology	Description and enables for integration
Java Docs	Tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, interfaces, constructors, methods, and fields. The releasing of API should be documented by Javadoc tool and the pages generated will be exposed through Internet, mainly Javadoc offers some annotations that will allow us to describe the functionalities and the APIs that we have to implement. <u>Feature</u> enabling the integration: APIs documentation (a first version will be upload on Redmine, according to the evolution of works of Task 3.5)
Netflix Eureka Service registry)	REST (Representational State Transfer) based service that is primarily used in the cloud for locating services. Its main scope is to store information like service description and calling point, in order to easily retrieve and expose them. It also can manage the versioning of the registered services allowing the user to retrieve a specific version (when available). This tool does not directly call the services on behalf of the users, it can only store and retrieve the service information. <u>Feature</u> enabling the integration: possibility for service client and/or the routers to discover the location of service instances; APIs versioning management; global APIs storage.

4.2 TECHNOLOGIES FOR APPLICATION INTEGRATION

This section summarizes the technologies used to support the application integration between HUMAN components.

Table 13: Technology supporting application integration

Technology	Description and enables for integration
MQTT	ISO standard (ISO/IEC PRF 20922) publish-subscribe-based messaging protocol and works on top of the TCP/IP protocol. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. The publish-subscribe messaging pattern requires a message broker. Feature enabling the integration: communication through low level broker.
Kafka	Fault-tolerant publish-subscribe messaging system, based on topics. A topic is basically a FIFO (First In First out) list of events, in which producers put their messages and consumers read the data. The capacity of each list is declared in topic definition. Client-server communication is implemented with a simple, high-performance, language agnostic TCP protocol. This protocol is versioned and maintains backwards compatibility with older version. Kafka clients are available in many programming languages. Kafka implements publish-subscribe protocol, it can deliver messages, persist and duplicate them in order to manage the scalability. All messages written to Kafka are persisted and replicated to peer brokers for fault tolerance. <u>Feature</u> enabling the integration: communication through event broker, scalability and open-source.
Schema registry	Tool that manages schemas using Avro for Kafka events, enabling users to save, edit, or reuse schemas for the required data. With schema version management, data consumers and producers can evolve at different rates. Moreover, data quality is greatly improved through schema validation. The schema registry exposes API in order to manage message schema through http queries. The URL of the current deployed server of schema registry is https://human.holonix.biz/middleware/api/schema-registry . Schema registry UI, i.e. the user interface allowing to edit, manage the schemas through user interface component, is currently deployed and reachable at the following URL: https://human.holonix.biz/middleware/schema-registry-ui/#/

Apache AVRO	<p>Data serialization system allowing data to be self-describing. Schema Registry is integrated with Kafka so that Avro schemas can be passed 'by reference'. Basically, when data are produced, the record and the schema can be passed as data input but this approach does not appear efficient. So, the solution adopted consists of associating a topic to a specific schema: in this way, consumers and producers can know the schema of the record received/sent retrieving the reference of Avro schema (associated to the topic). Overhead is minimized. <u>Feature</u> enabling the integration: interoperability, data serialization.</p>
Kafka proxy rest	<p>Tool is part of Confluent Open Source and Confluent Enterprise distributions. The proxy provides a RESTful interface to a Kafka cluster, making it easier to produce and consume messages. Not all the components are able to produce/consume data directly from Kafka, because Kafka is based on TCP protocol. So KAFKA REST PROXY offers the possibility to consume/produce data through http protocol. It offers also a set of functionalities that allow to subscribe topics, create consumer groups, etc. These functionalities are implemented by Kafka in TCP protocol and are also available through http level thanks to Kafka proxy. <u>Feature</u> enabling the integration: interoperability, HTTP protocol for producing/consuming messages.</p>

5. Integration testing plan

Given the proposed process of IT component integration [3] inside the HUMAN system, integration tests represent a fundamental activity in order to verify that:

- Components can be added to the system without introducing issues in other parts of the system;
- Any interfaces or APIs changed or added by the components have the expected behaviour, and the components are able to interact with the rest of the system.

According to the objective of defining an agile development methodology and to the strategy of continuous integration, all the components (hardware devices or services) will not be available for the integration at the same time, but the system will be gradually built up as new components will be developed and integrated into the system. So, when a new feature is released from integration testing, all the functionalities of that feature will be tested. A testing strategy has to be defined, together with the design of a plan for the HUMAN integration tests. Integration testing represents the last stage of system testing, leading on from unit testing. Integration testing can be executed adopting different approaches. The bottom-up approach described within the previous sections will be followed in the order to define the tests, whose specifications and design are in appendix. Test implementation will be reported within the deliverable D6.2. The integration tests will be implemented by Holonix and their scope is to ensure the right behaviour of the whole system.

Basically, there are two major ways of carrying out an integration test, i.e. the bottom-up method and the top-down method. Bottom-up integration testing begins with unit testing followed by tests of progressively higher-level combinations of units, called modules or builds. In top-down integration testing, the highest-level modules are tested first and progressively lower-level modules are tested after that. Bottom-up testing is usually performed first. Given this scenario, the integration testing plan of HUMAN has been organized in 4 different phases, starting from the lower-level modules and progressively going up to test the higher-level ones. Each phase corresponds to a specific level within the tree presented in Figure 7)

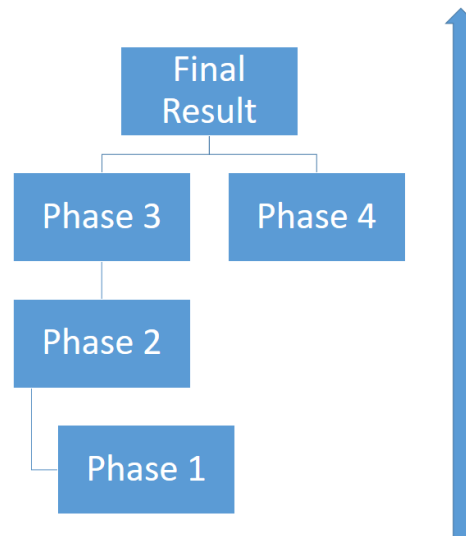


Figure 7: Bottom-up integration testing approach

This approach is further detailed within the next sections. Instead, a proposal of the testing procedures is reported in Appendix C.

5.1 TESTING PLAN OVERVIEW

This section provides the plan for the testing activities to be executed in order to check the success of the integration phase, whereas the execution and the analysis of the tests will be reported in detail in deliverable D6.2. However, some of these proposed tests - the ones necessary to make a preliminary working assessment of the integration solution previously introduced (communication brokers and iModels APIs) - have already been executed or started.

In the following table, the various phases of the testing plan are characterized in terms of objective, involved component and status.

Table 14: Integration testing plan

ID	Phase name	tested integration	Involved components and partners	Status
P1	MQTT interface testing	Collection of data from and pushing to the sensing layer	Wearables (Holonix) Exo (SSSA/IUVO) Sensing Layer (Holonix)	Passed
P2	Integration of sensing layer	Sensing layer shares the data through the Kafka event broker	Sensing Layer (Holonix) Event broker (Holonix)	On-going

	with Kafka interface	in order to make them available to the other components		
P3	Kafka interface integration with services	Testing the integration between Kafka event broker and the other HUMAN components	Models (Holonix) Sensing layer (Holonix) Intervention manager (Supsi) KIT (UCL) EXOs (IUVO) WOS (LMS)	On-going
P4	iModels testing	Integration between API and services	Intervention manager (Supsi) KIT (UCL) EXOs (IUVO) WOS (LMS) SII (Sintef)	Planned

5.2 PHASE 1

The scope of this first phase is to gather sensors data and send them to the sensing layer. Therefore, the integration between wearable and sensing layer based on the use of the developed MQTT broker is taken into account: the collected raw data will be sent through the broker and will be consumed by the sensing layer.

Integration tests are planned as reported in the following table

Table 15: Integration test, phase 1

ID	Test name	Sender (responsible partner)	Receiver (responsible partner)	Test data
P1.1	Devices - Sensing layer testing	Wearables (Holonix),	Sensing Layer (Holonix)	Physiological
P1.2	Devices - Sensing layer testing	Exo (SSSA/IUVO)	Sensing Layer (Holonix)	Exo_Upperlimb

Within this step, an integration test has been planned in order to check that the sensors data are correctly gathered. Note that the used library strictly depends on the technology of the wearable device.

In this step, the integration between Empatica E4 [4] and Smartwatch Huawei Watch 2 is defined, using EmpaLink [5] library that is able to:

- Connect to and manage one or more Empatica E4 devices via Bluetooth Low Energy (BLE)
- Receive real time raw data from the connected devices, such as Galvanic Skin Response (GSR), Blood Volume Pulse (BVP), and accelerometers
- Receive computed data derived from raw data, such as inter beat intervals (IBI)

5.3 PHASE 2

The second phase provides the integration test between the sensing layer and Kafka, after aggregating and enhancing the data through sensing layer. Sensing layer has to share the data through the middleware in order to make them available to the other IT components.

The current phase can be performed only after executing and completing the first ones. In particular, it consists of a “Postman test” that will be executed in order to obtain data and evaluate the expected behaviour through Kafka.

Note that Postman is a user interface platform that allows the developers to build API HTTP requests and test APIs. The produced scripts for test can be exported in JSON format and exchanged between developers.

Holonix is the only project partner involved in testing phase 2.

5.4 PHASE 3

The third testing phase, that is more complex than the previous ones, focuses on the integration between Kafka and the rest of HUMAN components. In particular, the integration of KAFKA with two categories of components is taken into account:

1. Core modules
 - 1.1. Models
 - 1.2. Sensing layer (see the previous phase in order to get more information about the integration of this component)
 - 1.3. Intervention manager
2. Services
 - 2.1. KIT
 - 2.2. EXOs

2.3. WOS

2.4. KSN

2.5. SII

Please refer to the deliverable D1.4 in order to get more details about the specific components.

The next sub-sections contain the testing plan concerning the integration between the previous listed components and Kafka. The partners involved within this phase are reported in the following table.

Table 16: KAFKA enabled integrations to be tested

Sender (responsible partner)	Receiver (responsible partner)	Test data
Sensing Layer (Holonix)	AI Models (SUPSI)	Physiological
KIT (UCL)	Middleware (Holonix)	Jobs, Task, etc.
Intervention manager (SUPSI)	SII (SINTEF), Sensing Layer (Holonix), KIT (UCL), EXO (IUVO), WOS (LMS)	Intervention

5.4.1 CORE MODULES – KAFKA INTEGRATION

These planned integration tests have to be executed within the core module of HUMAN system.

The different steps are:

- From Kafka to models, about their integration.
- Between models and Kafka, in order to ensure that the models are able to consume and publish data on Kafka.
- Between Kafka and Intervention manager, aiming at verifying the expected behaviour of the intervention manager-Kafka integration.

5.4.2 SERVICES – KAFKA INTEGRATION

This integration step will aim at checking that the services are able to produce and consume data to/from Kafka.

5.5 PHASE 4

The last phase of integration testing is focused on the APIs used by the services: for each API, a procedure for testing its availability and functionalities will be developed and executed.

Table 17: integration testing plan: phase 4

Sender (responsible partner)	Receiver (responsible partner)	Test data	HTTPS Query
Persistency layer (Holonix)	Exo (IUVO)	Worker	getWorkers
		Factory	getAnthropometry
			getExoskeletonID
			getExoskeletonSettings
Persistency layer (Holonix)	KIT (UCL)	Worker	LoginUser
		Task	LogoffUser
			DoesUserNeedToReviewJobInfo
			GetJobDifficultPoints
			GetJobTargets
			GetJobDescription
Persistency layer (Holonix)	WOS (LMS)	Worker	getWorkers
		Task	getLTIPrompts
		Event	getAssessments
			getAssessmentResults
			getTask
			getAsset
			getAssetList
Persistency layer (Holonix)	SII (SINTEF)	Task	putAnalysisResults
		Event	getJobs
			getJobSchedule
			getTaskSchedule
			getAllTaskScheduled
			getNet
			getEvents

getErrors
getIssues

Partners and components involved within this testing phase, as for the integration of the preliminary version of the APIs, are listed within the following preliminary table. The table will be updated when new APIs will be released to implement new queries, as requested by the refined versions of the HUMAN components.

6. Bug reporting

Redmine [6] has been chosen in order to report the bug detected in the HUMAN platform, during the integration testing phases.

Redmine is a free, open source, web-based project management and issue tracking tool. It allows users to manage multiple projects and associated subprojects. It provides tools for project wikis and forums, time tracking, and flexible, role-based access control. It also integrates various version control systems and includes a repository browser and viewer.

During the testing and validation phases, it will be offered to the end users to report bugs detected in their specific installations. The testing and validation activities conducted in each industrial scenario will be managed as projects, where activities are testing activities and it is possible to associate issues detected during the testing.

6.1 REDMINE OVERVIEW

After logging in, the user can select a project (among the accessible) in the upper right section and can get a page similar to the one shown in Figure 8.

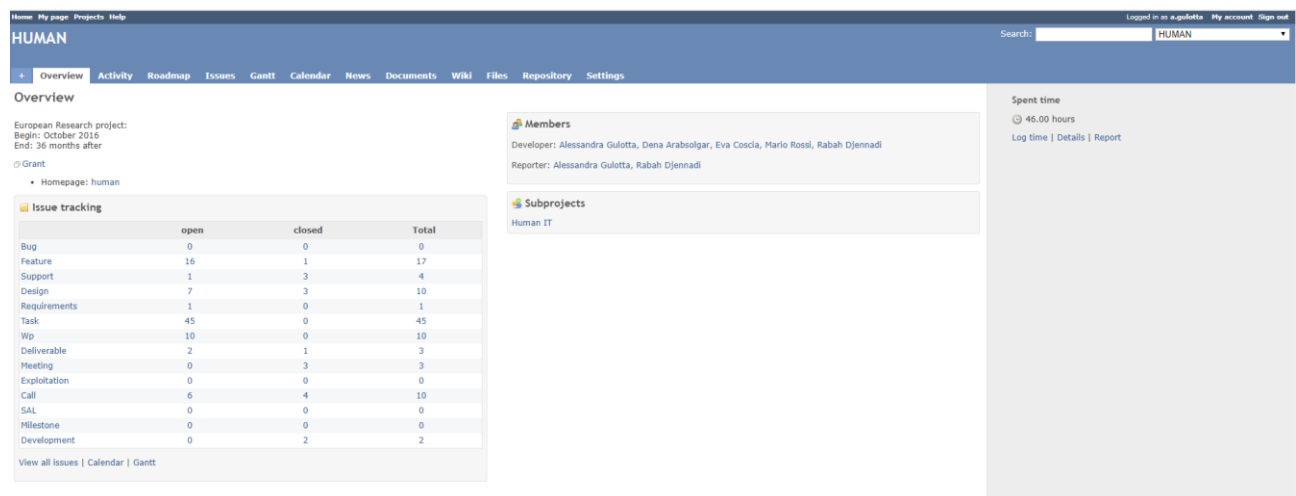


Figure 8: Redmine form

The Project overview can provide to the logged user the main information about the whole project. On left upper side in the “Issue tracking” area, it is possible to get a preview about how many tasks are open and closed for each tracker specified for the project. The “Members” area allows the management of the users and their access privileges. In the “Latest news” area, it is possible to see all the latest news for the particular project.

By clicking on the individual items, such as "issues", it is possible to visualize the reported "issues", together with their current status, their priority, the assigned user, etc. Going into the details of the

"issue", the user can access additional information, such as from description or file attachments useful for solving the problem.

To insert a new issue, the user have to click on the “+” button at the top left and fill the fields necessary for reporting, as shown in the screen below (Figure 9).

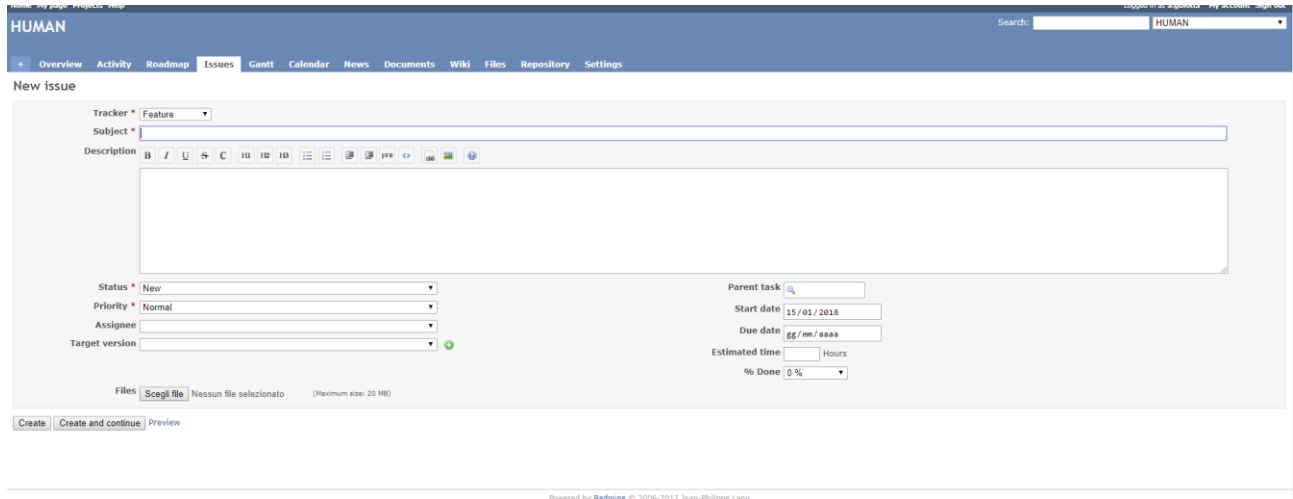


Figure 9: Screen of the “Issue” tab screen of redmine.

An issue could be tracked as a bug, functionality that is object of interest for the HUMAN project.

This is the URL to the current deployed solution, called “Human IT” is: <https://redmine.holonix.biz/projects/human-it>

Given the “bug project” Human IT, a subproject is proposed for the bug reporting of each end-user case (Airbus, COMAU and ROYO), allowing project partners to track their testing activity and report problems, errors and unexpected behaviours of the system. IT partners will receive, notifications of reported issues from Redmine and will be authorised to manage them.

7. Conclusions

Task 6.1 worked on the definition of the integration principles and guidelines to be followed by the development activities in WP2-WP5 to ensure that the final results of the specific WPs will be easily integrated.

These integration principles have indeed guided also the design of the HUMAN architecture presented in D1.4, that leverages on the adoption of communication solutions and data interoperability principles to avoid implementing one-to-one integration solutions and thus improving the scalability and maintainability of the HUMAN system.

The expected readers of D6.1 are both the IT partners that are in charge of developing the technical results of the current project, and all the other developers that will provide new hardware devices or new services to be integrated in the HUMAN platform.

In addition to the presentation of the integration principles, the document presents a plan for the execution of integration testing. Some of the planned tests have already been executed to check the readiness of the interfaces to which existing as well as new HUMAN components have to integrate.

The other phases will be completed as soon as the HUMAN components will be ready in their first version, thus to conclude the first integration by M20. After M20, the continuous integration methodology illustrated in Section 5 of this deliverable will be put in place in Task T6.4 (Continuous Improvement), to integrate new versions of existing components as well as new components, thus potentially extending even after the end of the project.

Appendixes

Appendix A. Current access points and URLs

- Middleware UI address: <http://ns3370643.ip-37-187-92.eu:22006/>
- Kafka broker address: <tcp://ns3370643.ip-37-187-92.eu:22007>
- MQTT broker: address <tcp://mqtt1.holonix.biz> ; port 8883
- APIs address: <https://human.demo.holonix.biz>
- Schema registry deployment: <https://human.holonix.biz/middleware/api/schema-registry>
- Schema registry UI deployment: <https://human.holonix.biz/middleware/schema-registry-ui/#/>
- Gitlab HUMAN code repository <https://gitlab.com/humanufacturing/>
- Redmine deployment: <https://redmine.holonix.biz/projects/human-it>

Appendix B. Current HUMAN repositories on Gitlab

- Repository for the Kafka clients: <https://gitlab.com/humanufacturing/messaging/1710-message-client-kafka>
- Repository for message schema: <https://gitlab.com/humanufacturing/messaging/schemas>
- Repository for Human service: <https://gitlab.com/humanufacturing/integration/session-service>
- Repository for the Human Core: <https://gitlab.com/humanufacturing/integration/human-core>
- Repository for the middleware: <https://gitlab.com/humanufacturing/integration/middleware>
- Repository for the data model implementation: <https://gitlab.com/humanufacturing/integration/data-models>
- Repository for MQTT broker: <https://gitlab.com/humanufacturing/integration/mosquitto-docker>
- Repository for guidelines: <https://gitlab.com/humanufacturing/integration/guidelines>

Appendix C. First proposal of testing procedures

A first proposal of testing procedures for integration is reported. It will be updated, modified and implemented (Task 6.2) according to effective integration aspects. Each test is identified with an alphanumeric code (ID), composed of more substrings: a suffix indicating the integration testing, a substring indicating the object or the domain of the testing and an incremental number.

For clarification, please consider the following example. “ITAPITM01” is composed by:

1. IT=Integration Test
3. APITM= Task model
4. 01= Incremental number

APPLICATION INTEGRATION TEST

KAFKA

Legend: IT (Integration Test), K (Kafka) and incremental number compose the ID

Table 18: SESSION/ JOB_REQUEST/ JOB_RESPONSE/ TASK

Integration test Id	Input description	Condition	Input format	Expected behaviour
ITK01	Session	(Header.Type = (Type) Payload)	AVRO	System produces an event coherent with the request through JOB_REQUEST topic, JOB_RESPONSE topic, TASK topic
ITK02	Other	Header.Type != (Type) Payload	AVRO	No Event

Table 19: PHYSIOLOGICAL/STRESS/INTERVENTION

Integration test Id	Input description	Condition	Input format	Expected behaviour
ITK04	Physiological	(Header.Type = (Type) Payload)	AVRO	- System produces an event through STRESS topic - System produces an event through INTERVENTION topic
ITK05	Other	(Header.Type != (Type) Payload)	AVRO	No Event

MQTT

Legend: IT (Integration Test), MQTT (MQTT) and incremental number compose the ID

Table 20: PHYSIOLOGICAL DATA

Integration test Id	Input description	Condition	Input format	Expected behaviour
ITMQTT04	Physiological	Device ID is present	Json	- System produces an event through STRESS Kafka's topic - System produces an event through INTERVENTION Kafka's topic
ITMQTT05	Physiological	device ID is missed		No event through Kafka

DATA INTEGRATION TEST

TASK MODEL APIS

Legend: IT (Integration Test), API (API), TM (Task model) and incremental number compose the ID

Table 21: GET JOBS

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM01	GET	Void		We expect to get a list of jobs

Table 22: GET JOBSCHEDULE

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM02	GET	Start: timestamp End: timestamp	Start < End	We expect to get s list of job instance scheduled in a certain timeframe.
ITAPITM03	GET	Start: timestamp End: timestamp	Start > End	We except to get an error exception message: "the start date is bigger than End"

Table 23: GET TASKSCHEDULE

Integration test Id	HTTP verb	Input Description	Condition	Expected behaviour
ITAPITM04	GET	JobSchedule. JobScheduleId:String	Existing JobScheduleId	We expect to get the tasks scheduled for a specific job instance.
ITAPITM05	GET	JobSchedule. JobScheduleId:String	JobScheduleId is null Or empty	We expect the error exception message containing “The parameter values are null or empty, please insert correct parameters”
ITAPITM06	GET	JobSchedule. JobScheduleId:String	Not Existing JobScheduleId	We expect the warning message containing “There is no results corresponding to parameter values within the database”

Table 24: GET ALL TASK SCHEDULED

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM07	GET	TaskSchedule.Start: timestamp End: timestamp	Start < End	We expect to list of job instance scheduled in a certain timeframe.
ITAPITM08	GET	TaskSchedule.Start: timestamp TaskSchedule.End: timestamp	Start > End	We except to get an error exception message: “the start date is bigger than End”

Table 25: GET NET

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
---------------------	-----------	-------------------	-----------	--------------------

ITAPITM10	GET	Job. JobId:String	JobId is null Or empty	We expect the error exception message containing "The parameter values are null or empty, please insert correct parameters"
ITAPITM11	GET	Job. JobId:String	Not Existing JobId	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 26: GET TASK

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM12	GET	Task.taskId:String	Existing taskId	We expect to get the information about a task.
ITAPITM13	GET	Task. taskId:String	taskId is null Or empty	We expect the error exception message containing "The parameter values are null or empty, please insert correct parameters"
ITAPITM14	GET	Task. taskId:String	Not Existing taskId	We expect the warning message containing "There is no results corresponding to parameter

				values within the database"
--	--	--	--	-----------------------------

Table 27: GET JOB

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM15	GET	Job. JobId:String	Existing JobId	We expect to get the information about a job.
ITAPITM16	GET	Job. JobId:String	JobId is null Or empty	We expect the error exception message containing "The parameter values are null or empty, please insert correct parameters"
ITAPITM17	GET	Job. JobId:String	Not Existing JobId	We expect the warning message containing: "There is no results corresponding to parameter values within the database"

Table 28: GET WORKING TASK

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM18	GET	Worker.WorkerId:String	Existing WorkerId	We expect to get the task on which the specified worker is currently working.
ITAPITM19	GET	Worker.WorkerId:String	WorkerId is null Or empty	We expect the error exception message containing "The parameter values are null or empty, please insert correct parameters"
ITAPITM20	GET	Worker.WorkerId:String	Not Existing WorkerID	We expect the warning message containing: "There is no results corresponding to parameter values within the database"

Table 29: ADVANCE IN JOB

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM21	GET	Worker.WorkerId: String Job.JobId: String	Existing WorkerId and Existing JobId	We expect to publish to the system that the worker advanced a step in the job. It returns the new task to accomplish.
ITAPITM22	GET	Worker.WorkerId: String Job.JobId: String	WorkerId is null Or empty =0 Or JobId is null or empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPITM23	GET	Worker.WorkerId: String Job.JobId: String	Not Existing WorkerID Or not Existing Job.JobId: String	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 30: NEW JOB

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPITM24	GET	Worker.WorkerId: String Job.JobId: String	Existing WorkerId and Existing JobId	We expect to publish to the system that the worker started a new job. Returns the first task of the new job.
ITAPITM25	GET	Worker.WorkerId: String Job.JobId: String	JobId is null Or empty Or WorkerId is null Or empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPITM26	GET	Worker.WorkerId: String Job.JobId: String	Not Existing WorkerID Or not Existing Job.JobId: String	We expect the warning message containing "There is no results corresponding to parameter values within the database"

EVENT, INTERVENTIONS AND FACTORY MODEL

Legend: IT (Integration Test), API (API), FM (Factory model) and incremental number compose the ID

Table 31: GET ASSET

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIFM01	GET	Asset.AssetId: String	Existing AssetId	We expect to retrieve information such as description status, 3D and positioning information regarding an asset. An asset can be a tool, a resource, a workstation.
ITAPIFM02	GET	Asset.AssetId:String	Asset.AssetId: String is null Or empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIFM03	GET	Asset.AssetId:String	Not Existing Asset.AssetId	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 32: get Calibration

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIFM04	GET	DeviceID: UUID WorkerID: UUID	Existing DeviceID And Existing WorkerID	We expect to retrieve the settings / calibration for the specified device and specified worker
ITAPIFM05	GET	DeviceID: UUID WorkerID: UUID	DeviceID is null or empty Or WorkerID is null or empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"

ITAPIFM06	GET	DeviceID: UUID WorkerID: UUID	Not Existing WorkerID Or not Existing DeviceID	We expect the warning message containing "There is no results corresponding to parameter values within the database"
-----------	-----	----------------------------------	---	--

Table 33: GET EVENTS

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIFM07	GET	Start: timestamp End: timestamp Source: String JobId: UUID	Existing JobId And Start < End	We expect to retrieve the events for a certain timeframe. Matching a certain source criteria and job criteria
ITAPIFM08	GET	Start: timestamp End: timestamp Source: String JobId: UUID	(JobId is null Or is empty) And Start < End	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIFM09	GET	Start: timestamp End: timestamp Source: String JobId: UUID	Not Existing JobId And Start < End	We expect the warning message containing "There is no results corresponding to parameter values within the database"
ITAPIFM10		Start: timestamp End: timestamp Source: String JobId: UUID	Start > End	We except to get an error exception message: "the start date is bigger than End"

Table 34: getErrors

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIFM11	GET	Start: timestamp End: timestamp JobId: UUID	Existing JobId And Start < End	We expect to retrieve the errors recorded for a certain timeframe.
ITAPIFM12	GET	Start: timestamp End: timestamp JobId: UUID	(JobId is null Or is empty) And	We expect the error message containing "The parameter values are null or

			Start < End	empty, please insert correct parameters"
ITAPIFM13	GET	Start: timestamp End: timestamp JobId: UUID	Not Existing JobId And Start < End	We expect the warning message containing "There is no results corresponding to parameter values within the database"
ITAPIFM14		Start: timestamp End: timestamp JobId: UUID	Start > End	We except to get an error exception message: "the start date is bigger than End"

Table 35: GET ISSUES

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIFM15	GET	Start: timestamp End: timestamp JobId: UUID	Existing JobId And Start < End	We expect to retrieve the reported issues recorded for a certain timeframe.
ITAPIFM16	GET	Start: timestamp End: timestamp JobId: UUID	(JobId is null Or is empty) And Start < End	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIFM17	GET	Start: timestamp End: timestamp JobId: UUID	Not Existing JobId And Start < End	We expect the warning message containing "There is no results corresponding to parameter values within the database"
ITAPIFM18		Start: timestamp End: timestamp JobId: UUID	Start > End	We except to get an error exception message: "the start date is bigger than End"

Table 36: GET INTERVENTIONS DETAILS

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIFM19	GET	Type: String	Existing Type	We expect to retrieve information about the

				<p>intervention prompts. For the LT, which have to be assessed by the engineer, The prompts should include but not be limited to: timestamp of prompt, area affected, task affected, trigger reason, priority / importance.</p> <p>For the STI it can include the cause, the affected worker and the effective command string for the device</p>
ITAPIFM19	GET	Type: String	Type is null Or is empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIFM20	GET	Type: String	Not Existing Type	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 37: GET INTERVENTION DETAILS

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIFM21	GET	InterventionID: UUID	Existing InterventionID	We expect to retrieve the intervention detail for the specified intervention.

ITAPIFM22	GET	InterventionID: UUID	InterventionID is null Or is empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIFM23	GET	InterventionID: UUID	Not Existing InterventionID	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 38: GET worker DETAILS

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIWM01	GET	WorkerId: UUID	Existing WorkerId	We expect to retrieve the list of anthropometric measures of the specified worker.
ITAPIWM02	GET	WorkerId: UUID	WorkerId is null Or is empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIWM03	GET	WorkerId: UUID	Not Existing WorkerId	We expect the warning message containing "There is no results corresponding to parameter values within the database"

WORKER MODEL

Legend: IT (Integration Test), API (API), WM (Worker model) and incremental number compose the ID

Table 39: GET WORKER

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIWM04	GET	WorkerId: UUID	Existing WorkerId	We expect to retrieve the details about the specified worker.
ITAPIWM05	GET	WorkerId: UUID	WorkerId is null Or is empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIWM06	GET	WorkerId: UUID	Not Existing WorkerId	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 40: START SHIFT

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIWM07	GET	WorkerId: UUID JobId: UUID WorkstationId: UUID	Existing WorkerId <u>And</u> existing JobId And existing WorkstationId	We expect to publish the information in the system that Worker "WorkerId" started Job "JobId" on Workstation "WorkstationID" (aka: SessionStart)
ITAPIWM08	GET	WorkerId: UUID JobId: UUID WorkstationId: UUID	WorkerId is null Or is empty JobId is null Or is empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"

			WorkstationId is null Or is empty	
ITAPIWM09	GET	WorkerId: UUID JobId: UUID WorkstationId: UUID	Not Existing WorkerID Or Not Existing JobId Or Not Existing WorkstationId	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 41: STOP SHIFT

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIWM10	GET	WorkerId: UUID WorkstationId: UUID	Existing WorkerId And Existing WorkstationId	We expect to publish the information in the system that Worker "WorkerId" ended his working shift (aka: SessionEnd)
ITAPIWM11	GET	WorkerId: UUID WorkstationId: UUID	WorkerId is null Or empty Or WorkstationId is null Or empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIWM12	GET	WorkerId: UUID WorkstationId: UUID	Not Existing WorkerID Or not Existing WorkstationId	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Table 42: UPDATE SHIFT

Integration test Id	HTTP verb	Input description	Condition	Expected behaviour
ITAPIWM13	POST	WorkerId: UUID newStatus: String	Existing WorkerId And	We expect to receive message containing "The system with information about

			newStatus String is not (null Or empty)	the worker has been updated"
ITAPIWM14	POST	WorkerId: UUID newStatus: String	WorkerId is null Or is empty	We expect the error message containing "The parameter values are null or empty, please insert correct parameters"
ITAPIWM15	POST	WorkerId: UUID newStatus: String	Not Existing WorkerID	We expect the warning message containing "There is no results corresponding to parameter values within the database"

Appendix D. References

- [1] “Kafka REST Proxy,” [Online]. Available: <https://docs.confluent.io/current/kafka-rest/docs/index.html>. [Accessed 15 01 2017].
- [2] “Postman Test scripts,” [Online]. Available: https://www.getpostman.com/docs/postman/scripts/test_scripts. [Accessed 15 01 2017].
- [3] M. A. a. C. U. e. Ould, Testing in software development., Cambridge University Press, 1986.
- [4] “Empatica E4,” [Online]. Available: <https://www.empatica.com/research/e4/>. [Accessed 15 01 2017].
- [5] “Empatica dev,” [Online]. Available: <http://developer.empatica.com/>. [Accessed 15 1 2017].
- [6] “Redmine,” [Online]. Available: <https://www.redmine.org/>. [Accessed 15 01 2018].
- [7] R. a. I. A. K. Singh, An Approach For Integration Testing In Online Retail Applications., arXiv preprint arXiv:1207.2718, 2012.
- [8] “Zookeeper,” [Online]. Available: <https://zookeeper.apache.org/>. [Accessed 21 12 2017].
- [9] “Schema Registry UI,” [Online]. Available: <https://github.com/Landoop/schema-registry-ui>. [Accessed 21 12 2017].
- [10] “Schema Registry,” [Online]. Available: <https://docs.confluent.io/current/schema-registry/docs/index.html>. [Accessed 21 12 2017].
- [11] “LabView,” [Online]. Available: <http://www.ni.com/it-it/shop/labview.html>. [Accessed 15 01 2017].
- [12] “Kafka,” [Online]. Available: <https://kafka.apache.org/>. [Accessed 21 12 2017].
- [13] “Javadoc,” [Online]. Available: <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>. [Accessed 15 1 2017].
- [14] [Online]. Available: <https://spring.io/guides/gs/service-registration-and-discovery/>. [Accessed 15 1 2017].
- [15] [Online]. Available: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>. [Accessed 7 2 2018].