18 SETTEMBRE 2018

D6.3.1 NESTORE Platform Shared components & Architecture

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

www.nestore-coach.eu

DELIVERABLE ID:	**WP6/D6.3.1/TASK NUMBER 6.3**
DELIVERABLE TITLE:	NESTORE Platform Shared components & Architecture
RESPONSIBLE PARTNER:	ROPARDO
CONTRIBUTORS:	Ciprian Candea, Marius Staicu, Gabriela Candea, Claudiu Zgripcea (ROPARDO) , Silvia Orte (Eurecat), Isabelle Kniestedt (TUD), Daniele Segato (NEOS), Petia Radeva (UB), Antonino Crivello, Filippo Palumbo, Loredana Pillitteri, Vittorio Miori (CNR-ISTI), Giovanna Rizzo (CNR), Christina Röcke (UZH)
NATURE	Report
DISSEMINATION LEVEL:	CO
FILE:	NESTORE Platform Shared components & Architecture
REVISION:	V8
DUE DATE OF DELIVERABLE:	2018.08.31
ACTUAL SUBMISSION DATE:	2018.09.20
CALL	European Union's Horizon 2020 Grant agreement: No 769643
TOPIC	SC1-PM-15-2017 Personalized coaching for well-being and care of people as they age

Document History

REVISION	**DATE**	**MODIFICATION**	**AUTHOR**
0.1	2018.07.23	Document Creation, initial TOC	Gabriela Candea (ROPARDO)
0.2	2018.07.30	Content written	Ciprian Candea, Marius Staicu, Claudiu Zgripcea (ROPARDO)
0.3	2018.08.14	Content written	Ciprian Candea, Marius Staicu, Claudiu Zgripcea (ROPARDO), aniele Segato (NEOS), Petia Radeva (UB)
0.4	2018.09.02	Content written	Ciprian Candea, Marius Staicu, Claudiu Zgripcea (ROPARDO)
0.5	2018.09.14	Content written	Silvia Orte (Eurecat), Isabelle Kniestedt (TUD), Angelini Leonardo(HESSO), Ciprian Candea (ROPARDO)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

0.6	2018.09.17	Content written and review	Petia Radeva (UB), Antonino Crivello, Filippo Palumbo, Loredana Pillitteri, Vittorio Miori (CNR-ISTI)
0.7	2018.09.18	Review	Ciprian Candea (ROPARDO)
0.8	2018.09.18	Review	Giovanna Rizzo (CNR), Christina Röcke (UZH)

Approvals

DATE	NAME	ORGANIZATION	ROLE
2018.08.14	Marius Staicu, Gabriela Candea	ROPARDO	Reviewer
2018.09.18	Giovanna Rizzo (CNR), Christina Röcke (UZH)	CNR, UZH	Reviewer
2018.09.19	Ciprian Candea	ROPARDO	WP Leader
2018.09.20	Giuseppe Andreoni	POLIMI	Scientific Coordinator

Short Abstract

Present document aims to describe in detail the NESTORE ecosystem architecture and explains its technical specifications on both the implementation criteria and the requirements.

NESTORE adopt an evolutionary architecture approach:

“An evolutionary architecture designs for incremental change in an architecture as a first principle. Evolutionary architectures are appealing because change has historically been difficult to anticipate and expensive to retrofit. If evolutionary change is built into the architecture, change becomes easier and cheaper, allowing changes to development practices, release practices, and overall agility”.

Key Words

NESTORE Architecture, Shared Component, End User, API, Sensors, IOT, Platform, Cloud, Applications

Table of Contents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Table of Figures

1. Executive summary

One of the NESTORE project's main goal is to create an ecosystem to give elderly people a coaching platform / guidance system for optimizing their lifestyle. Since this ecosystem is mostly composed by ICT technologies, such as smartphone applications, social networks and games, the system architecture of this ecosystem acts a strategic role in the entire project.

Present document aims to describe in detail the NESTORE ecosystem architecture and explains its technical specifications on both the implementation criteria and the requirements.

NESTORE adopt an evolutionary architecture approach:

"An evolutionary architecture designs for incremental change in an architecture as a first principle. Evolutionary architectures are appealing because change has historically been difficult to anticipate and expensive to retrofit. If evolutionary change is built into the architecture, change becomes easier and cheaper, allowing changes to development practices, release practices, and overall agility".

The following chapters will describe the Ecosystem from different points of view:

- The ecosystem description in terms of functionalities that each part will provide
- The Architecture of the System and the overall ecosystem requirements, giving a detailed description of each part that compose the architecture itself
- The interactions between the modules and/or the ecosystem.

2. Architecture overview

NESTORE architecture must be developed and integrated the three interconnected and complementary components, which are the Monitoring System, the DSS and the Virtual Coach.

Resulting architecture is supporting NESTORE platform using components that are available on the market – where possible open source modules – closed integrated.

Based on the T6.1 "Platform Requirements" NESTORE architecture is developed accordingly to functional and non-functional requirements addressing but not limited to: communication interface, sub system integration, data security (security and confidentiality of the sensible data).

Architecture is based on the following parameters:

- Quality of technology (efficiency, effectiveness, usefulness, completeness and accuracy),
- Reliability of the system (reliability, security, interoperability and ease of repair and maintenance)
- Use of monitoring (frequency, modus operandi, user profile, context in the platform's use).
- Problem solving (useful alerts, false alarms, emergency calls, quality life improvement, etc.)

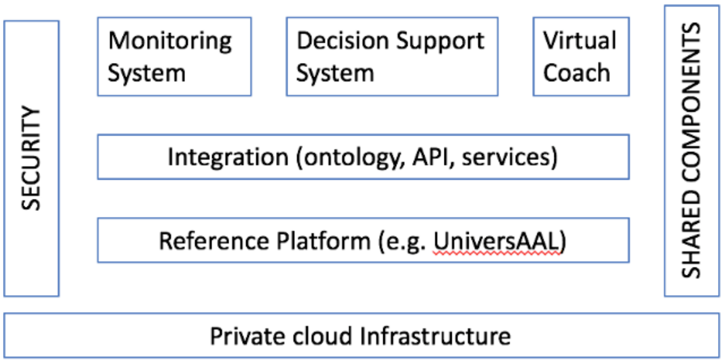

SECURITY
Monitoring System
Decision Support System
Virtual Coach
SHARED COMPONENTS
Integration (ontology, API, services)
Reference Platform (e.g. UniversAAL)
Private cloud Infrastructure

Figure 1 NESTORE Architecture from DoA

Shared modules are represented by Identity Manager, Security Manager and Private Cloud that are used by NESTORE platform and its components. NESTORE Identity Manager Module is necessary for end user identities management across the platform components, APIs, the cloud, mobile, and sensors, regardless of the standards on which they are based.

NESTORE platform expose a wide range of data, which are subject to security and privacy concerns like: User data, sensor data, Actuator access, Service membership.

Each of these areas offer a proper security level which is audited for compliance with existing rules. NESTORE Platform is running the applications using a private cloud.

2.1 NESTORE Platform

In Figure 2 NESTORE Use Cases - Architecture, main components are identified. As can be seen from the picture, the principal components of the NESTORE ecosystem are essentially three:

- Elderly People, which are the End Users of this platform. They will interact with the ecosystem with their smartphone and by using some web applications.
- NESORE cloud, which is the most important part of the ecosystem since it hosts all the back-office part and is in charge to retrieve, elaborate and store the data coming from the platform users.

The End User are using the NESTORE applications on their environment / their house by accessing specific coaching applications (in form of Web, smartphone or wearable application) that are running on different devices (tables, phones, wearable) accessing Internet.

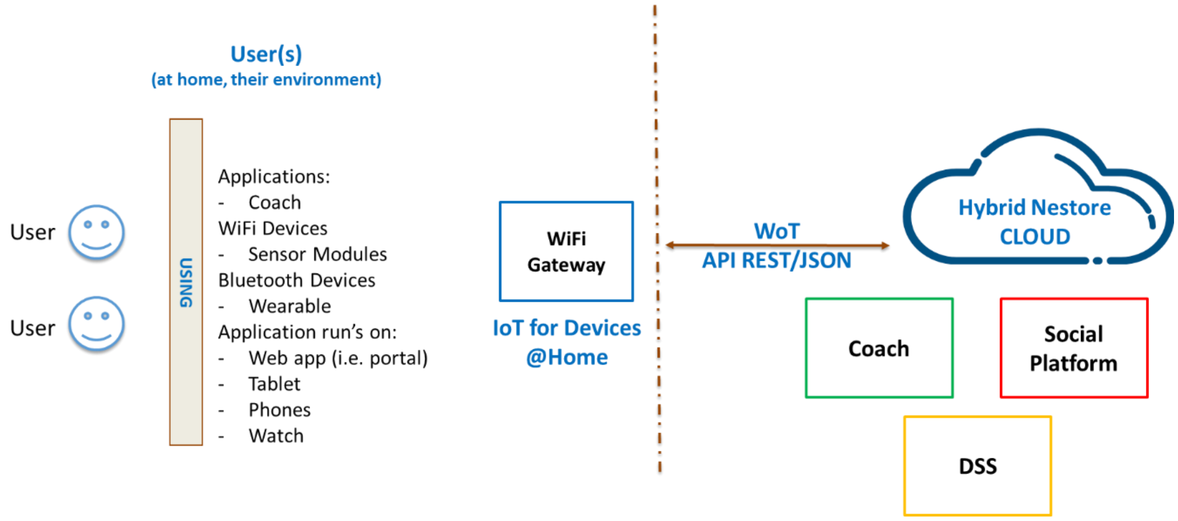


Figure 2 NESTORE Use Cases - Architecture

The environmental sensors are connected the NESTORE platform using WiFi connection and is using the IoT / WoT approach.

The NESTORE Cloud is accommodating the back end needed applications and shared modules.

Entire NESTORE architecture is built around Users that represents real center around whom the ecosystem and the services are conceived, designed and developed.

2.2 System Architecture requirements

According to the definition of NESTORE platform architecture, several and different system requirements are strictly necessary for guaranteeing a proper functioning of every feature composing the system.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

The result expected in satisfying all those requirements is to implement a good and stable system able to provide an enough efficient, scalable and reliable service to its users.

In case of NESTORE cloud will be implemented using a Private Cloud platform, many problems such as network availability and hardware fault-tolerance will be addressed directly from the service providers. Conversely, each block of the architecture will have to considering several aspects in order to assure a constant service delivery such as:

- Mechanism to obtain high service availability, e.g. 99,9%
- A robust software fault-tolerance mechanism, e.g. with more instances of the same software or with specific techniques to recover a faulty software instance without interrupting the service
- Mechanisms to avoid bottlenecks that will result in service interruption/long delays

Each block must satisfy these requirements, since the NESTORE cloud is composed by a lot of modules that are strictly dependent from each other and a fault in one module could cause a performance decay / a service fault in the entire ecosystem.

2.2.1 Flexibility

Flexibility is a key characteristic of a cloud computing systems.

The NESTORE platform will allow its End Users to access the platform from different web-enabled devices (desktops, laptops and smartphones). With such capability, private and public NESTORE data will be accessible to any authorized individual from essentially any place that has Internet connectivity.

Besides this "work from anywhere" characteristic, flexibility is also present in a cloud computing system at a hardware level.

Other similar examples of cloud flexibility properties are: increased collaboration and automatic software updates.

2.2.2 Scalability

Scalability is the ability of a system to adapt its capacity on a rising workload / amount of data.

A system can scale both vertically and horizontally. In the former, the system increases its computational capacity by increasing hardware performances (e.g. CPU clock frequency, memory, etc.) while in the latter, the system replicates its instances to increase its computational capacity.

Since we will use cloud computing, horizontal scaling solution is more feasible and easy to use. This feature is often provided directly by the cloud service provider, so that we can use it easily. Hence, when computing and bandwidth needs get higher than usual, the NESTORE platform will automatically demand more resources for its cloud-based services.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

2.3 Sensors

For NESTORE sensors are refer as:

- by “sensor”, NESTORE refers to the hardware components, which convert quantifiable changes into raw numerical data (e.g. a 3-axis accelerometer converts the physical acceleration into numerical data in [g]);
- by “sensor module”, NESTORE refers to the hardware platform that supports one of many sensors, to monitor/process/record/stream different parameters (e.g. a smart bracelet monitors the physical activity of the wearer by processing the numerical acceleration values of the 3-axis accelerometer into daily activity profiling such as 50 minutes of running, 200 minutes of walking, etc.);
- by “extracted parameter”, NESTORE refers to the already mentioned parameters which are provided by an embedded low-processing of the raw numerical data (e.g. running time, daily energy expenditure, sleep duration, etc.).

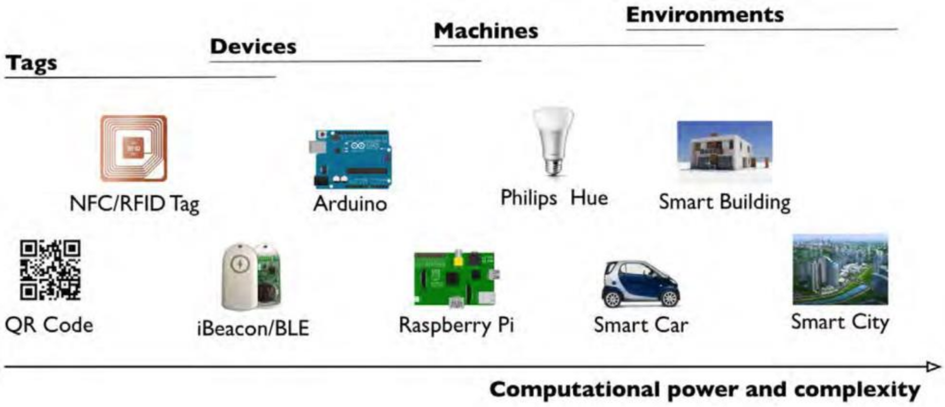


Figure 3 Sensors and Sensors Module Sample (Dominique Guinard, 2016)

The sensors and sensors modules are detailed in terms of architecture and functionalities in the WP3 deliverables. In the following section, the IoT architecture will be defined.

Sensors / sensor module represents the first interactions between End Users and the NESTORE platform. Data acquired by sensors are used by different NESTORE coaching applications providing to End User: reactive feedback and short-term and long-term monitoring. The measured parameters refer two categories: body and environmental parameters.

Reactive feedback – is used to monitor different parameters in instantaneous (i.e. instantaneous heart rate monitoring module) manner and provide feedback to user (i.e. user can check values on smartphone).

Short Term - monitor different parameters over a short period of time (i.e. hours) by collecting important and specific information, which relates to physiological parameters at rest, during intensive physical activities and during recovery periods.

Long Term: monitor different parameters daily, ideally 24/7 collecting an important amount of data providing information on the physical activity lifestyle, used on long term coaching.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 769643

Concerning the interface between sensor modules and the NESTORE platform, it is constituted by wireless communication. For the wearable devices communication is realized between the wearable to smartphone using Bluetooth connection and from smartphone to NESTORE cloud using Internet communication. For environmental sensor module communication is realized between the sensor module and local IoT gateway followed by Internet communication with NESTORE cloud.

2.4 NESTORE applications

NESTORE applications represent the main interfaces that End User will interact with NESTORE Platform. The NESTORE will provide of a series of web and mobile applications that will provide a framework of services to the final users.

The applications, functions and their features will evolve through the whole life cycle of the NESTORE project. This means that the apps can consistently change throughout the entire development of the NESTORE platform trying to adapt the services to the peculiar characteristics of the users following the user-centered design approach.

User Profile: a profile of the user that contains:

- customizable avatar to personalize the private/public user profile inside the NESTORE platform;
- content sharing, i.e. the possibility to collect, receive, store and share content;
- feedback messages to make the user aware about his/her habits and give advice;

Monitoring:

- Nutrition and Food Diary, to collect nutritional information for behavior recognition.
- MyStats, i.e. sensors dedicated app to monitor the user’s physical activity through the sensors embedded inside the mobile (first level of monitoring) and through the set of sensors compatible with the NESTORE ecosystem (i.e. the so-called “NESTORE sensors”, developed by the consortium and other commercial sensors).
- MyHome, i.e. sensors dedicated app to monitor the user’s environmental sensors

Virtual Coach: The coach will guide the user towards the different wellbeing pathways providing tailored interventions based on the HAPA and SOC models

- Chatbot: natural dialogue with the user;
- tangible interfaces for self-reflection and behavior change;
- social and environmental support through the social platform;
- serious games and gamification;

Social Platform

- Share and offer knowledge and services provided by the users to other users;
- Offer services and knowledge of external entities, such as public institutions, enterprises, shops to NESTORE End Users
- End Users can share their personal advancements in the different intervention dimensions, to confront their result with other participants’ results, and will support social collaborative and competitive games

2.4.1 Serious Games

The goal of the serious games, presented under a single 'game suite', is to engage users with the coaching platform over time and to contribute to the maintaining of physical, social, and mental well-being. The NESTORE game suite is developed following an iterative (agile) development method. Functionality and design of the game suite (e.g. game mechanics) will change over time as the design will change to accommodate user feedback and integrate expert knowledge. However, underlying architecture will see fewer changes to ensure consistency with the overarching NESTORE system. The following section focuses on the architecture of the game suite. For an overview of the app's design, we refer to the preliminary version of D5.5.

The NESTORE game suite is developed using the Unity3D game engine, one of the most widely used middleware tools for professional game development. The motivation for using Unity3D within NESTORE is the availability of extensive tools, assets, and libraries (which include support for creating augmented and virtual reality applications), its solid documentation, and its ability to build apps across multiple (mobile) platforms. Game code is programmed in C#. Since Unity is structured around building standalone apps, the NESTORE game suite will be its own application running on the user's device (e.g. phone or tablet) next to other NESTORE apps. One constraint is that any mobile device will need an accelerometer to utilize some of the intended mixed-reality aspects of the game suite, meaning that web/pc is not supported. While support for both Android and IOS is possibly due to Unity3D's cross-platform capabilities, only one operating system can be assured to be fully supported due to development limitations.

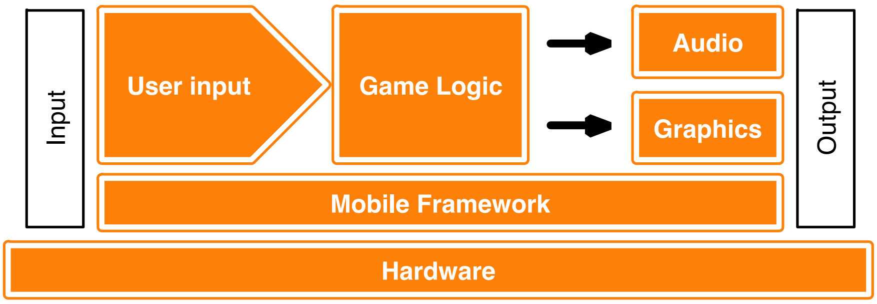


Figure 4 Game architecture on client side

Devices running the NESTORE game suite connect to a server for client-to-client communication and to store game and user data. Data is stored in a game-specific database on the NESTORE server (running Node.js). Data stored includes e.g. login data, personal game data (high scores) and multiplayer game-ID's (identifying instances of multiplayer session connected to a particular player). Similarly, instances of multiplayer games are stored (game states). Most data is stored on the server, while the client device holds the game application.

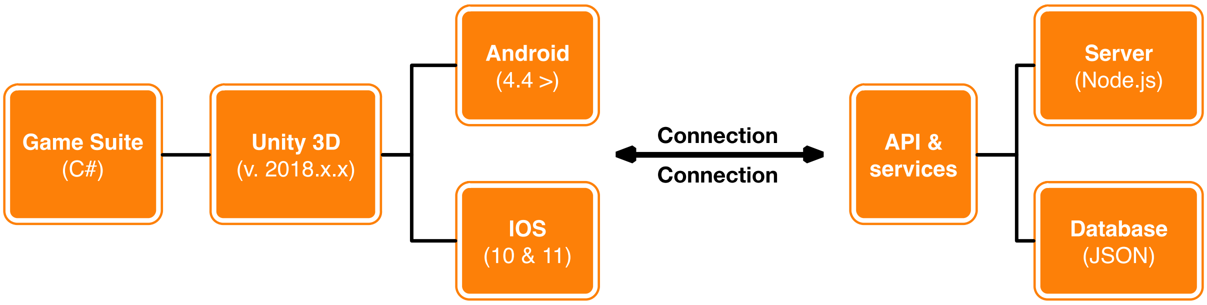


Figure 5 General Architecture

For certain aspects of the gameplay, the game suite will be required to consult additional data on the server, e.g. to check whether users have completed coaching activities since they last logged into the game. For this purpose, the game suite will need to read from those databases. It will, however, not write to any database other than the one dedicated to storing game data. Data in the game database should be readable by other NESTORE components as well (e.g. for the coach to recognize game activities), but no data should be written to the database other than by the game suite itself.

The NESTORE game suite, out of all the NESTORE components, arguably deals with the least sensitive data as it mainly pertains to in-game performance data instead of (identifying) personal data. The most sensitive aspect of the data stored is the user's login ID and password. However, efforts will be made to ensure secure storage and communication of the data from and to the NESTORE game suite. Currently, the following security measures are planned and in the process of being prototyped:

- All communication between server and client will take place over Websockets.
- Socket communication is facilitated by the Socket.IO library on both client and server side.
- Communication is encrypted via AES256 (CBC) encryption. AES is the adopted standard for encryption worldwide, including government institutions.
- Additionally, each message between client and server uses a new, randomly generated initialization vector (IV), transmitted in plaintext, that is required to decrypt the message.
- Sensitive data, such as passwords, are not stored as plaintext but are hashed using SHA256. Designed by the United States National Security Agency, this cryptographic hash function is widely used, e.g. for cryptocurrencies such as Bitcoin. The function provides a good balance between security and computation speed.
- For security reasons, the game server component should use a separate database (e.g. MongoDB, currently using lowdb during prototyping).
- Any initial connection requires authorization against the game user database. Follow-up functionality is only available after such authorization has been granted. Any disconnect requires renewed authorization by the client.

2.4.2 Decision Support System

The Decision Support System (DSS) is a software that can be considered the intelligence of the Nestore System. DSS processes data to take decisions in complex situations due to large number of indicators expected to be measured. The aim of the Decision Support System is to develop the data processing elements needed to provide tailored feedback based on integrated information sources. One of the main objectives is to extract knowledge from data sources to both facilitate the subsequent decision making and adapt the general model to infer valuable information. These data will be continuously mined to extract the required indicators.

A twofold framework is proposed for providing the intelligence to the system. On one side, a short-term analysis devoted to process the data and mine it to extract valuable indicators. This module will provide tailored feedback based on integrated information sources. On the other side, long-term analysis will capture the general trend over a longer period. This module will oversee recognizing the individual behavioral habits and to predict possible declines to propose personalized guidelines. Concretely, it will assess if behavior change has indeed occurred in the user and is not just a temporary fluctuation of their habit. With this aim, trends and patterns will be extracted from daily/weekly indicators gathered in the short-term.

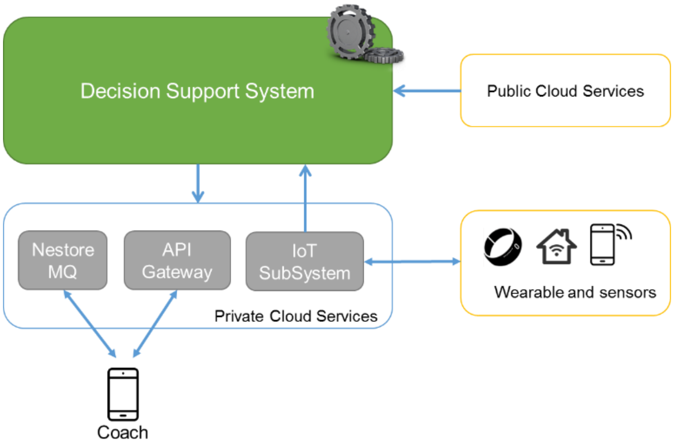


Figure 6 DSS General Architecture

The analysis will provide further level of understanding of the status of the final users, which will help in personalizing all the interaction with the coach. As depicted in Figure 6, the DSS is fed by different Public Cloud Services as the LogMeal API and the information coming from the wearables and sensors through the IOT SubSystem.

The DSS generates two kind of results which are directly consumed by the Coach:

- Reminders: sent to the Coach through the Nestore MQ (i.e. meal time reminders);
- Recommendations: published in form of Rest Services and consumed by the Coach through the API Gateway.

2.4.3 Virtual Coach

The Virtual Coach orchestrate the information flow between the user and the NESTORE Cloud intelligence, i.e., the Decision Support System and the repositories for the static and dynamic user profiles. The virtual coach is composed of cloud components and different user interfaces. The proposed architecture of the Virtual coach is shown in Figure 7 Virtual Coach Architecture. Dashed elements have not been implemented yet

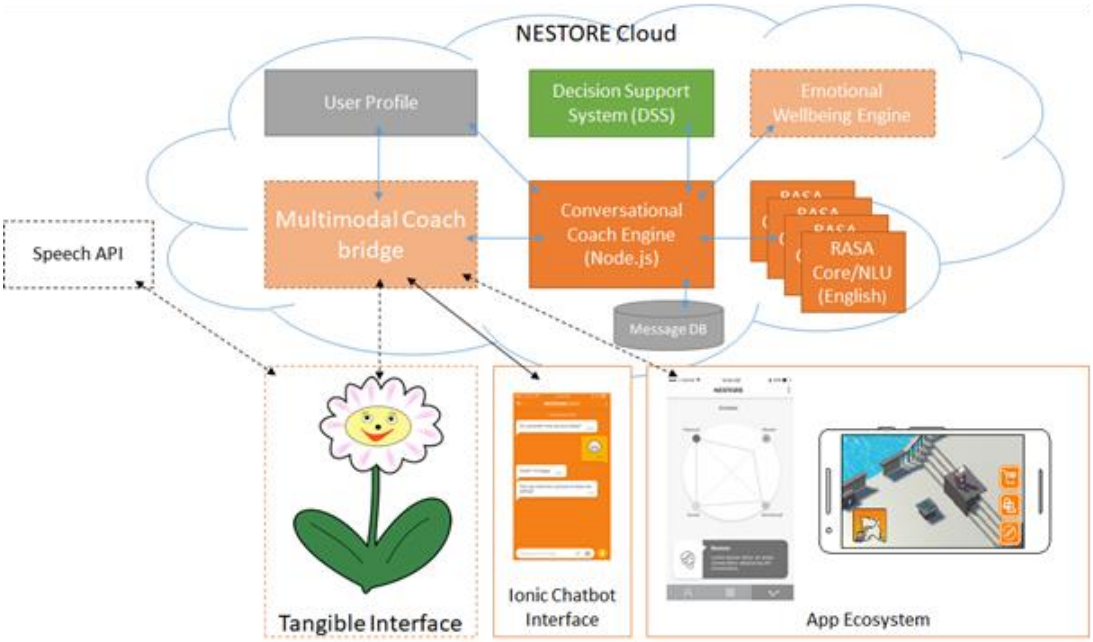


Figure 7 Virtual Coach Architecture. Dashed elements have not been implemented yet

Coaching recommendations and reminders provided by the DSS are treated by the Conversational Coach Engine and dispatched through the Multimodal Coach Bridge to the proper user interface, according to user preferences and context information (such as user position or active apps).

Conversations are managed by the Coach Engine developed in Node.js, which manages the context of the conversation and switch between different scenarios according to DSS inputs (e.g., reminders or recommendations) and user inputs. In particular, user's free-text inputs are treated by 4 instances of the RASA Core/NLU Server, each trained in the different languages supported by the chatbot (English, Spanish, Italian and Dutch). Scripted conversations (chatbot messages and questions and user's multiple-choice answers) are stored in the message DB in each of the 4 supported languages.

Data collected from the chatbot are used to update the static profile of the user (such as user personal data and preferences) through the Nestore MQ bus. Data related to the dynamic profile are sent instead to the DSS for further treatment. In the context of the conversation flow, particular sentences spelled or written by the user are sent to the Emotion Well-Being Engine that analyzes semantically the emotional valence of the phrase.

The output of this module is used by the Conversational Coach Engine to adapt the conversation in empathic manner. In particular, for the tangible interfaces, conversations are also adapted and converted to and from speech through external Speech Recognition and Text-To-Speech API. The architecture of the tangible interface of the coach is still under definition. Please refer to D5.3.1 for further details.

Messages from the coach might also be dispatched in different parts of the app ecosystem, depending on app usage and on the content of the message. The chatbot interface and the app interface are developed in Ionic to ensure cross-platform (Android and iOS compatibility).

Further details on the chatbot architecture and conversation management can be found in D5.2.1, while further details on the chatbot interface design might be found in D5.6.1.

Please note that at current state, dotted elements have not been implemented yet, while a functioning integrated prototype includes already a first version of all the other elements.

2.5 Web portal

The web portal is another direct user interface, representing the main access to the NESTORE system for End Users.

This user interface is designed to reach the NESTORE main target, the older persons, and it will present different graphics, a simple representation of the platform activity through widgets, and a strong capability of customization.

The sign-in system will provide different access to the platform contents, in relation to the type of logged-in user role.

The main functionality of the web portal that will be relevant for the End Users will concern mainly the social aspects of the system and the relation sharing services and knowledge. Each End User will be able to:

- have access to his profile to control his personal data;
- to store and share content, knowledge, post services request / provide;
- to create social groups based on circles and integrated with the most popular Social Networks;
- to have access to specific forum of external services providers experts (i.e. physicians, nutritionists, fitness instructors, psychologist), i.e. public space of discussion among users and Question and Answers (Q&A) sessions with the team of experts,
- manage their GDPR settings

2.5.1. External providers portal

The external provider portal is the user interface where those can publish the services offers and knowledge to NESTORE End Users. The external providers are represented by (but not limited):

- public institutions,
- enterprises,
- shops

The authentication of the stakeholder in the NESTORE web portal will give the possibility to reach part or all the NESTORE End Users to promote initiatives related to the NESTORE aims (events, discounts, special meals and menus, etc.).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

2.4 Data exchange

For NESTORE inter-application communication / integration the RESTFull API over HTTPS where selected as preferred technology. In this decision we evaluate the OData standard conclusion are in the annex "OData Evaluation".

Based on the NESTORE requirements application integration will take place at levels based on classification proposed by (Linthicum, 2000) and is detailed on D 6.2 (NESTORE, D6.2 Evaluation of universAAL solution, 2018) for NESTORE project.

The acronym API comes from Application Programming Interface. An API is a set of functions and procedures that fulfill one or many tasks for being used by other software. It allows to implement the functions and procedures that conform the API in another program without the need of programming them back. As RESTful systems usually communicate with the Hypertext Transfer Protocol (HTTP), a REST API is a library based completely on the HTTP(S) standard. It is used to add functionality to a software somebody already owns safely. The functionality of an API is usually limited by the developer. The Hypertext Transfer Protocol (HTTP) is a stateless application-level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems (RFC 7230.2014).

In a RESTful system, clients and servers negotiate the representations of resources via HTTP. RESTful systems apply the four basic functions of persistent storage, CRUD (Create, Read, Update, Delete), to a set of resources. In terms of the HTTP standard, those actions can be translated to the HTTP methods (also known as verbs): POST, GET, PUT, and DELETE. Other HTTP methods that are also used but not as often as the formers are OPTIONS, HEAD, TRACE, PATCH and CONNECT.

The communication between NESTORE components is made through representations of resources. For NESTORE API, the format of those representations where selected as JSON. JavaScript Object Notation (JSON) is a lightweight data-interchange format. It was derived from the ECMAScript Programming Language Standard (ECMA-404 2013; RFC 7159.2014). JSON structure can be defined as a ordered list of objects (array of objects). These objects are a collection of name/value pairs separated by colons (:). The utilization of JSON instead of other standard format like XML is due to its simplicity. Both XML and JSON are human readable, but JSON does not need closing tags and is easier to read and is less dense.

A guide line for creating new Web Services API is developed and released for having a common description of the resources and it is on the **Annex** – "**REST API Guideline**".

3. Architecture main components

3.1 Cloud services

NESTORE cloud platform is developed as a Private Cloud using Open Source modules. The resulting Cloud could be deployed using an existing cloud service provider if it is required. In this way the entire platform will be centralized, minimizing the inter-operational problems (e.g. network issues) and optimizing the latency in the data exchange among modules. Using a cloud solution manage any aspect related to the hardware (maintenance of servers, fault-tolerance of the hardware, and so on…).

Nowadays, cloud computing is increasingly playing a key-role in the enterprises business, and consequently there are many service providers that offer very large cloud computing solutions, from Software as a Service (SaaS) to Infrastructure as a Service (IaaS).

Today, the main players in the IaaS market are:

- Microsoft Azure (azure, 2018)
- Google Cloud Platform (cloud, 2018)
- Amazon Web Services (aws, 2018)
- Rackspace (rackspace, 2018)

For the NESTORE project the Cloud architecture is accommodated by one of the project partners and the entire solution can be migrated to any IaaS provides.

Each of providers has different peculiarities and services, and in this respect, the choice of which service provider to adopt strictly depends on what kind of services are needed.

3.1.2 NESTORE Cloud architecture

NESTORE Cloud architecture (Figure 8 High Level NESTORE Cloud) address different use case scenarios. On the designed architecture NESTORE is addressing the deployment of all NESTORE services needed for End Users and all needed development tools.

For development tools we refer self-hosted project management tools, source control, documentation, API development tools, etc. . The management tools accommodated in NESTORE Cloud are: Nexus, GitLab, Taiga, NESTORE document management and collaborative server, ApiCurio.

On the other side NESTORE Cloud provide all services needed for the Coaching applications.

During the analyses phase was identified the need to connect the NESTORE private cloud to different Public cloud services like: LogMeal API provided by the University of Barcelona.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

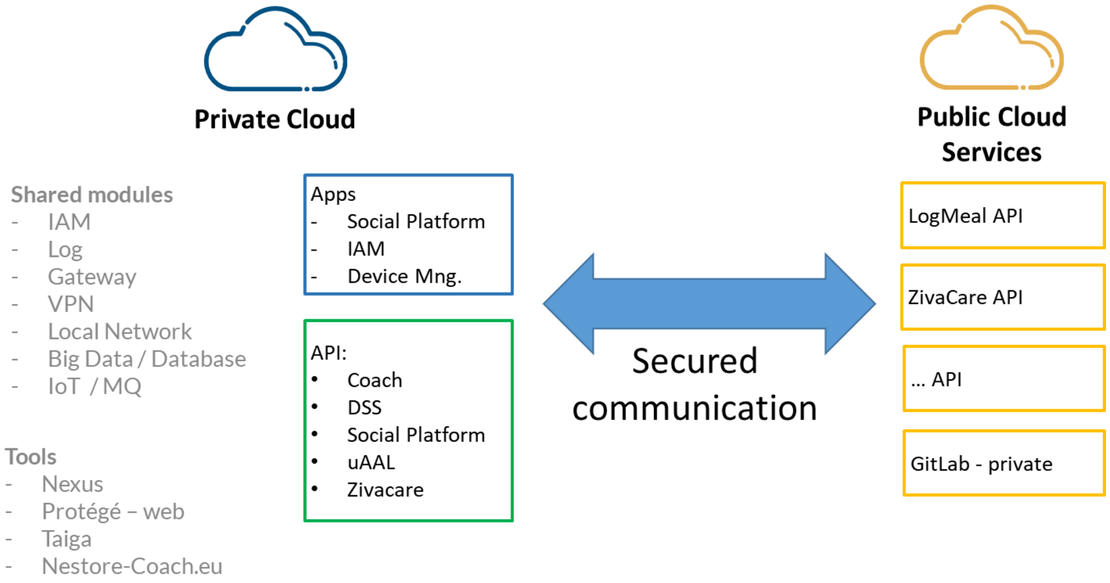


Figure 8 High Level NESTORE Cloud

For the Coaching services the cloud services are accommodating 2 big parts, 1) shared modules 2) dedicated services.

Shared modules are represented by Identity Manager, Security Manager, API Gateway, Virtual Private Network that are used by NESTORE platform and its components.

- Identity Manager Module is used to connect and manage identities across the platform components, APIs, the cloud, mobile, and sensors, regardless of the standards on which they are based.
- Log – log services for audit purposes
- Gateway - API gateway that is the single-entry point for all clients
- Virtual Private Network - secures the private network, using encryption and other security mechanisms to ensure that only authorized users can access the network and that the data cannot be intercepted.
- Databases – database storage needed for different NESTORE components / modules
- MQ - provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact each other at the same time
- IoT – Internet of Things sub-system

Part of the shared module are presented in more details in a dedicated chapter “Shared Components”.

It is important to note that for the modularity of the architecture, a IaaS provider choice does not represent a constraint to the architecture, because each module can be easily adapted to any cloud services.

There are also optional services whose usage is foreseen in the case of a sensible increase in the number of NESTORE users, after the end of the pilot period. One of those is the usage of a caching system for improving the performances of the ecosystem.

Caching could be performed in memory using any standard de-facto such as Redis, that makes it easy to deploy, operate, and scale an in-memory cache in the cloud. The service improves the performance of web applications by allowing you to retrieve information from fast, managed, in-memory caches, instead of relying entirely on slower disk-based databases.

Redis can manage data persistence and availability through a set of different geographic locations on a cloud environment. Using a caching system can guarantee inter-process communication and the evolution of NESTORE towards a microservice architecture.

3.1.2 Cloud technology stack

For the implementation are used the Virtual Machines (Figure 9 Virtual Machines) as containers where different services are accommodated. With this approach an Elastic Infrastructure is developed and preconfigured with virtual server images, storage and network connectivity that may be provisioned by NESTORE using a self-service interface. Monitoring information is provided to inform about resource utilization required for traceable and automation of management tasks.

Hypervisors are virtual machine monitor(VMM) that enables numerous virtual operating systems to simultaneously run on a server system. These virtual machines are also referred as guest machines and they all share the hardware of the physical machine-like memory, processor, storage and other related resources. This improves and enhances the utilization of the underlying resources.

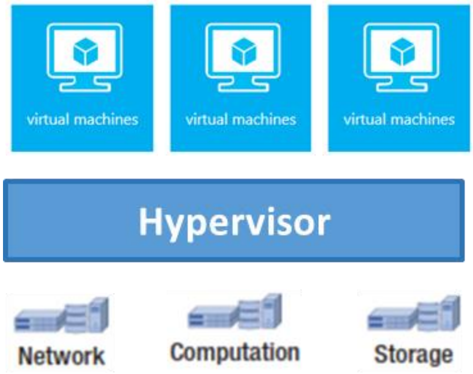


Figure 9 Virtual Machines

The hypervisor isolates the operating systems from the primary host machine. The job of a hypervisor is to cater to the needs of a guest operating system and to manage it efficiently. Each virtual machine is independent and do not interfere with each another although they run on the same host machine. They are no way connected to one another. Even at times one of the virtual machines crashes or faces any issues, the other machines continue to perform normally.

A dedicated Virtual Lan (VLAN) network is deployed using switch subdivided into a set of virtual port addresses; for NESTORE different subnet are provided subdivided into separate domains of local IP addresses and associated ports (for development tools and for NESTORE Coaching applications). Within a VLAN domain, devices can communicate with one another without the use of routing.

For accessing the NESTORE Cloud a virtual private network (VPN) is deployed and configured. VPN produces a virtual address space and encrypts the traffic to become private, by way of how the forwarding tables connect their address points. Technically, this is a virtualization of a network, although it has a much narrower scope and scale than the virtual networks used to stage hypervisor-hosted and containerized workloads. Using VPN, the connection between Private Cloud and Public Clouds can be realized, depending on the required topology.

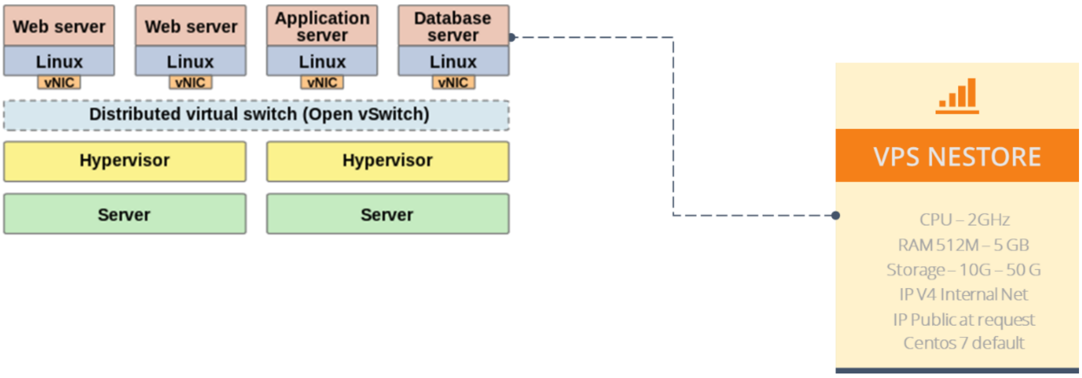


Figure 10 NESTORE specific cloud

Virtual Private Server (VPS) is a virtual server that the user perceives as a dedicated/private server even though it is installed on a physical computer running multiple operating systems.

For deployment of all NESTORE components – shared or dedicated modules – the VPS are used (Figure 10 NESTORE specific cloud). This give to the project all the flexibility needed and latter all the set-up can be exported and deployed on other infrastructure or can be exposed as templates.

For management of the VPS we choose Proxmox Virtual Environment that is an open source server virtualization management solution based on QEMU/KVM and LXC. Easley can manage virtual machines, containers, highly available clusters, storage and networks with an integrated, easy-to-use web interface or via CLI.

3.2 NESTORE MQ

Messaging is a key strategy employed by NESTORE cloud distributed environment. It enables NESTORE applications (i.e. ChatBot, LogIn/Register, etc.) and services (i.e. DSS services) to communicate and cooperate. It also sustains a scalable and resilient solutions. Messaging supports asynchronous operations, enabling NESTORE to decouple a process that consumes a service from the process that implements the service. The NESTORE message bus enables a message to be posted to a queue via HTTP (always HTTPS) and this will ensure a simple and strong security model as well a universal accepted transport protocol represented by HTTP(S).

A message queue receives messages from an application and makes them available to one or more other applications. In the NESTORE architectural scenarios, if application A needs to send updates or commands to applications B and C, then separate message queues can be set up for B and C. A would write separate messages to each queue, and each dependent application would read from its own queue (the message being removed upon being dequeued). Neither B nor C need to be available for A to send updates. Each message queue is persistent, so if an application restarts, it will begin pulling from its queue once it is back online. This helps break dependencies between dependent systems and can provide greater scalability and fault tolerance to applications.

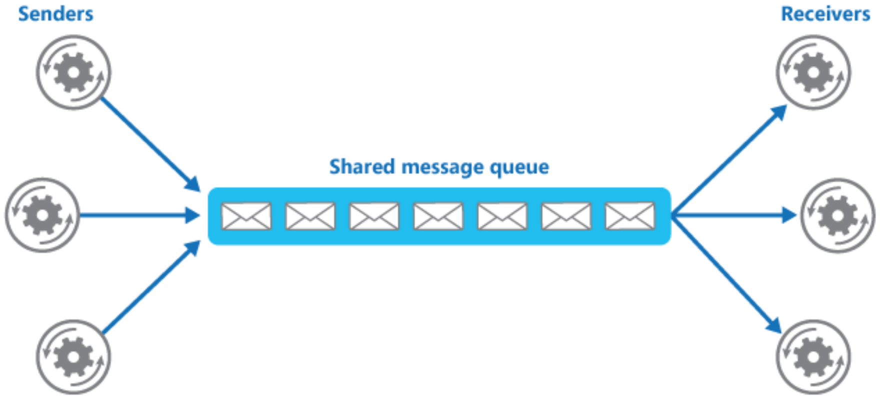


Figure 11 MQ Sender - Receiver

The modern message queue implementations (such as Apache MQ) support having a single message be read by multiple endpoints. Messages become “invisible” to applications that have read them for some period of time before actually being removed. During this time, the message can still be read by other applications. This blurs the line between queues and buses, especially as it pertains to the 1:1 correspondence between queues and destination applications (Figure 11 MQ Sender - Receiver).

Messages carry a payload as well as metadata, in the form of key-value pair properties, describing the payload and giving handling instructions to Service Bus and applications. NESTORE will allow application to send / receive messages encoded as MQTT for the IoT sub-system as well payloads encoded as JSON objects for the inter application communication.

3.2.1 NESTORE MQ integration with uAAL

Integration of the NESTORE MQ with the uAAL is realized by a dedicated architecture component “NESTORE - uAAL Bridge” that is detailed on the next chapter. Next is described the role of NESTORE MQ for the integration.

In NESTORE, the reference scenario is more oriented to the person rather than the environment, enlarging the AAL space and opening a research challenge from the architectural point of view. In such an extended virtual ecosystem, hardware as well as software components can “live” while being able to share their capabilities. In this space, the universAAL platform can be still used to facilitate the sharing of three types of capabilities: Context (data based on shared models), Service (control) and User Interaction (view). Therefore, connecting components to the universAAL platform is equivalent to using the brokerage mechanisms of the uAAL platform in these three areas for interacting with other components in the system. Such connectors, together with the application logic behind the connected component, are called altogether “AAL Applications”. Figure 12 The integration with universAAL is equivalent to “talking” to its buses with specific roles - shows the universAAL reference architecture with the adaptation mechanism used to integrate 3rd party services, as in the case of NESTORE applications.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 769643

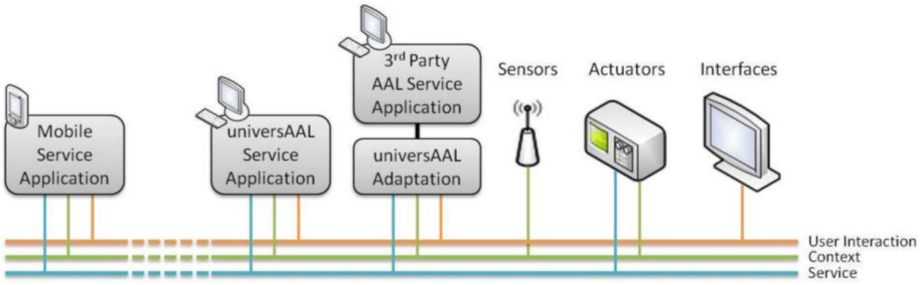

Mobile Service Application
universAAL Service Application
3rd Party AAL Service Application
universAAL Adaptation
Sensors
Actuators
Interfaces
User Interaction
Context
Service

Figure 12 The integration with universAAL is equivalent to "talking" to its buses with specific roles

The universAAL platform is enriched with the technological requirements deriving from the NESTORE project.

In NESTORE, devices installed in the user's home are abstracted with the Web of Things (WoT) approach reusing existing and well-known Web standards (REST, JSON, MQTT). In this view, the integration with universAAL is equivalent to "talking" to its buses with specific roles. We use MQTT for creating the shared message queue among the NESTORE module and uAAL buses.

Figure 13 Separation of concerns in different MQTT topics and their relationships with uAAL buses - shows the interaction between the uAAL buses and the NESTORE applications by means of the above-mentioned aggregation of MQTT topics.

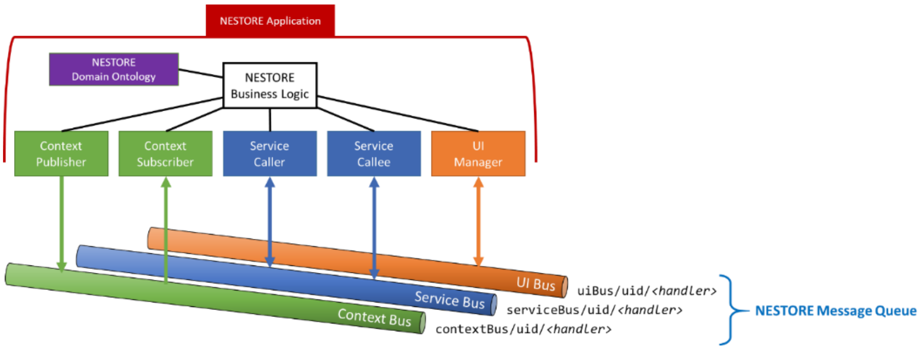


Figure 13 Separation of concerns in different MQTT topics and their relationships with uAAL buses

3.2.2 NESTORE MQ support for IoT

In NESTORE, devices installed in the user's home are abstracted with the Web of Things (WoT) approach reusing existing and well-known Web standards (REST, JSON, MQTT). In this view, the integration with universAAL is equivalent to "talking" to its buses with specific roles. In particular, we use MQTT for creating the shared message queue among the NESTORE module and uAAL buses.

MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple, and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimize network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices, and for mobile applications where bandwidth and battery power are at a premium [web4].

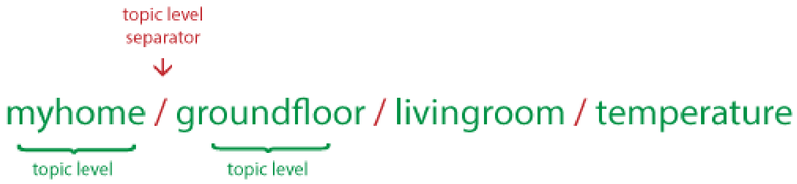


Figure 14 A generic MQTT topic

MQTT is based on the principle of publishing messages and subscribing to topics and they are used to decide on the MQTT broker which client receive which message. A topic is a UTF-8 string, which is used by the broker to filter messages for each connected client. A topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator in Figure 14 A generic MQTT topic).

3.3. Big Data sub-architecture

The Big Data Manager is a complex architectural component that represents a whole architecture itself. The following requirements must be fulfilled:

- Ingest high-velocity streams of data coming from the devices and the simulation tools;
- Store large datasets of row data-driven;
- Process the streamed and the stored data;
- Expose a suitable (possibly SQL-like) interface for Analytics and Monitoring.

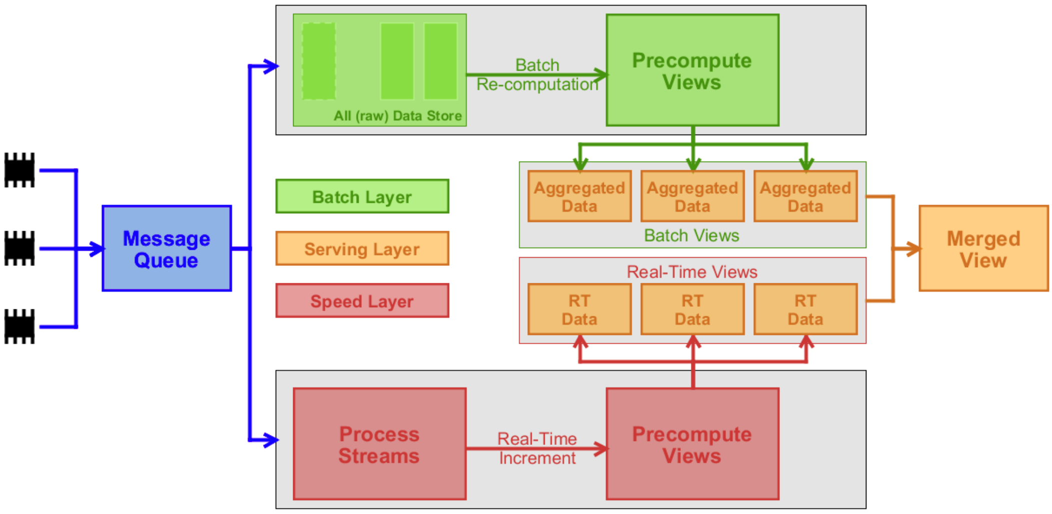


Figure 15 Lambda Architecture of the Big Data Component

The Big Data Component will adopt the Lambda Architecture (LA) (Fay Chang, 2008) as shown in Figure 15 Lambda Architecture of the Big Data Component. LA was introduced as a generic, linearly scalable and fault tolerant data processing architecture. LA is suitable for Big Data systems because it addresses data velocity, volume and variability altogether.

The input for LA is raw data that is typically streamed from external systems (hardware devices, software applications etc.) often via a specific data ingestion tool as LinkedIn Gobblin, Apache NIFI or Apache Flume.

LA is a three-layer architecture. The Batch Layer is responsible for batch processing the input data. A master dataset (often referred to as Data Lake) stores the data in different formats. A typical scenario is that the data ingested by the system are appended to existing Big Data files (Hadoop Sequence Files). Pre-computation is applied periodically on batches of data. The purpose is to offer the data in a suitable aggregated form for different batch views. Note that the batch layer has a high processing latency because it is intended for historical data.

It is the purpose of the Speed Layer to offer a low latency, real-time view of the data. The speed layer processes the input data as they are streamed in and it feeds the real-time views defined in the serving layer.

The Serving Layer has the main responsibility to offer a merged view on the data. This view includes old data as well as the most recent data. The batch views must be indexed in the serving layer so that they can be queried in low latency, too. Therefore, this layer responds to queries coming from external systems.

Designing and setting up a Big Data environment, here in the form of a LA, is a complex task that starts with doing some structural decisions. In particular, the designer faces with the decision whether to build from scratch the system or rely on pre-assembled solutions (e.g., Cloudera CDH or Oryx2). The main drawback of complete off-the-shelf solutions is that they often include a large number of applications whose interactions can be problematic. With the aim at keeping the architecture the simplest possible, we therefore lean toward to create a LA picking only the software that will be considered strictly necessary.

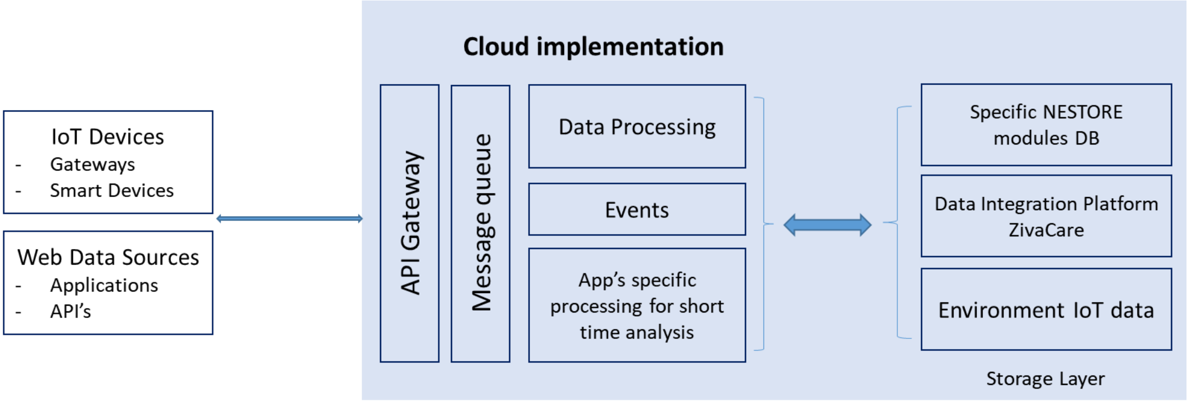


Figure 16 NESTORE Big Data sub-architecture

In what follows some high-level considerations about the possible technological alternatives are presented; among them we choose the elements to form a tentative architecture for the NESTORE specific Big Data architecture (Figure 16 NESTORE Big Data sub-architecture).

Batch layer

The field of Big Data is bursting with literally hundreds of tools and frameworks, each with specific characteristics; recently, however, appeared on the market some new solutions, that natively extend MapReduce (J. Dean, 2004) paradigm reduce and, among other things, provide a more flexible and complete programming paradigm paving the way to the realization of new and more complex algorithms.

The solution proposed for NESTORE platform to implement this layer, Apache Spark (M. Zaharia M. C., 2010), claims to be up to 100x faster than Hadoop on memory and up to 10x faster on disk. This is mainly due to a particular distributed, in memory data structure called Resilient Distributed Datasets (RDD). Shortly, Apache Spark attracted the interest of important players and gathered a vast community of contributors; only to mention a few: Intel, IBM, Yahoo!, Databricks, Cloudera, Netflix, Alibaba and UC Berkeley. Moreover, Spark implements both map-reduce and streaming paradigm, features out-of-the-box a SQL-like language for automatic generation of jobs and support several programming languages (Java, Scala, Python and R).

Stream processing engine

If the batch processing engine enables the analysis of large historical data (often referred to as Data at Rest), the stream processing engine is the component of the Lambda Architecture that is in charge of continuously manipulating the incoming data in quasi real-time fashion (i.e., the Data in Motion scenario). Recently, stream processing has increased in popularity. Only within the Apache Foundation, we identified several tools supporting different flavors of stream processing. Among them is Spark Streaming (M. Zaharia T. D., 2013), the tool proposed to be used to implement this layer.

Spark Streaming relies on Spark core to implement micro-batching stream processing. This means that the elements of the incoming streams are grouped together in small batches and then manipulated. As a consequence, Spark shows a higher latency (about one second). Spark Streaming is a valid alternative owing to the rich API, the large set of libraries, and its stability.

Spark can work in standalone mode featuring on its own resource manager or it can rely on external resource managers. Other resource managers exist (e.g. Apache Mesos) but they are related more to cluster management than on Big Data. Nonetheless, Spark can be executed over both YARN and Mesos.

All data store

A central role in the Lambda Architecture is played by the All Data Store, which is the service in charge of storing and retrieving the historical data to be analyzed. Depending on the type of data entering the system this element of the platform can be realized in different ways. In NESTORE we decided to implement it through different solution as are presented on the Chapter "**NESTORE Data**".

In case of one single NoSQL database particularly suitable for fast updates is recommended to be used Apache Cassandra (A. Lakshman, 2010). It is the most representative champion of the column-oriented group. It is a distributed, linear scalable solution capable to ensure high volumes of data. Cassandra is widely adopted (it is the most used column-oriented database) and features a SQL-like query language named CQL (Cassandra Query Language) along with a Thrift8 interface. As far as stream views are concerned, Cassandra has been successfully used to handle time series for IoT and Big Data.

Message queuing system

In a typical Big Data scenario, data flows coming from different sources continuously enter the system; the most used integration paradigm to handle data flows consists in setting up a proper message queue. A message queue is a middleware implementing a publisher/subscriber pattern to decouple producers and consumers by means of an asynchronous communications protocol. Message queues can be analyzed under several points of view, in particular policies regarding Durability, Security, Filtering,

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Purging, Routing and Acknowledgment, and message protocols (as AMQP, STOMP, MQTT) must be carefully considered.

Message queue systems are not a novelty and many proprietary as well as open source solutions have appeared on the market in the last years. Among the open source ones there is Apache Kafka (J. Kreps, 2011). A preliminary analysis seems to demonstrate that Kafka is the most widely used in big players' production environments as for instance in LinkedIn, Yahoo!, Twitter, Netflix, Spotify, Uber, Pinterest, PayPal, Cisco, Coursera among the others. Kafka is written in Java and originally developed at LinkedIn; it provides a distributed and persistent message passing system with a variety of policies. It relies on Apache Zookeeper (P. Hunt, 2010) to maintain the state across the cluster. Kafka has been tested to provide close to 200 thousand messages/sec for writes and 3 million messages/sec for reads, that is an order of magnitude more that its alternatives.

The Message Queue NESTORE platform implementation decision is presented under the Chapter "**Nestore MQ**".

Serving layer

This layer provides a low-latency storage system for both batch and speed layers. The goal of this layer is to provide an engine able to ingest different types of workloads and query them showing a unified view of data. The rationale is that the outcomes of the different computations must be suitably handled to later be further processed. In particular, batch views will contain steady, structured and versioned data whereas stream views will contain time-related data.

3.4 IoT sub-architecture

A typical IoT solution is characterized by many devices (i.e. things) that may use some form of gateway to communicate through a network to an enterprise back-end server that is running an IoT platform that helps integrate the IoT information into the existing enterprise. The roles of the devices, gateways, and cloud platform are well defined, and each of them provides specific features and functionality required by any robust IoT solution.

In picture (Figure 17 NESTORE IoT solution) NESTORE high level IoT sub-architecture is presented. Central point is represented by the IoT private cloud where all the computations and storage take place. Specific NESTORE platform applications are managing data on the private cloud, especially reading and processing the data sent by devices.

Different types of IoT devices as are defined on the Figure 3 Sensors and Sensors Module Sample (Dominique Guinard, 2016) can connect and push data intro the cloud. One special device is represented by gateway that enables machine-to-machine communication.

Gateway is a physical device or software program that serves as the connection point between the cloud and controllers, sensors and intelligent devices. All data moving to the cloud, or vice versa, goes through the gateway, which can be either a dedicated hardware appliance or software program. A gateway provides a place to preprocess that data locally at the edge before sending it on to the cloud. When data is aggregated, summarized and tactically analyzed at the edge, it minimizes the volume of data that needs to be forwarded on to the cloud, which can have a big impact on response times and network transmission costs.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

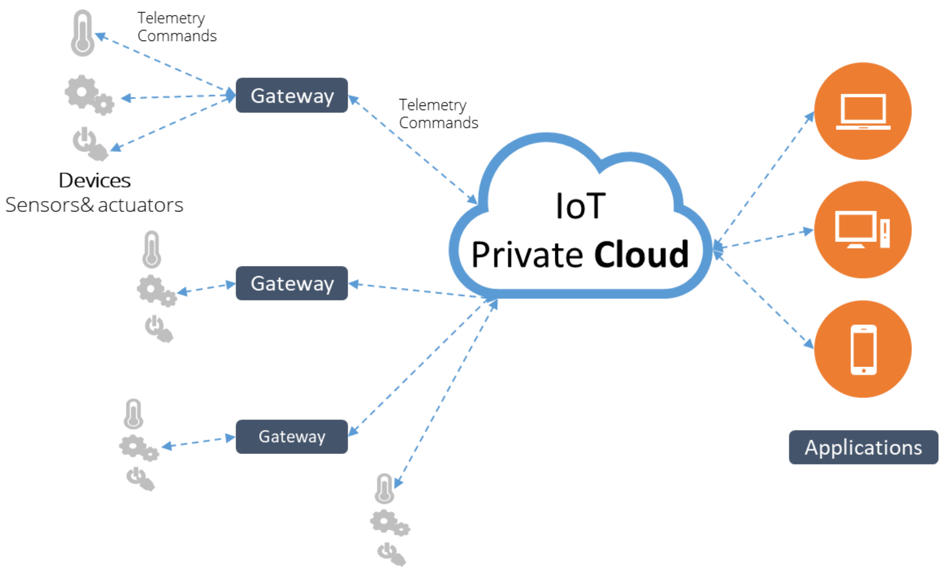


Figure 17 NESTORE IoT solution

IoT devices usually are focusing on transmitting acquired data over a Machine2Machine (M2M) networking layer (OSI layers 1 to 6) and we can find a lot of proprietary protocols like Bluetooth, ZigBee, etc. Usually these devices are connected to a IoT gateway.

On contrast the Web of Things relies exclusively on application level protocols and tools (OSI layer 7). Mapping any device into a Web mindset makes the Web of Things agnostic to the physical and transport layer protocols used by devices. In the Web of Things, devices and their services are fully integrated in the Web because they use the same standards and techniques as traditional Web sites. This means that can be written applications that interact with embedded devices in exactly the same way as would interact with any other Web service that uses Web APIs and in particular using RESTful architectures.

An IoT Solution requires substantial amount of technology in the form of software, hardware, and networking.

Cross stack functionality need to be considered for any IoT architecture(Figure 18 IoT components - by Eclipse Foundation):

- Security – Security needs to be implemented from the devices to the cloud. Features such as authentication, encryption, and authorization need be part of the solution stack.

- Ontologies – The format and description of device data is an important feature to enable data analytics and data interoperability. The ability to define ontologies and metadata across heterogeneous domains is a key area for IoT.

- Development Tools and SDKs – IoT Developers will require development tools that support the different hardware and software platforms involved.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

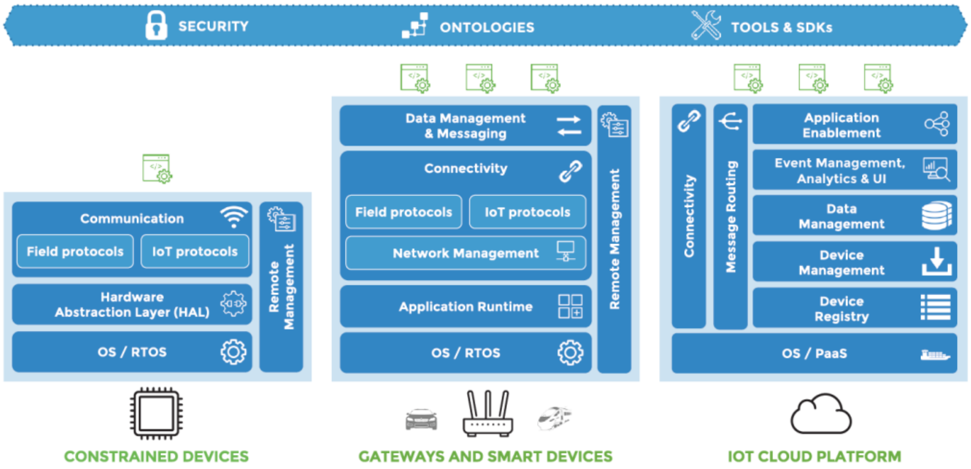


Figure 18 IoT components - by Eclipse Foundation

On the NESTORE platform we are implementing a simple IoT platform that reuse shared components and functions. (Figure 19 NESTORE IoT components)

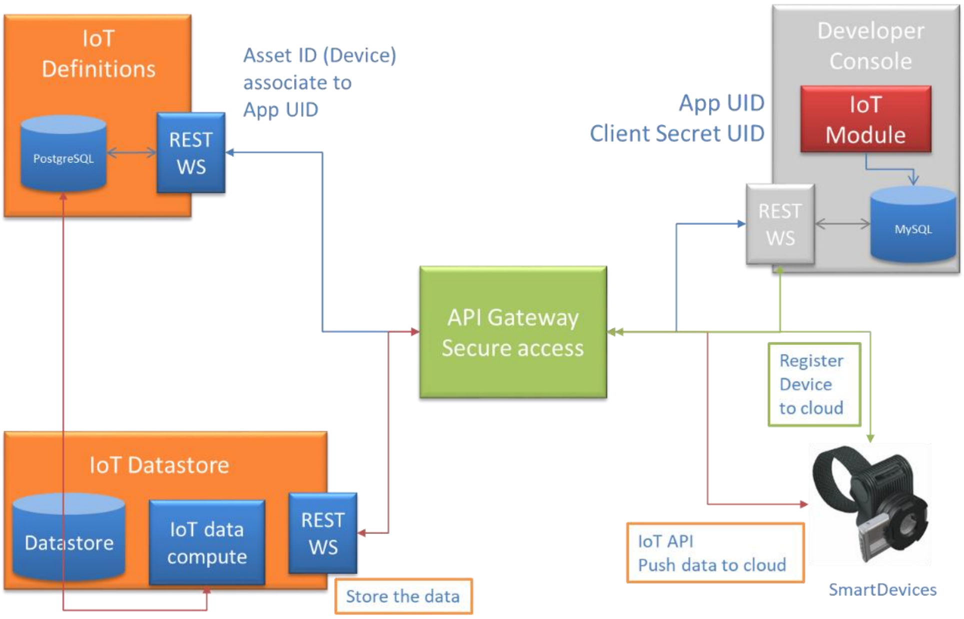


Figure 19 NESTORE IoT components

The proposed components assuring the IoT functionality and security. On the Developer Console all needed setting are created and users are enrolled. Each End User device is registered with the IoT and then data generated can be pushed on the IoT datastore. Under the IoT definitions the devices configurations are managed. All components are exposing a RESTful API for easy integration with all other NESTORE platform components.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

WoT Agent

As presented before one important role on IoT sub-architected is represented by the gateway that can be a dedicated hardware device or a software and under NESTORE platform we are defined it as wotAGENT and represent an essential IoT component.

To interface the IoT devices or other related data sources to the Private Cloud implementation of Agent as a software "driver" is required. Agents are software components that enable a centralized perspective on all aspects of the IoT network and central operation of the IoT network.

Because the M2M (machine to machine) devices are implementing different protocols (i.e. low-level serial links to IT protocols like web services) for communication, connectivity plus other parameters need a common layer – that is represented by the wotAgent.

Main functions of the wotAgent are:

- Translate the device – specific interface protocol into a reference protocol
    - The configuration of parameters, readings, events and other information are either send to an wotAgent ("push") or queried by an wotAgent ("pull") through a device-specific protocol on one side. The wotAgent will convert these messages into the protocol that Private Cloud requires. It will also receive device control commands from Private Cloud ("switch off that relay") and translate these into a kind of protocol the device requires
- Translate the specific domain model of the device intro a reference domain model
    - The Configuration parameters, readings, events, they all have their device-specific name (and possibly units). An wotAgent for a device will transform this device-specific model to the Private Cloud reference model. For example
- Enable secure remote communication using various network architectures
    - Devices can provide a protocol that is unsuitable for secure remote communication with Cloud environments. The protocol only supports local networking and does not pass through firewalls and proxies and it can contain sensitive data in clear text form. To avoid security issues like these, an wotAgent can be co-located to the device and provide a secure, internet-enabled link to the remote Private Cloud.

Deployment of the IoT can be realized around the wotAgent that can be implemented at different levels.

1) On the private cloud – then sensors must be technically able to send data to the cloud wotAgent

2) At the level of sensors network – then the wotAgent will be implemented on a local gateway device

3) On mobile device – the mobile device is responsible to acquire data from other specialized sensors or devices and the mobile wotAgent is transferring data to the cloud.

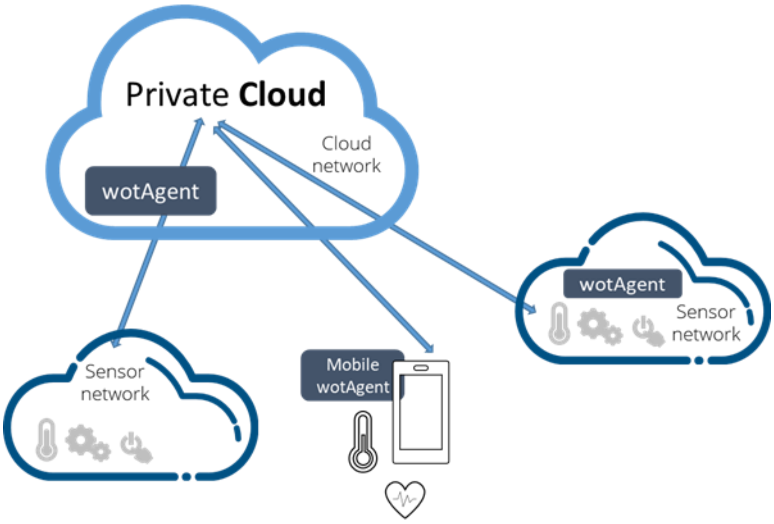


Figure 20 IoT deployment using wotAgent

First scenario supposes that:

- The device / sensor is using a communication protocol that is secured and Internet enabled
- There exist a VPN between the device/sensor network and Cloud

Device side wotAgents are running on the same network with the sensors and can run on a dedicated gateway, mobile phone, router, etc.. One must take in consideration the power needs of the wotAgent in respect to the running device and network as well the computation needs – memory, processors.

Datastore

For implementing the datastore that respond to the requirement of the NESTORE platform where selected a time series approach.

Time series involves the use of data that are indexed by equally spaced increments of time (minutes, hours, days, weeks, etc.). Due to the discrete nature of time series data, many time series data sets have a seasonal and/or trend element built into the data. All data generated by the NESTORE sensors are time series data.

For easy data push from the devices (environmental or wearable) a specialized API was designed. For each datatype defined by NESTORE under device management module a specific API resource is created.

For each API resource devices can send one data object or an array of objects with a maximum limit defined. The array is limited because of the needed resources like: data transfer channel – Internet speed and processing time of the array on the server side.

4. Shared components

4.1 Developer platform

The Developer Platform enables SaaS business model on top of different self-developed API resources. Expose API for integration with web, mobile and server-side applications.

Integration of specific business flow of the proprietary API in a SaaS business model using secure process.

Multi-application / Multi-projects approach can be easily handled with the Developer platform. The Developer Platform (DP) handle authorization, user management, as for defined application. Developer platform REST API support an easy integration with NESTORE applications – frontend and backend and enable proprietary API for SaaS business model.

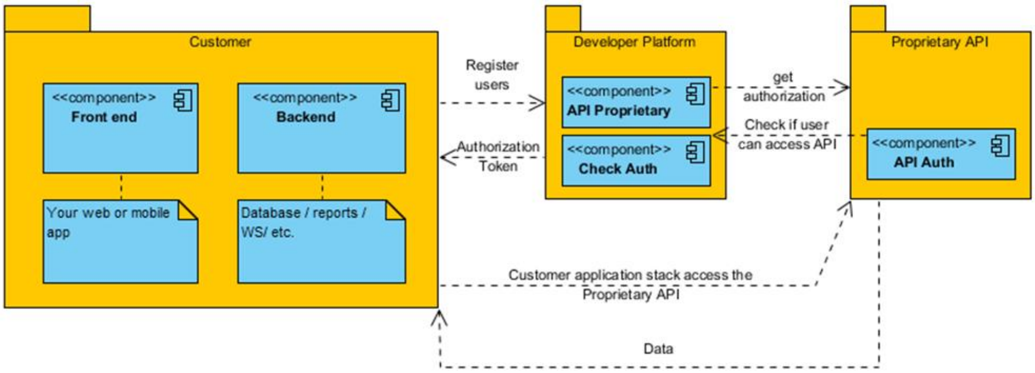


Figure 21 NESTORE Developer

At a structural level, architecture works like this:

1. Creating account on developer platform

    a. For the different NESTORE API integration dedicated Module is deployed that realize the integration

    b. Create Application's

    c. Assign Users for applications – each user is getting authorization code

    d. Authorization source (optional): API's can implement a dedicated authorization mechanism where developer platform is verifying the authorization for specific user

2. From NESTORE applications users are authorized to access data from the API's.

3. APIs are checking if a user has the authorization and if true serve data (optional – in case that is used on a trusted connection / trusted network is not needed)

Within the NESTORE Developer Platform are integrated different sub-systems like: gateway, IAM, third party data providers, developed NESTORE API's as one service for the NESTORE applications.

4.2 Device Management

Devices that are in use by the End Users need to be managed on the NESTORE platform. For this activity a dedicated developer module was created.

Because on NESTORE we are working with devices that are supposed to be installed at End User site (i.e. environmental sensors) as well devices as wearable the term Asset is used as generic term.

An asset is defined as an item of property owned by a person or company, regarded as having value and available to meet debts, commitments, or legacies (oxforddictionaries, 2018).

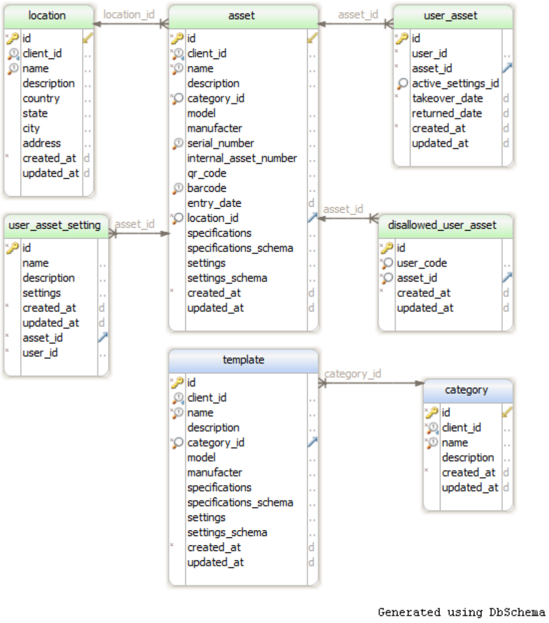


Figure 22 Device Management Database

An asset is modelled using the following data structures (Figure 22 Device Management Database):

- Categories of assets;
- Asset templates;
- Asset
- Asset - user;
- Asset location.

In order to ease the definition of assets, asset templates are used.

An asset template holds all the default and common characteristics for the assets derived from that particular template. Model and Manufacturer are examples of common characteristics. Other asset template specifications may be introduced.

An asset will be identified by serial number, bar code or QR code. For example, an authorized user may use a smartphone to scan the QR code of an asset and identify it in the application using a mobile app. An asset may also have custom properties that can be defined on the Developer Platform.

A set of custom settings can be defined for each instance as well the settings can be personalized for different exploitation scenarios – i.e. for each user – how it interacts with the asset! Only one configuration may be active at any moment.

The assets may have a geographical location associated with them to know where an asset is currently physically located.

The Asset Module allow the management of:

- Applications
- Assets Templates
- Users
- Assets instances = what Device belong to the specific End User

Each asset instance can be assigned / un assigned to a specific user offering a simple support.

For NESTORE platform different assets templates where defined covering the environmental data, and other templates will have defined during the project execution.

Asset templates already defined:

- Temperature
- CarbonDioxide
- Distance
- Humidity
- Light
- Motion
- Voltage
- Location

For each data type we are following at least next elements:

- The data will be represented as follow:
- DeviceID – the UID of the device that sent data
- Unit – the measurement unit
- Value – the value
- Timestamp - ISO8601 date-time: Date and time of log entry

Data generated by each sensor will be stored in the datastore component.

4.3 Log Services

Logging is a fundamental part of NESTORE platform that is composed by many applications. Every application has a varying flavor of logging mechanism. On the NESTORE platform we consider logging events that are End User relating, like when End User data where acquired, when specific data algorithm where applied, etc. We are not considering logging debug data relative to a specific application; this level of logs is going to be implemented and managed at application level.

Log entry should answer following questions:

Who(UserUUID), **When** (Timestamp), **Where** (Context, applicationName, Database), **What** (SpecificAppAction-Category), **Result** (JSON format) - Figure 23 Log System Architecture

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Main benefit of log system:

1. Centralize your log storage. This lets you apply policies in one place. Centralizing logs reduces the complexity and risk of maintaining policies in multiple places.

2. Duplicated data could create problems when enforcing policies, one centralized source or data

3. Structured logs make it easier to mask or anonymize sensitive data; applications log directly in a structured JSON format.

4. Anonymize Sensitive Data Fields in Logs - Identify and anonymize sensitive data fields before data is shipped to remote storage. Multiple techniques could be applied like hashing, encryption or removal of sensitive data fields.

5. Encrypted Logs in Transit - usage only encrypted channels to transmit log data to central storage.

A well-designed logging system is a huge utility for system administrators and developers, especially the support team. Logs save many valuable hours for both the support team or developers. As users execute programs at the front end, the system invisibly builds a vault of event information (log entries) for system administrators and the support team.

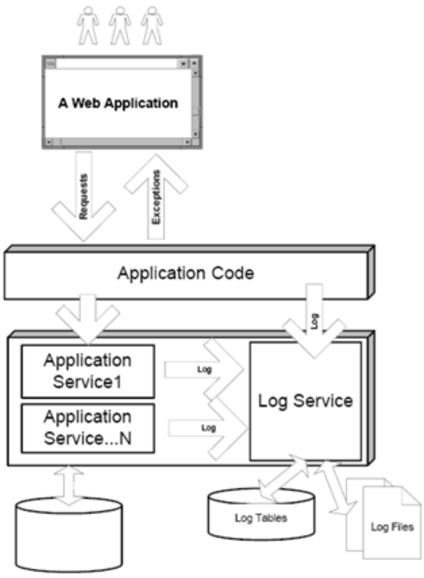


Figure 23 Log System Architecture

The Logging section for the Nestore Project is implemented in PostgreSQL with TimescaleDB. It consists of an API developed with PostgREST.

On the log system a JSON structure can be sent from the application as is in the bellow example.

```
{
"userid": "123456",
"applicationName": "logMeal",
"applicationUID": "ID to identify the thread or specific instance",
“applicationContext”: “JSON like data from the application”,
“applicationAction”: “exception/event/state/http”,
“applicationActionDetails”: “JSON with Action Details i.e. API call”
"actionHttpMethod": "GET/POST/PUT/DELETE",
"createdate": "2018-09-05T11:34:01.785474+03:00"
}
```

In the centralized logging system can be stored different log data about categorized on:

- exception
- event
- state
- http

Application exceptions should be logged: Major application exceptions can be logged on the central repository. Some exceptions are managed exceptions which are thrown by application as a warning or as a validation error to the user. These detailed debug logs - all validation errors or application exceptions are included in local logs.

Application events should be logged: For major components of an application may be log lifecycle events like start, stop and restart. Some security-related events may be logged such as unauthorized URL access attempts, user logins etc. Some resource thresholds may be exceeded and should also be logged.

Application states should be logged: In the application development process we should ask that “What could go wrong here in this code”. If this state occurs, application may throw an exception or log it (if we don’t want to interrupt current process) with some levels like Error, Warning or Information. In the centralized log system application must log different state that are relating to End Users – i.e. lunching profiling algorithm.

HTTP requests should be logged: Platform integrates a lot of applications that are integrated with help of Web Services. We may need what is coming from the user with full parameter details and details of each Web Service call for to trace the execution.

On the Annex “**Log System**” implementations details are presented.

4.4 Identity management

Identity management, also known as identity and access management (IAM) is referred as the security discipline that "enables the right individuals to access the right resources at the right times and for the right reasons" (Gartner, 2018). It addresses the need to ensure appropriate access to resources across NESTORE heterogeneous environments and to meet increasingly rigorous compliance requirements.

The core objective of IAM systems is one digital identity per End User individual. Once that digital identity has been established, it must be maintained, modified and monitored throughout each user’s “access lifecycle.”

IAM systems provide administrators with the tools and technologies to change a user’s role, track user activities, create reports on those activities, and enforce policies on an ongoing basis.

By using a IAM system is a required step to respond at the compliancy requirement i.e. GDPR, HIPPA by providing the tools to implement comprehensive security, audit and access policies.

IAM systems allow NESTORE to extend access to its information systems across a variety of on-premises applications, mobile apps, and SaaS tools without compromising security. By providing greater access to outsiders, NESTORE can drive collaboration, enhancing productivity, End User satisfaction and research and development.

A typical IAM system comprised four basic elements: a directory of the personal data the system uses to define individual users (identity repository); a set of tools for adding, modifying and deleting that data (related to access lifecycle management); a system that regulates user access (enforcement of security policies and access privileges); and an auditing and reporting system (to verify what's happening on your system).

Main IAM capabilities that are used on the NESTORE platform are:

- Authentication: Verification that an End User is who/what it claims to be using a password, or distinctive behavior such as a gesture pattern on a touchscreen.

- Authorization: Managing authorization information that defines what operations End User can perform in the context of a specific application.

- Roles: Roles are groups of operations and/or other roles. For example, a user administrator role might be authorized to reset a user's password, while a system administrator role might have the ability to assign a user to a specific server.

- Interchange: The SAML protocol is a prominent means used to exchange identity information between two identity domains (Working Groups - Identity Commons, 2018). OpenID Connect is another such protocol (OpenID, 2018).

User access

User access enables End Users to assume a specific digital identity across NESTORE applications, which enables access controls to be assigned and evaluated against this identity. (Figure 24 NESTORE IAM Web Log In UI). The use of a single identity for a given user across multiple systems eases tasks for administrators and users. It simplifies access monitoring and verification and allows the organizations to minimize excessive privileges granted to one user. User access can be tracked from initiation to termination of user access.

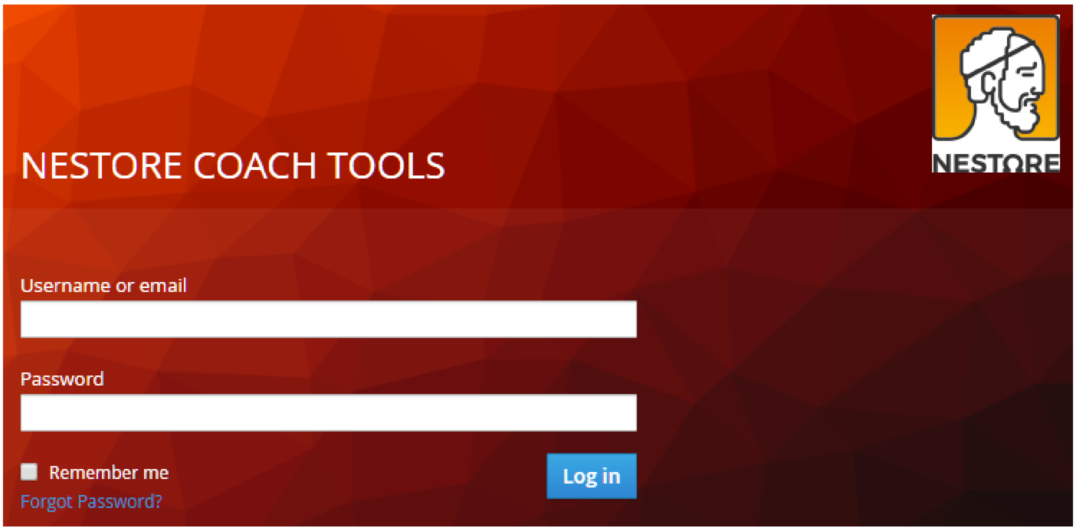


Figure 24 NESTORE IAM Web Log In UI

Single sign on

Single sign-on (SSO) is a session and user authentication service that permits a user to use one set of login credentials (e.g., name and password) to access multiple applications. The service authenticates the end user for all the applications the user has been given rights to and eliminates further prompts when the user switches applications during the same session. On the back end, SSO is helpful for logging user activities as well as monitoring user accounts.

This means that NESTORE applications don't have to deal with login forms, authenticating users, and storing users. Once logged-in to IAM, users don't have to login again to access a different application.

This also applied to logout. IAM provides single-sign out, which means users only have to logout once to be logged-out of all applications that use NESTORE IAM.

Identity federation

As the name implies, identity federation comprises one or more systems that federate user access and allow users to log in based on authenticating against one of the systems participating in the federation. (Figure 25 Identity Brokering and Social Login) This trust between several systems is often known as "Circle of Trust". In this setup, one system acts as the Identity Provider (IdP) and other system(s) acts as Service Provider (SP). When a user needs to access some service controlled by SP, he/she first authenticates against the IdP. Upon successful authentication, the IdP sends a secure "assertion" to the Service Provider. "SAML assertions, specified using a markup language intended for describing security assertions, can be used by a verifier to make a statement to a relying party about the identity of a claimant. SAML assertions may optionally be digitally signed.

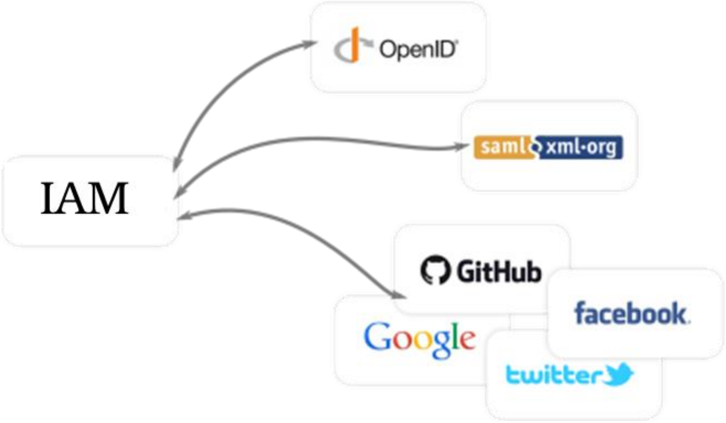


Figure 25 Identity Brokering and Social Login

NESTORE IAM

The identity and access management vendor landscape are a crowded one, consisting of both pureplay providers such as Okta and OneLogin and large vendors such as IBM, Microsoft and Oracle. Below is a list of leading players based on Gartner's Magic Quadrant for Access Management, Worldwide, which was published in June 2017. (Figure 26 Gartner 2017 Magic Quadrant for Access Management)

For implementation of the IAM NESTORE platform we select an open source alternative – KeyCloak. (keycloak, 2018) Keycloak is an open source Identity and Access Management solution aimed at modern applications and services. It makes it easy to secure applications and services with little to no code.

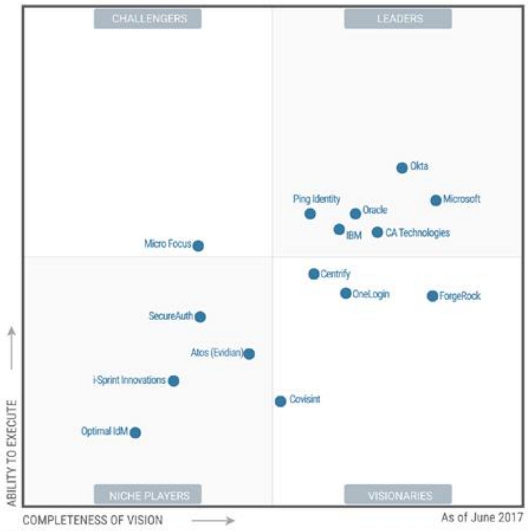
Keycloak is based on standard protocols and provides support for OpenID Connect, OAuth 2.0, and SAML.

Keycloak provides fine-grained authorization services as well in case that role-based authorization is not responding to different NESTORE applications needs. This allows to manage permissions for all services from the Keycloak admin console and gives you the power to define exactly the policies are needed.

Client Adapters

Keycloak Client Adapters makes it easy to secure NESTORE applications and services. Using adapters available for several platforms and programming languages or using standard protocols that can be used: OpenID Connect Resource Library or SAML 2.0 Service Provider library.

It is supported to use a proxy to secure NESTORE applications which removes the need to modify all the specific NESTORE application at all.



Source: Gartner (June 2017)

Figure 26 Gartner 2017 Magic Quadrant for Access Management

Admin and Account Management Console

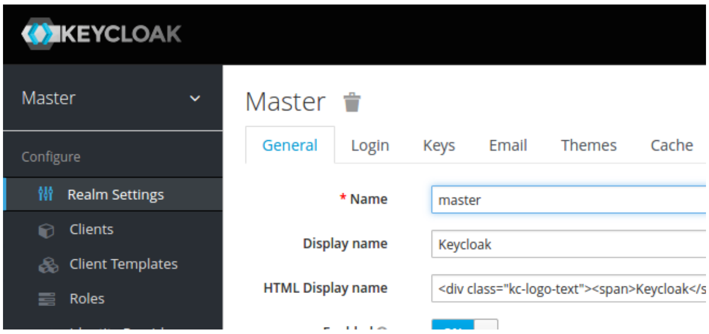
Through the admin console administrators can centrally manage all aspects of the Keycloak server (Figure 27 IAM Admin Console).

- Enable and disable various features.
- Configure identity brokering and user federation.
- Create and manage applications and services, and define fine-grained authorization policies.
- Manage users, including permissions and sessions

Through the account management console End Users can manage their own accounts (Figure 28 IAM Account Management Console). They can update the profile, change passwords, and setup two-factor authentication. End Users can also manage sessions as well as view history for the account.

In case of social login or identity brokering End Users can also link their accounts with additional providers to allow them to authenticate to the same account with different identity providers.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

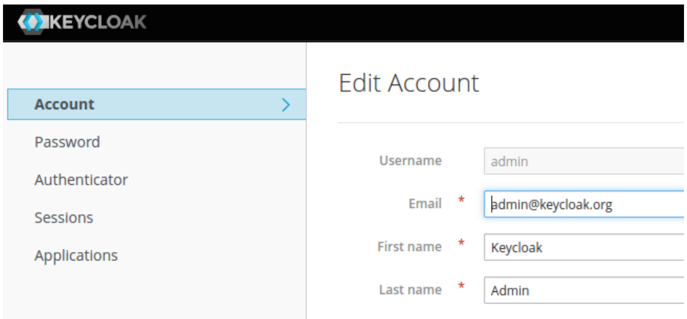


* Name: master

Display name: Keycloak

HTML Display name: <div class="kc-logo-text"><span>Keycloak</s

Figure 27 IAM Admin Console



Username: admin

Email *: admin@keycloak.org

First name *: Keycloak

Last name *: Admin

Figure 28 IAM Account Management Console

5. NESTORE Data

5.1 Data Sources

In the NESTORE are plenty of different data sources (such as sensors, data coming from mobile apps, data produced from the system itself, etc.). For collecting and organizing all these data, different in origin, size and shape it is proposed to use a unified official - so-called - "**NESTORE DATA MODEL**" and is detailed based on the NESTORE ontology developed on the WP2 and described as uAAL ontology under the "D2.3 - The NESTORE specific ontology" (NESTORE, D2.3 - The NESTORE specific ontology, 2018).

Such a data model describes how data should be formatted. This structure is very important to maintain a unique representation of data among services and apps that share them for guaranteeing interoperability among the different components of the system.

Next data sources where identified:

- NESTORE sensors - integrated / NESTORE developed, collecting information about End User – environmental and body data
- Third party's sensors, collecting information i.e. Fitbit
- Data coming from smartphone's sensors (accelerometer, gyroscope, compass, GPS, etc.)
- Data coming from NESTORE applications such e-diary, game and other apps describing the End User lifestyle and behavior.

Under the D2.3 is to provide a formal description of the integrated knowledge provided by Task 2.5 (Conversion of the knowledge into ontology) to guarantee interoperability and integration of NESTORE data model to the UniversAAL platform.

5.2 End User Profile

The definition of a common reference model, which represents the End User Profile, is proposed as a solution to harmonize data from different sources, providing in this way a systematic manner to classify and integrate the valuable knowledge. The common data model collects all the information and defines the reciprocal relationships between the individual's different aspects, aggregating and unifying all the information, thus improving significantly the potential of the NESTORE system. The definition of such virtual individual model stems from a conceptual framework of relations linking status, behaviors of the individual in different domains as well the context and the relation.

Following the approach of recent researches in domain (e.g. the Pegaso VIM [1] and the related uAAL models [2]), the reference model is represented through a set of ontologies developed under the WP2, which offers the possibility to represent formal semantics. An ontology-based approach offers significant key advantages to the whole project because it enables to:

- Represent a formal semantics.
- Efficiently model and manage distributed data.
- Ease the interoperability of different applications.
- Exploit generic tools that can infer from and reason about an ontology, thus providing a generic support that is not customized on the specific domain.

Due the fact that NESTORE is embracing the uAAL detailed data structures about the End User Profile are delivered under the "**D2.3 The NESTORE specific ontology**".

5.3 Third party data providers

5.3.1 LogMeal

LogMeal is an API (Application Programming Interface) developed by engineers and researchers from the University of Barcelona, aimed at satisfying various needs directly related to the automatic analysis of foods, both small and large companies and people for individual use from an image.

LogMeal's API communications are based on the HTTP protocol. Detailed documentation is available on the Annex "LogMealAPI" explained with example.

The LogMeal API is requiring the preregistration and is managing two main concepts: Company and Final User. For NESTORE platform the Company is represented by the NESTORE itself and Final Users are NESTORE End Users.

For LogMeal API the two types of Users: companies and final_users, which belong to a certain company. The Users of type company are allowed to create and manage final_users, but are not allowed to get recognition information from an image. The recognition service is only enabled for final_users.

The first step to use LogMeal's API is to SignUp your company with an email, where you will receive a token (let's call it a cToken). This token must be used allowing the creation of a new final_user. After this, you can create final_users for your company. A token (let's call it a uToken) for this final_user will be generated and returned, which must be used any following call reserved for final_users.

First of all, sign up your company sending your company name and email. This has to be done once only. A token will be generated and sent to your email (cToken).

The url for accessing this service is http://www.logmeal.ml:8088/companySignUp

With the cToken generated you can add users to your company. This users will have access to image recognition information. A token for this user will be generated and returned in response, let's call it uToken. Url is http://www.logmeal.ml:8088/userSignUp.

The API of LogMeal implements two use cases (see Figure 29 Use case diagram of LogMeal):

1. Analyse image

Eurecat system calls LogMeal API sending an image and the uToken corresponding to the specific user. LogMeal algorithms analyse this image, giving information about the dish type, food category (if dish type is food) and dish class. This extracted information and an image id are sent back to Eurecat system.

1.1 If the dish type ('drinks', 'ingredients', 'food' or 'no food') predicted from LogMeal algorithms is not the correct one, Eurecat system calls again to LogMeal API specifying the image id and the correct type.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

LogMeal algorithms are forced to predict a dish class from this specific dish type, and the recognition results are sent back to Eurecat system.

2. Confirm food & get recipe

When the recognition information is sent to Eurecat system, some proposals (from 3 to 5 depending of dish type) are included concerning the dish class. Due to the large number of dishes recognized from LogMeal algorithms it is hard to assure that the most probable recognized dish class will be the correct one. At this point, the user confirms one of these proposed dish classes (or select a new one) and Eurecat system calls LogMeal API sending this confirmed dish class and the image id. The generated recipe of this dish is sent back to Eurecat system.

The core of our API is made up of a set of models based on Deep Learning, Computer Vision and Automatic Learning that are applied to extract relevant visual information from food images. Each of these models constitutes a particular type of service provided by the API, among which: Food detection vs. non-food; the recognition of the family of food, for example: meat, dairy products, etc .; the recognition of food, for example: hamburger, dumpling, sushi, etc. ; the recognition of ingredients, for example: egg, cream, salt, etc .; the extraction of nutritional information, for example: calories, carbohydrates, proteins, etc..

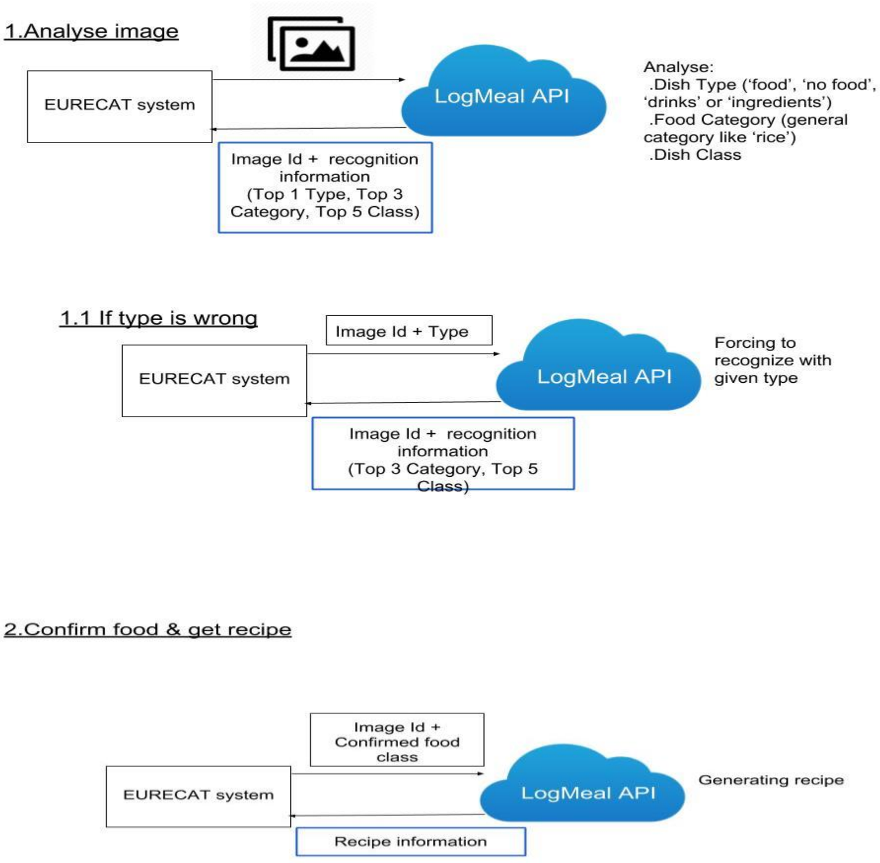

1.Analyse image
EURECAT system
LogMeal API
Image Id + recognition information (Top 1 Type, Top 3 Category, Top 5 Class)
Analyse:
.Dish Type ('food', 'no food', 'drinks' or 'ingredients')
.Food Category (general category like 'rice')
.Dish Class
1.1 If type is wrong
Image Id + Type
EURECAT system
LogMeal API
Forcing to recognize with given type
Image Id + recognition information (Top 3 Category, Top 5 Class)
2.Confirm food & get recipe
Image Id + Confirmed food class
EURECAT system
LogMeal API
Generating recipe
Recipe information

Figure 29 Use case diagram of LogMeal

LogMeal contains the following components:

- LogMeal database - a database of 300 food classes with 200,000 tagged images, a database of more than 3000 ingredients and 13 categories of foods with more than 200,000 tagged images.
- Algorithms for the recognition of foods - 1 API for automatic recognition of foods 300 classes.
- Algorithms for the recognition of food category - 1 API for automatic recognition of foodstuffs 13 families.
- Algorithms for the location of foods - 1 API for automatic recognition of foods 13 families.
- Algorithms for the deducing of food ingredients

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

5.3.2 ZivaCare

The already developed ZivaCare (zivacare, 2018) system is proposed as a collector system for NESTORE data. In principle, a NESTORE sensor will send data to a smartphone through Bluetooth connection (as depicted in Figure 30 The process of gathering user data). A NESTORE mobile application will gather these data and will send it through Internet to the NESTORE Cloud. There, the NESTORE Sensor Data Connector will relay these data to ZivaCare.

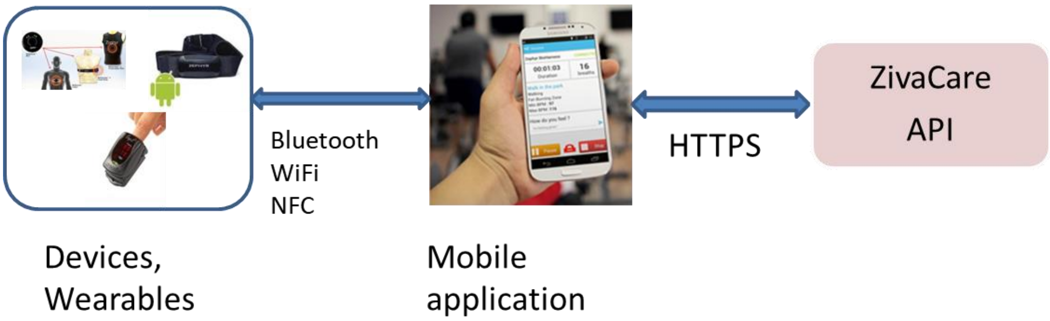


Figure 30 The process of gathering user data

From a technical point of view, an overview of the ZivaCare system is presented in the following picture.

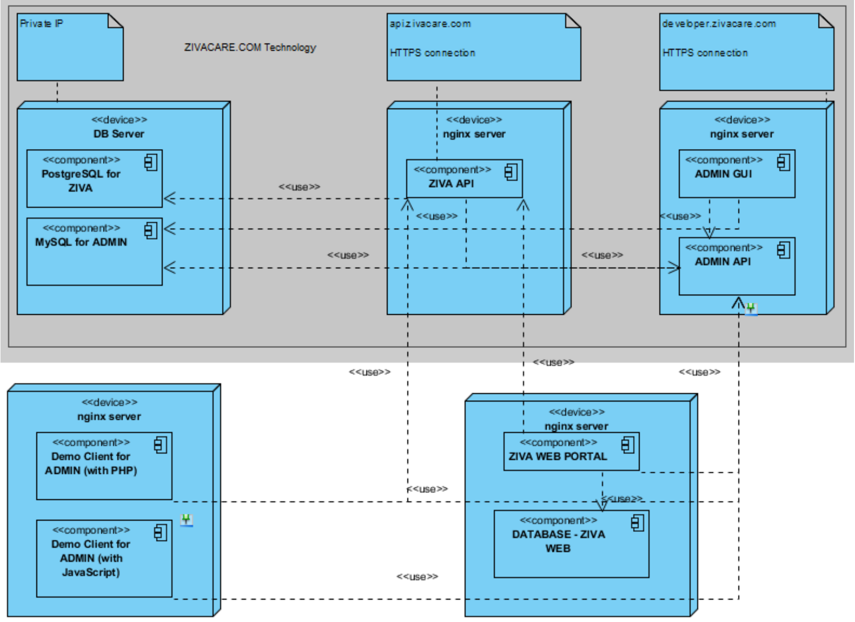


Figure 31 The ZivaCare system

ZivaCare has the following main components: Ziva API and Ziva Developer. Ziva API provides a unified and normalized API for accessing users' health data. Developers can create applications that retrieve data from a variety of data sources and enable users to share those data with applications of their choice. Ziva Developer also provides an administration Graphical User Interface for developed and registered applications, their users, data and connections to different data sources (for NESTORE and third-party sensors).

The ZivaCare API platform gives access to an individual's full range of personal health data through a single API. The system handles authentication and authorization, user management, and data processing. This is done by retrieving user's data from a wide variety of applications, sensors, and testing services, and serving the data through REST API.

At a structural level, the ZivaCare architecture works in this way:

1. End User authorizes application(s) to access his data via a popup inside the application or platform.

2. The platform retrieves, processes, normalizes, and stores all the data from the devices and services that the user connects to the system.

3. The platform serves this normalized data through a wide variety of API calls for the application to ingest.

For the ADMIN API we have the following Key Scenarios (

Figure 32 Use case diagram for an administrator of a Client App):

- Register Developer / Log in existing one, using ADMIN GUI;
- Register New Applications, using ADMIN GUI;
- Edit / View Application details, using ADMIN GUI;
- Register Users for a specific Application, either from the ADMIN GUI, or from the Application using the API;
- Access protected ZIVA API for gathering application/user specific biometric data that are stored in ZIVA API from Client Application.

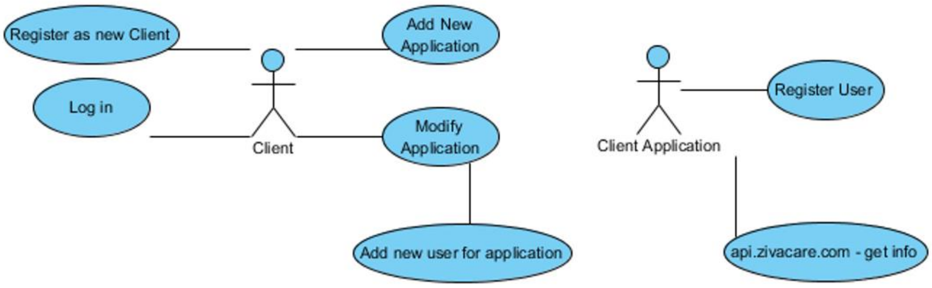


Figure 32 Use case diagram for an administrator of a Client App

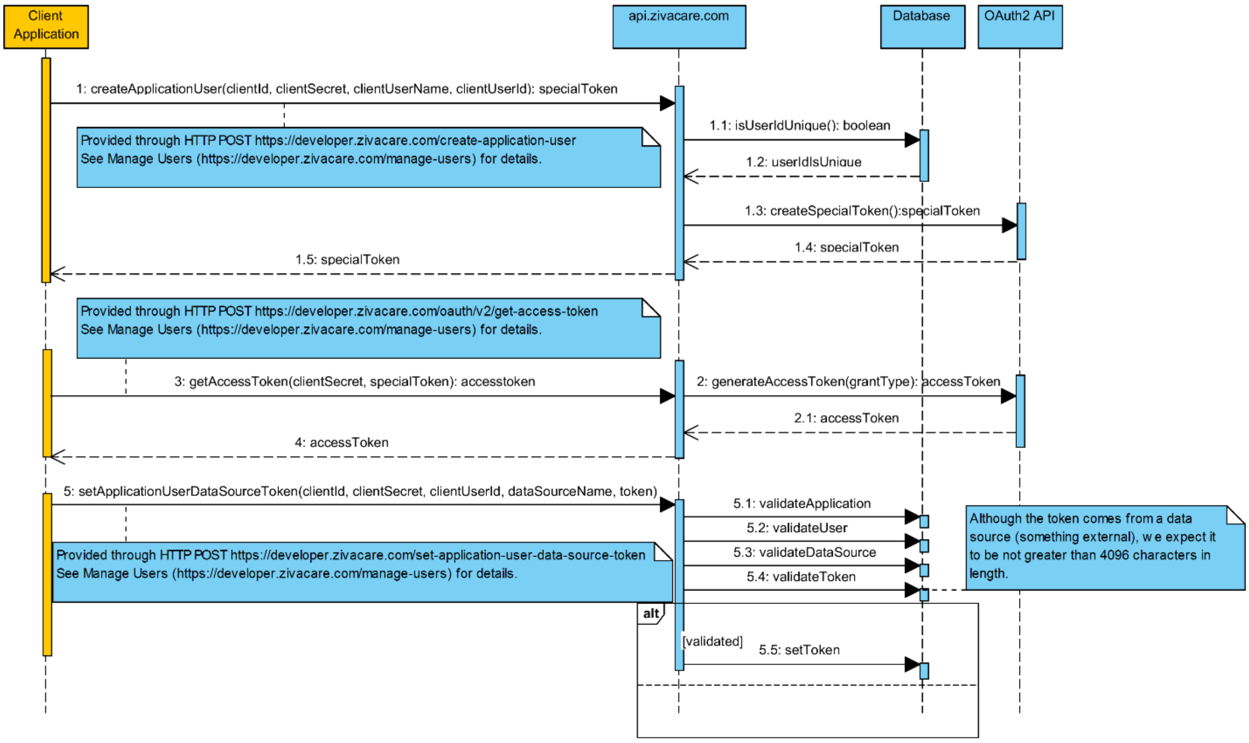

Client Application
api.zivacare.com
Database
OAuth2 API
1: createApplicationUser(clientId, clientSecret, clientUserName, clientUserId): specialToken
Provided through HTTP POST https://developer.zivacare.com/create-application-user
See Manage Users (https://developer.zivacare.com/manage-users) for details.
1.1: isUserIdUnique(): boolean
1.2: userIdIsUnique
1.3: createSpecialToken():specialToken
1.4: specialToken
1.5: specialToken
Provided through HTTP POST https://developer.zivacare.com/oauth/v2/get-access-token
See Manage Users (https://developer.zivacare.com/manage-users) for details.
3: getAccessToken(clientSecret, specialToken): accesstoken
2: generateAccessToken(grantType): accessToken
2.1: accessToken
4: accessToken
5: setApplicationUserDataSourceToken(clientId, clientSecret, clientUserId, dataSourceName, token)
5.1: validateApplication
5.2: validateUser
5.3: validateDataSource
5.4: validateToken
Provided through HTTP POST https://developer.zivacare.com/set-application-user-data-source-token
See Manage Users (https://developer.zivacare.com/manage-users) for details.
Although the token comes from a data source (something external), we expect it to be not greater than 4096 characters in length.
alt
[validated]
5.5: setToken

Figure 33 Sequence Diagram for creating an application user in the ZivaCare system

For example (see Figure 33 Sequence Diagram for creating an application user in the ZivaCare system), creating an application user with the ZivaCare system involves invoking the following Web Services.

1. https://developer.zivacare.com/create-application-user

INPUT: curl -X POST -d "clientId=CLIENT_ID&clientUserId=CLIENT_USER_ID&clientUserName=CLIENT_USER_NAME" https://developer.zivacare.com/create-application-user

(JSON) OUTPUT:

```
{ "token": "SPECIAL_TOKEN" }
```

Please note that CLIENT_USER_ID must be unique in the system. Typically, this parameter can be set to the user's e-mail.

2. https://developer.zivacare.com/oauth/v2/get-access-token

INPUT: curl -X POST -d "clientSecret=CLIENT_SECRET&specialToken=SPECIAL_TOKEN" https://developer.zivacare.com/oauth/v2/get-access-token

(JSON) OUTPUT:

```
{
"id": "CLIENT_ID",
"ziva_user_code": "ZIVA_USER_CODE",
"name": "CLIENT_USER_NAME",
"email": "CLIENT_USER_ID",
"access_token": "ACCESS_TOKEN"
}
```

ZivaCare, as part of the NESTORE platform is presented in Figure 34 The ZivaCare system, as part of the NESTORE ecosystem.

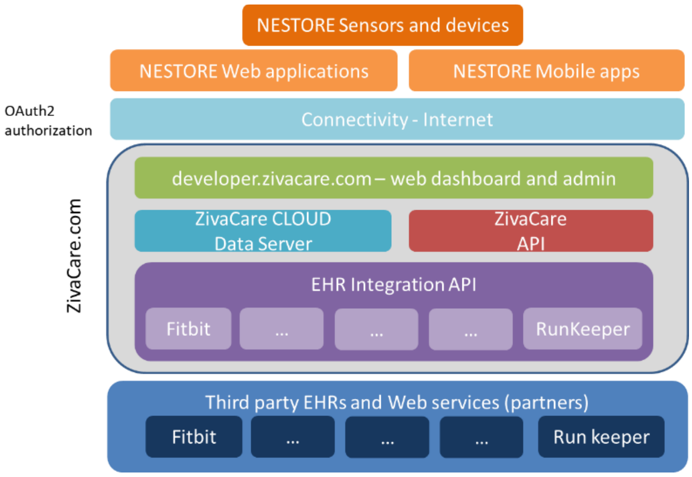


Figure 34 The ZivaCare system, as part of the NESTORE ecosystem

NESTORE sensors and devices send data to NESTORE web or mobile applications. These applications can transmit such data through the Ziva API into the Ziva Cloud, which will be part of the NESTORE Cloud. Obviously, stored data can also be later on retrieved from the Cloud, again through the Ziva API. Such data can be used in the NESTORE Web Portal for reporting purposes, recommender systems and so on.

ZivaCare provides a single authentication point and it operates with the OAuth2 [11] authorization protocol. Details about this are presented in the Annexe "**Zivacare**".

5.3.3 Third parties' sensors

The wearable market is strongly fragmented, and many actors are present in it, it is planned to use a data aggregator hub to include non-NESTORE devices and services data without regarding its origin. For such purpose, an aggregator service function presented by ZivaCare will be used. Other aggregator systems such as HumanAPI (humanapi, 2018) can be used. These aggregators are providing many benefits:

- Access to a wide variety of third party devices and services data (physical, sleep and physiological data).
- Unique API to access all data without regarding its origin.
- Maintained and updated API.
- Easy inclusion of future devices and services without further development.
- Secured and HIPAA-compliant infrastructure

The NESTORE system will include the ZivaCareAPI service, which will act as a hub for NESTORE End Users to register and authorize End User accounts from 3rd party providers (i.e. FitBit). Once the user authorizes and registers their account and sources, NESTORE would have access to the data from 3rd party backend services. For integrating this kind of data into the NESTORE cloud, we will use ZivaCare to interface.

The use of the ZivaCare system in this scenario is the one presented in Figure 35 The process of gathering user data from third-party systems.

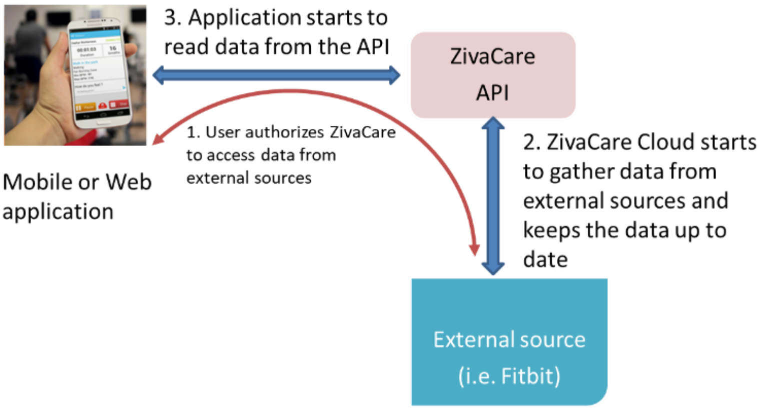

3. Application starts to read data from the API
ZivaCare API
1. User authorizes ZivaCare to access data from external sources
Mobile or Web application
2. ZivaCare Cloud starts to gather data from external sources and keeps the data up to date
External source (i.e. Fitbit)

Figure 35 The process of gathering user data from third-party systems

The process starts when a NESTORE End User gives authorization to the ZivaCare system to a third-party health data source. Existing user health data is automatically gathered from third-party sensors or devices, through a synchronization mechanism. Once these data are available in the NESTORE Cloud, NESTORE web and mobile applications can use it to display to the user different statistics. Obviously, the data will be secured and accessible only to the owner. Just the owner decides who can have access to her/his data.

As part of the ZivaCare system, **SyncEngine** is responsible with the synchronization of user's data between ZivaCare and various third-party data providers. The synchronization is done by invoking agents (which are provider specific), using a RESTful Web Service interface, based on a specific and configurable schedule. Details about this are presented in the Annex "**Zivacare**".

As already mentioned, the External source module shown in Figure 34 includes third-party systems. In particular, these systems are identified as: the Nokia Body+ smart scale and the Murata contact less bed sensor.

The Nokia Body+ smart scale

The Nokia Body+ (NB+) smart scale is a weight and body composition sensor. Additionally, NB+ allows the evaluation of fat mass, muscle mass, bone mass and water mass through a built in bioelectric impedance. In the Nestore Architecture, the NB+ sensor is responsible to send weight and body composition data of the user to the Nestore Cloud.

The Nestore cloud can exchange data with the Nokia Health cloud. In fact, Nokia has introduced the Health Mate API in order to allow developers to create applications that take advantage of the Nokia devices and the data they record. Data are only available after synchronization between user mobile application and Nokia platform.

As shown in Figure 36, the process starts through a first pairing between the NB+ and a smartphone/tablet. Successively, the sensor is able to send data to the Nokia Health cloud using API services (Nokia Health API references are available at http://developer.health.nokia.com/oauth2/). The Decision Support System (DSS), described in section 2.4.2, according to the Zivacare system described in section 5.3.2, stores data about users' weight and body composition.

Considering the Nestore scenario, the DSS developed in the Nestore cloud has to be able to perform correctly the authentication handshake. Further details about the software module in charge of elaborate the raw data retrieved will are available in deliverable D4.1. The final output of this module is refined data of users' weight and body composition as part of the NESTORE variables. Further details about the NESTORE variable from the physiological status are available in deliverable D3.2.1.

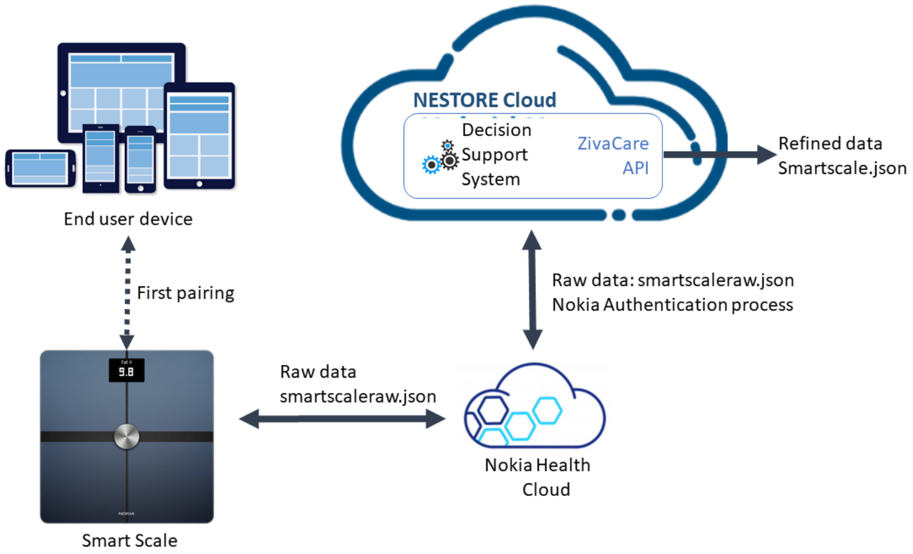


Figure 36 Architecture of the smart scale data process

Nokia Health platform requires OAuth2.0 in order to allow data exchange between software applications and users' device. Figure 37 shows the diagram of the OAuth process between the Nokia Health Server and an external software application.

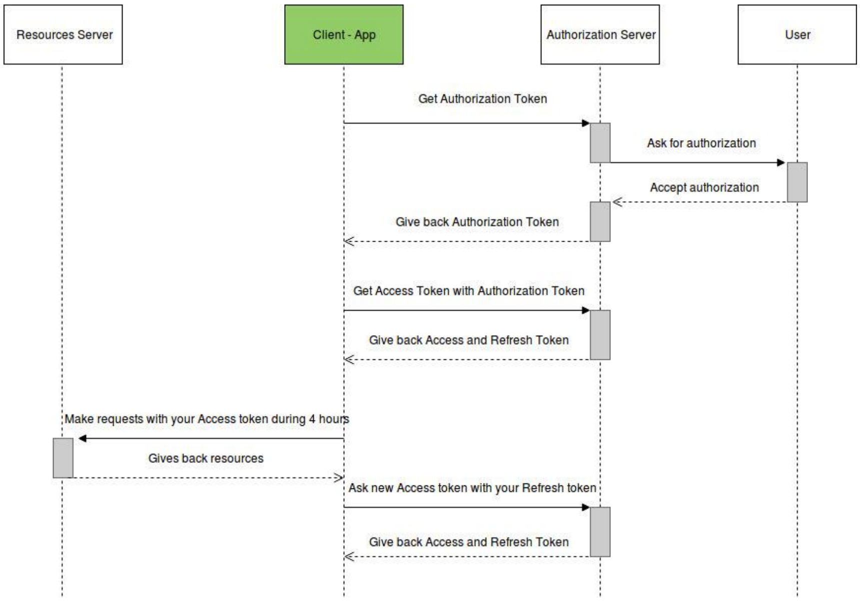

Resources Server
Client - App
Authorization Server
User
Get Authorization Token
Ask for authorization
Accept authorization
Give back Authorization Token
Get Access Token with Authorization Token
Give back Access and Refresh Token
Make requests with your Access token during 4 hours
Gives back resources
Ask new Access token with your Refresh token
Give back Access and Refresh Token

Figure 37 Diagram of the OAuth process between DSS and Nokia Health cloud

As shown in Figure 36, data produced into the smart scale data process are smartscaleraw.json and smartscale.json.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Smartscaleraw.json contains:

```
{
   "scaleID":"xxxxxxx-xxxx-xxxx-xxxx-",
   "timestamp":1531481898832,
   "weight":75.88,
   "fatMass":20,
   "muscleMass":40,
   "water":55,
   "boneMass":30,
   "BMI":20
}
```

Smartscale.json contains the information in smartscaleraw.json and the information required by the zivacare system as explained in section 5.3.2.

To summarise:

1. The NB+ smart scale performs an unique pairing with the end user mobile device

2. When the end user uses the smart scale, the Nb+ smart scale will sent end user data to the Nokia Health Cloud. In this phase, the data format produced is shown in smartscaleraw.json

3. The Nokia Health Cloud and the Nestore Cloud are synchronized according to the OAuth specification

4. The DSS provides to store the data according to the Zivacare API. In this phase, the data format produced is shown in smartscale.json

SLEEP MONITORING: THE SCA11H MURATA SENSOR

The Murata SCA11H sensor is an accelerometers-based contactless bed sensor able to collects users' physiological parameters during the sleep. In Nestore architecture, this sleep sensor is responsible to collect several parameters (e.g. respiration rate, heart rate, beat-to-beat timing).

From the Nestore architectural point of view, these raw variables are gathered using a direct connection, over TCP, between the sleep sensor and the Nestore cloud via Wi-Fi interface. The Decision Support System (DSS) will refine the raw data variables offering sleep quality parameters as final output. Figure 38 shows the pipeline of the data process. Details of the Remote Agent/ Refine module are available in deliverable D4.1.

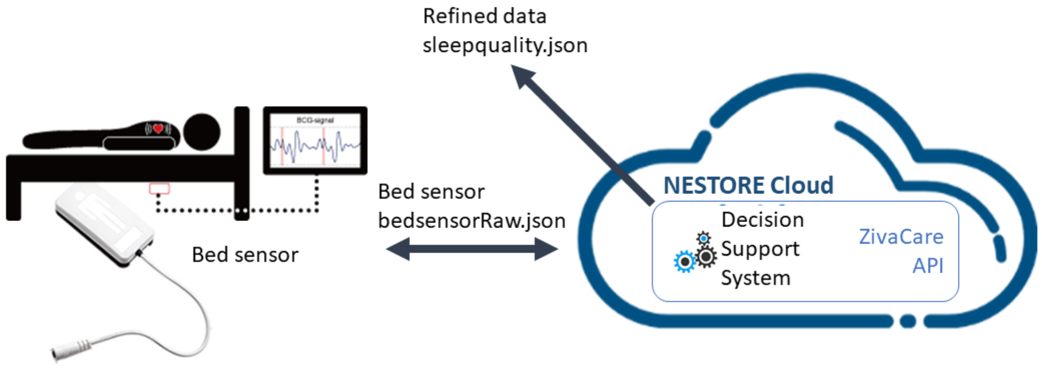


Figure 38 Architecture of the sleep monitoring data process

The sleep sensor is able to send data in local or cloud configuration. Considering the Nestore use case, the process starts when, during the deploying phase, the sleep sensor is set in cloud communication. In this phase, a sensor calibration is required. Furthermore, the sleep sensor calibration parameters with cloud server is required after power on/reset and then, by default, every 24 hours. Further details about the calibration requirements and setting are available in deliverable D3.2.1.

Through the cloud configuration mode, the sleep sensor is able to send processed data to cloud server according to cloud server interface API specification. The API contains interfaces for: sending sensor measurements to the cloud; upgrading the sleep sensor firmware; updating and sending notification about the sleep sensor calibration parameters.

As Figure 38 shows, data produced into the smart scale data process are bedsensorraw.json and sleepquality.json.

Bedsensorraw.json contains:

```
{
   "networkid":"net",
   "nodeid":"node",
   "istantvalues":"Timestamp,HR,RR, SV, HRV, SS, Status, B2B,B2B’, B2B’’"
}
```

Sleepquality.json contains the information required by the zivacare system as explained in section 5.3.2 and the following data structure:

```
{
   "UUID":"5CF8A15D8714",
   "timestamp":1531481898832,
   "perceivedCalmSleep":2,
   "sleepEfficiency":0.7,
   "totalSleepTime":100,
   "sleepOnset":1531481898832,
   "sleepOffset":1531481899832,
   "timeInBed":[
      {
         "start":1531481898832,
         "stop":1531481899832
      }
```

```
           ],
           "totalsleepTime":100,
           "wakeAfterSleepOnset":100,
           "sleepOnsetLatency":100,
           "awekenings":[
              {
                 "start":1531481898832,
                 "stop":1531481899832
              }
           ],
           "awake":[
              {
                 "start":1111,
                 "stop":1113,
                 "mins":100
              }
           ],
           "deep":[
              {
                 "start":1111,
                 "stop":1113,
                 "mins":100
              }
           ],
           "REM":[
              {
                 "start":1111,
                 "stop":1113,
                 "mins":100
              }
           ],
           "light":[
              {
                 "start":1111,
                 "stop":1113,
                 "mins":100
              }
           ],
           "sleepQualityIndex":5
        }

}
```

To summarise:

1. The sleep sensor is calibrated and set with the network configuration

2. When the end user sleeps, the sleep sensor will sent the raw data to the Nestore Cloud. In this phase, the data format produced is shown in bedsensorRaw.json

3. The DSS will refine the raw data and, according to the Zivacare system specification, will produce sleep quality parameters. In this phase, the data format produced is shown in sleepquality.json

5.4 BLE Beacon tags: air quality, socialization, and sedentariness detection

In order to infer about air quality and behavioural data of the Nestore end-user, BLE tags are deployed into the users' house. In particular, five BLE tags are provided to retrieve periodically temperature and humidity information and others five BLE tags are provided to the Nestore secondary user (e.g. friends, parents) as portable objects (e.g. keyfobs). Datasheet specifications of the hardware are provided in deliverable D3.2.1.

From a technical point of view, as Figure 39 shows, five BLE tags are deployed in point-of-interest and are responsible to gather temperature and humidity relative rate at different location. The sedentariness detection process uses the point-of-interest through a proximity-based approach. These data are gathered from the wearable device worn by the end user. Successively, the data are sent to the mobile wotAgent in order to be sent to the Nestore cloud through the wotAgent ad described in section 3.4. The data synchronization process and the authorization process are already described in section 5.3.3.

The socialization detection process requires the dissemination of five BLE tags to the secondary users. The beacons sent from these five tags will be used to infer information about the social interaction between the end user and the second users (more details will be available in deliverable D4.1).

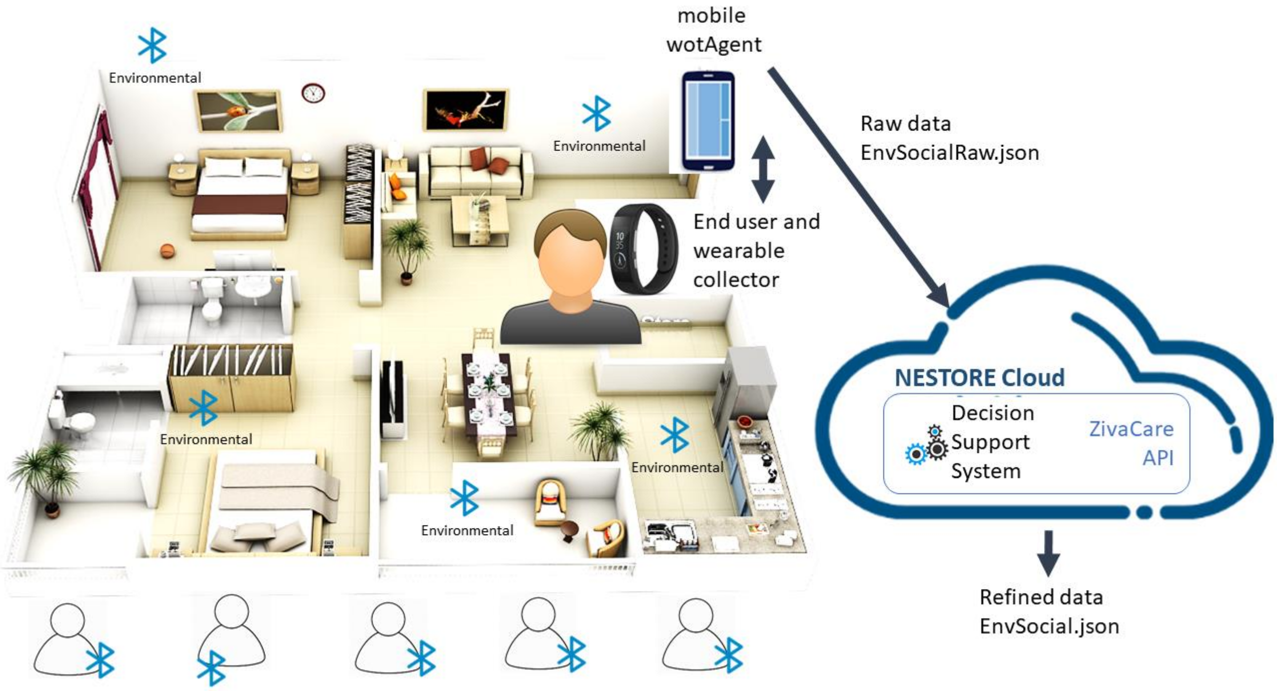


Figure 39 Environmental sensors: architecture of the data process

As shown in Figure 39, data produced into the BLE beacons data process are: envsocialraw.json and envsocial.json.

Envsocialraw.json contains:

```
{
   "beaconUUID":"b23143fe-ec81-47a8-9c3c-200000000003",
   "timestamp":1531481898832,
   "temperature":25.1,
   "humidity":13.5,
   "accelerometer":true,
   "battery":95.2,
   "rssi":-82
}
```

Envsocial.json contains the information required by the zivacare system as explained in section 5.3.2 and the following data structure:

```
{
   "socialdetection":[
      {
         "userIds":[
            1321321,
            1321322,
            1321323
         ],
         "starttime":[
            1531481898832,
            1531481898832,
            1531481898832
         ],
         "endtime":[
            1531481899832,
            1531481899832,
            1531481899832
         ]
      }
   ],
   "numberofinteract":3,
   "interactionsseconds":126,
   "interactionlocation":4
}
```

To summarise:

1. Five environmental BLE tags gathers temperature and humidity relative rate from point-of interest. Five BLE tags are given to second users.

2. The wearable collector gathers data from the environmental tags. When a socialization events occurs, the wearable collecotr gathers data from the socilization tags worn by the second users.

3. The wearable collector exchange data with the mobile wotAgent. In this phase, the data format produced is shown in EnvSocialRaw.json

4. The mobile wotAgent is synchronized with the DSS through the wotAgent.

5. The DSS will refine the raw data and, according to the Zivacare system specification, will infer about air quality, socialisation and sedentariness. In this phase, the data format produced is shown in EnvSocial.json

5.5 Data storage

As described in chapter “Cloud Services” it is supposed that - according to the system functional requirements - an architecture based on cloud services will be used.

Due to scalability needs of the overall system, the database used to collect and aggregate data should be focused on scalability and rapid access time. Different databases are used within NESTORE platform, from relation databases to NoSQL depending on application needs. On the Annex “**NoSQL Investigation**” are few selected NoSQL recommended to be used under NESTORE platform.

Data acquisition and storage involves two different areas that must be addressed separately to provide the best results.

Data acquisition is a process where high amount of data (in number) is pushed into the system at high rate. A good example could be monitoring user location changes during a day. In these cases, the two most significant constraints are endpoint availability and writing concurrency. The solution adopted should be able to scale up to millions of users offering an endpoint capable of managing continuous data pushed into the system.

Such kind of issues are solved nowadays using stream processing systems (such as Apache Storm or Amazon Kinesis) that decouple data acquisition from data processing, by the creation of a shared environment (a stream) where data is pushed with scalability rules that only must manage the endpoints. Each consumer then acquires data from the stream and manages all the required computation or aggregation, to consolidate it to a standard or to a NoSQL database.

NoSQL databases have a great variety and many different capabilities that can be leveraged based on the specific data model and structure. For these reasons, a complex architecture where not only a database exists have been proposed.

For storing the time series data, a specialized database can be used. Due to the data generated by different sensors NESTORE “big data” datasets are being dwarfed by another type of data, one that relies heavily on time to preserve information about the change that is happening. A time series database (TSDB) is a software system that is optimized for handling time series data, arrays of numbers indexed by time (a datetime or a datetime range). A workable implementation of a time series database can be deployed in a conventional SQL-based relational database provided that the database software supports both binary large objects (BLOBs) and user-defined functions. SQL statements that operate on one or more-time series quantities on the same row of a table or join can easily be written, as the user-defined time series functions operate comfortably inside of a SELECT statement. In case of using time series database on the NESTORE platform we consider to use TimeScaleDB (timescale, 2018).

6. Security and Privacy

6.1 Security

The growing globalization of data flows, via social networks, cloud computing, search engines, location-based services, etc., increases the risk that people can lose control of their own data.

In the EU as well in USA there are specific laws that define the legal and technical requirements for processing of health data. The most important are:

- EU General Data Protection Regulation (GDPR), which replaced the old directive in 2016,

- US Health Insurance Portability and Accountability Act (HIPAA) of 1996

The EU GDPR framework harmonize the rules regarding data protection of EU citizens' data better that directive 95/46/EC did.

GDPR defines high level requirements that each EU state member must implements; national laws define details that must be respected. Other laws are a must to be respected i.e. e-Privacy Regulation – or the new Cookie law envisioned for 2019.

The data protection strengthens citizens' rights and thereby help restore trust. Better data protection rules mean you can be more confident about how your personal data is treated, particularly online. The new rules will put citizens back in control of their data, notably through:

- The right to be forgotten: When you no longer want your data to be processed and there are no legitimate grounds for retaining it, the data will be deleted. This is about empowering individuals, not about erasing past events or restricting freedom of the press.

- Easier access owned data: A right to data portability will make it easier for you to transfer your personal data between service providers.

- Putting citizen in control: When citizen's consent is required to process your data, you must be asked to give it explicitly. It cannot be assumed. Saying nothing is not the same thing as saying yes. Businesses and organizations will also need to inform End User without undue delay about data breaches that could adversely affect you.

- Data protection first, not an afterthought: 'Privacy by design' and 'privacy by default' will also become essential principles in EU data protection rules – this means that data protection safeguards should be built into products and services from the earliest stage of development, and that privacy-friendly default settings should be the norm – for example on social networks.

Under the NESTORE project we are focused to provide GDPR compliancy Coaching applications.

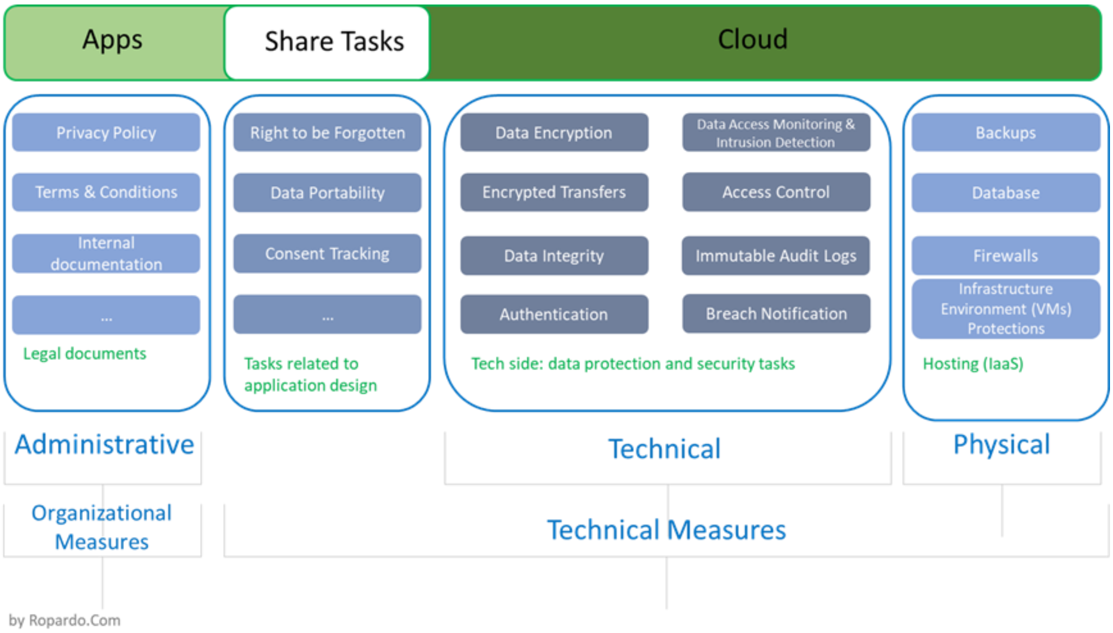


Figure 40 GDPR and Tech

We identify main categories (Figure 40 GDPR and Tech) where we must address solutions as follow:

1) Organizational measures ensure that data processing is legal (EU context as well country level); specific that an application is properly regulated within terms and conditions. That can address: involve collecting users' explicit consent, notify the data protection authority, define website privacy policy, perform security risk assessments etc.

2) Technical measures are really high level of importance, since GDPR does not define a way to comply with security and technical dispositions. This part affects the client, API, and backend security. Must be addressed requirements on authentication, access control, encryption of data in transfer and at rest (storage), secure audit log, security monitoring and updates, backup, and reliability (QoS and SLA).

Implementing technical requirements demands significant amount of work, knowledge, and time.

Physical requirements must be also addressed for both situations local equipment's that collect sensitive data as well for cloud datacenters/infrastructure.

6.2 Data Security

Data security refers to protective digital privacy measures that are applied to prevent unauthorized access to computers, databases and websites. Data security also protects data from corruption.

As described, the NESTORE platform will be cloud-based. Security is related to this technological choice and next is analyzed basic data security measures needed for NESTORE.

- Secure access – Customer access points, also called API endpoints, allow secure HTTP access (HTTPS) so that we can establish secure communication sessions with AWS services using SSL.

- Built-in firewalls – We can control how accessible your instances are by configuring built-in firewall rules – from totally public to completely private, or somewhere in between. And when our instances reside within a VLAN subnet, you can control egress as well as ingress.

- Unique users – The Identity and Access Management (IAM) tool allows to control the level of access NESTORE users have to infrastructure services. With IAM, each user can have unique security credentials, eliminating the need for shared passwords or keys and allowing the security best practices of role separation and least privilege.

- Multi-factor authentication (MFA) – IAM provides built-in support for multi-factor authentication (MFA) for use with IAM Accounts as well as individual IAM user accounts.

- Private Subnets – The VLAN service allows to add another layer of network security to our instances by creating private subnets and even adding an VPN tunnels between our institution/company network and our VPSes.

- Encrypted data storage – the data and objects stored on VPS are encrypted automatically using Advanced Encryption Standard (AES) 256, a secure symmetric-key encryption standard using 256-bit encryption keys.

- Security logs – provides logs of all user activity within NESTORE account. You can see what actions were performed on each of your Cloud resources, when by whom.

In the NESTORE Private Cloud infrastructure, the security responsibilities will be shared: IaaS has secured the underlying infrastructure and each application must secure anything that is stored / transfer on the infrastructure.

6.3 API Authorization

For the API authorization NESTORE is using single authorization point and it operates with the OAuth2 ((IETF), 2018) authorization protocol. The implemented OAuth2 mechanism is presented in Figure 41 OAuth Principles

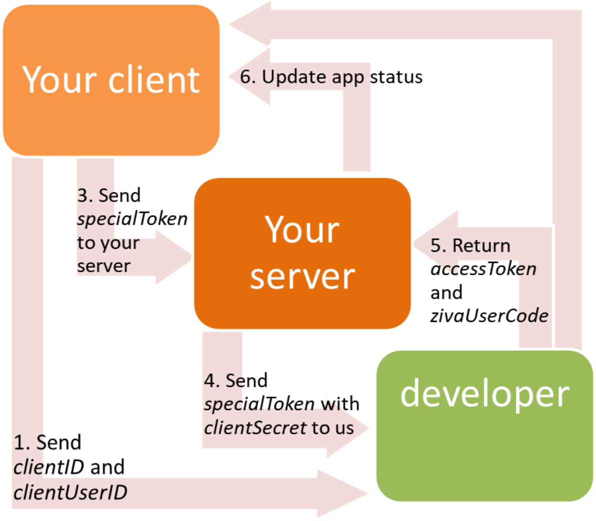

Your client
6. Update app status
3. Send *specialToken* to your server
Your server
5. Return *accessToken* and *zivaUserCode*
4. Send *specialToken* with *clientSecret* to us
developer
1. Send *clientID* and *clientUserID*

Figure 41 OAuth Principles

Main OAuth2 concepts are presented below.

Roles

The Third-Party Application: "Client"

- The client is the application that is attempting to get access to the user's account. It needs to get permission from the user before it can do so.

The API: "Resource Server"

- The resource server is the API server used to access the user's information.

The Authorization Server

- This is the server that presents the interface where the user approves or denies the request. In smaller implementations, this may be the same server as the API server, but larger scale deployments will often build this as a separate component.

The User: "Resource Owner"

- The resource owner is the person who is giving access to some portion of their account.

Creating an App

OAuth process is based on the application developer, and first must register a new app with the NESTORE services. When registering a new app, register basic information such as application name, etc. In addition, you must register a redirect URI to be used for redirecting users to for web server, browser Redirect URIs

Redirects URIs

- The service will only redirect users to a registered URI, which helps prevent some attacks. Any HTTP redirect URIs must be protected with TLS security, so the service will only redirect to URIs beginning with "https". This prevents tokens from being intercepted during the authorization process. Native apps may register a redirect URI with a custom URL scheme for the application, which may look like demoapp://re

Client ID and Secret

- After registering the app, a client ID and a client secret are generated. The client ID is considered public information, and is used to build login URLs, or included in Javascript source code on a page. The client secret must be kept confidential. If a deployed app cannot keep the secret confidential, such as single-page Javascript apps or native apps, then the secret is not used, and ideally the service shouldn't issue a secret to these types of apps in the first place direct. -based, or mobile apps.

Authorization

- The first step of OAuth 2 is to get authorization from the user. For browser-based or mobile apps, this is usually accomplished by displaying an interface provided by the service to the user. OAuth 2 provides several "grant types" for different use cases. The grant types defined are:

- Authorization Code for apps running on a web server, browser-based and mobile apps

- Password for logging in with a username and password
- Client credentials for application access
- Implicit was previously recommended for clients without a secret but has been superseded by using the Authorization Code grant with no secret.

These use cases are used within NESTORE platform and associated shared components. More technical details about OAuth can be accessed at [The OAuth 2.0 Authorization Framework (RFC 6749) - https://tools.ietf.org/html/rfc6749)

By using the OAuth2 NESTORE platform is ready for integration with different services like: google, amazon, etc. in terms of user identity management.

6.4 Privacy

A level of privacy is essential to note that in this document it refers only to the privacy aspects. It is also important to note that what is defined here are general recommendations that should be reflected in the design that is made of the different areas of the ecosystem.

Privacy is strictly related to the previous Security issue.

In the pilot test we will adopt the informed consent of the national institutions responsible for the Electronic Healthcare Data Record to match both privacy and integration with national standards, laws and requirements.

Local Ethical Committees are providing their specific requirement to be harmonized into the clinical trial documentation.

7. Deployment View

7.1 Product documentation

Documentation will be produced to assess the different needs for the different users involved in using the NESTORE system.

Those documents probably will be:

- The documentation related to each single module (each sub-system of the whole NESTORE)
- The integration and deployment documentation
- The End User guides/manuals

Firstly, the documents related to NESTORE modules will be written when developments will be on its final stage, and it will be included in the final deliverables. This type of documentation is more technical and provides all the information that are needed to interact with the module along with its functionalities and outputs.

This documentation will be produced internally by each sub-team composed by those partners in charge of each specific single module.

Secondly an integration and deployment documentation will be provided along with the integration infrastructure and the released ecosystem.

Thirdly, probably and if it will be considered consistent with the End Users interfaces, a short End User guide will be written to be used in the pilots and will be as clear as possible; the documentation will include some tutorials and users will be able to access it directly for their smartphone.

7.2 Internationalization

Internationalization and localization are means of adapting computer software to different languages, regional differences and technical requirements of a target market. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text.

The internationalization of the ecosystem is fundamental for its global use and dissemination, eliminating language barriers and cultural differences through the requirement for multi-lingual and culturally aware.

The target countries where the NESTORE applications will be distributed during the pilot phase are the following European Union countries: Spain, Italy and Netherland. The ecosystem will be translated and localized only in English.

8. References

References

(IETF), I. E. (2018). *The oauth 2.0 authorization framework: Bearer token usage*. Retrieved from tools.ietf.org/html/rfc6750

A. Lakshman, P. M. (2010). Cassandra. *ACM SIGOPS Oper. Syst. Rev., vol. 44, no. 2*, 35.

aws. (2018). Retrieved from aws.amazon.com

azure. (2018). Retrieved from azure.microsoft.com

cloud. (2018). Retrieved from cloud.google.com

Dominique Guinard, V. T. (2016). *Building the Web of Things: With examples in Node.js and Raspberry Pi.* Greenwich CT USA: Manning Publications Co.

Fay Chang, J. D. (2008). Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 4:1--4:26.

Gartner. (2018). *Identity and Access Management (IAM)*. Retrieved from blogs.gartner.com/it-glossary/identity-and-access-management-iam/

humanapi. (2018). Retrieved from humanapi.co

J. Dean, S. G. (2004). MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI - Symp. Oper. Syst. Des. Implement*, (pp. 137–149).

J. Kreps, L. C. (2011). Kafka: a Distributed Messaging System for Log Processing. *CM SIGMOD Work. Netw. Meets Databases*, 6.

keycloak. (2018). *keycloak*. Retrieved from keycloak.org/about.html

Liferay. (2018). *liferay*. Retrieved from liferay.com

Linthicum, D. (2000). *Enterprise Application Integration.* Essex: Addison-Wesley Longman Ltd. doi:0-201-61583-5

M. Zaharia, M. C. (2010). Spark : Cluster Computing with Working Sets. *HotCloud'10 Proc. 2nd USENIX Conf. Hot Top. cloud Comput*, (p. 10).

M. Zaharia, T. D. (2013). Discretized Streams: Fault-Tolerant Streaming Computation at Scale. *Sosp, no. 1*, 423-438.

NESTORE. (2018). *D2.3 - The NESTORE specific ontology.*

NESTORE. (2018). *D6.2 Evaluation of universAAL solution.* NESTORE.

Newman, S. (2015). *Building microservices.* O'Reilly Media, Inc.

OpenID. (2018). *OpenID Connect*. Retrieved from openid.net/connect/

oxforddictionaries. (2018). Retrieved from oxforddictionaries.com/definition/english/asset

P. Hunt, M. K. (2010). ZooKeeper: Wait-free Coordination for Internet-scale Systems. *USENIX Annual Technical Conference Vol. 8*, (p. 11).

rackspace. (2018). Retrieved from rackspace.com

timescale. (2018). Retrieved from timescale.com

Working Groups - Identity Commons. (2018). Retrieved from idcommons.org/working-groups/

zivacare. (2018). Retrieved from docs.zivacare.com

9. Annex.

9.1 OData Evaluation

OData (http://www.odata.org/) is a standard proposed by Microsoft in 2007 and approved as standard by OASIS (https://www.oasis-open.org/) in 2014, after reaching its current version (4.0).

Without going into details, the Standard describe how to write a RESTful API. It is similar to WSDL cause it requires to have a self-descripting interface and similar to JDBC cause it define how relationship and query, filters, should be handled.

The Protocol have several implementations in different languages, the only official one is for .NET language, maintained by Microsoft.

OData has some good part and ideas, but we decided not to use it and this section highlight the main reasons why.

1. A good standard has a wide adoption and is well know, unfortunately this is not true for OData which is used only by few companies like Microsoft itself (which is the proposer and maintainer), SAP and IBM. None of these companies adopt OData on every service.

Notable mentions are Netflix and eBay which both initially adopted OData dropping it shortly after without providing an official reason.

2. There are several implementations for many programming languages. The only officially supported and maintained one is the .NET implementation from Microsoft.

The specification is very complex and extensive, but it contains several optional features, as a result most implementations do not support OData fully, the only implementation fully supporting it is the .NET one.

Furthermore, each implementation has its own documentation with different technologies and skill set required to use.

Among the NESTORE partners no one has the knowhow to work in .NET. The alternatives are too fragmented and do not provide enough support of the specification to define a good enough subset of features to be used by the different partners.

Due to the complexity of the specification it is not feasible to implement it from scratch, the workload would be too big.

Design concerns

Due to the complexity all implementations require the adopter to describe the database structure to build the set of services on top of it. This creates a tight coupling between the RESTfull API and the Database exposing implementation details while making upgrades and evolution of the API really hard to achieve.

The protocol being generic tend to generate cluttered APIs which are less intuitive to use and harder to maintain then an API written and designed manually.

For example, this was the Netflix API URL to obtain the list of movies filtering by rating generated by OData:

http://odata.netflix.com/Catalog/Titles

?$filter=Type eq 'Movie' and (Rating eq 'G' or Rating eq 'PG-13')

while this was the same API implemented by Netflix without the OData protocol:

http://api.netflix.com/movies?ratings=G,PG-13

there is no doubt the second one is easier to read, understand and maintain.

Conclusions

The only available implementation of OData that could have been advanced enough to meet the needs of NESTORE is the .NET one, maintained by Microsoft, but none of the partners have the knowhow in .NET to adopt it. OData is not widely adopted and has some design concern.

9.2 REST API Guideline

NESTORE propose to use JSON as exchange format and use UTF-8 Encoding everywhere in the data exchanged and any database.

Every time a unique identifier, ID, is needed we propose the use of string UUIDs or GUIDs for all of them, example: 577fc672-74b3-4009-a712-9d19e9d34062. Avoid incremental integers if possible to facilitate data migration and avoid ID collisions.

When a Date or a Timestamp is to be presented we should use an ISO8601 string with millisecond precision in the UTC time zone (with the Z at the end) to allow easier string-based date comparisons and ordering, example: 2018-04-06T16:57:13.243Z.

If the time zone information is important should be saved in a dedicated field using a string of the TZ Database standard from IANA (https://www.iana.org/time-zones), example: Europe/Bucharest, Europe/Rome, Europe/London; if the time zone is related to a country or a physical place avoid ambiguous time zones like GMT, BST and CET which are not constant in a given region (UK is on GMT during Winter and on BST during Summer).

Entities

Every entity in the system should have at least these fields:

- uuid: the ID of the entity
- created: the timestamp of the creation of the entity
- lastModified: the timestamp of the last update of the entity, can also be useful for HTTP caching

Otherwise the structure of the entity is free to be defined, example of a “puppy” entity:

```
{
  "uuid": "f9d2395e-1948-4093-b709-4a584def93e6",
  "name": "Fuffy",
  "breed": "dog",
  "gender": "male",
  "birth": "2017-08",
  "appearance": {
    "type": "spotted",
    "mainColor": "brown",
    "spotColor": "white",
    "spotsCount": 8
  },
  "created": "2018-04-06T16:57:13.243Z",
  "lastModified": "2018-04-06T16:57:13.243Z"
}
```

CRUD

CRUD (Create, Read, Update, Delete) APIs are the most common APIs and usually establish a common ground to build upon.

Follow the REST specification and use HTTP verbs and structure the API for CRUD and this section detail the structure of the services to be built for a hypothetical entity “Puppy”.

Base Endpoint

Each entity should have a base endpoint, in plural form, for our example:

/puppies

which should act as entry point for the entity.

Create Entity

API to create a new entity.

HTTP Verb / Path:

POST /puppies

The **Body** of the request contain the JSON object of the entity, in this case a puppy, the actual json structure and data are specific to the entity but should follow these basic set of rules:

- an **uuid** can optionally be provided, if not the service generates one on creation
- the **created** and **lastModified** attributes should be generated by the server but can be provided in the body, which is usually useful for import / data migration procedures

The Response status code should be

➔ 201 on success
➔ 409 if the entity already exist (same uuid)
➔ 4xx on client error, malformed json / missing parameters / unauthorized etc
➔ 5xx on server error

The Response headers should contain a **Location header** with the link to the resource just created.

The Response body of an error should contain a json detailing the issue for debugging purposes.

Read Entity

API to read a single entity.

HTTP Verb / Path

GET /puppies/${uuid}

The Body of the request should be empty. The body of the response should contain the full entity JSON.

The Response status code should be:

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 769643

➔ 200 on success
➔ 404 when the uuid doesn’t match any entity
➔ 4xx on client error
➔ 5xx on server error

The Response body of an error should contain a json detailing the issue for debugging purposes.

Update Entity

API to update an existing entity.

HTTP Verb / Path:

PUT /puppies/${uuid}

The Body of the request contain the JSON object of the entity, in this case a puppy, the actual json structure and data are specific to the entity but should follow these basic set of rules:

- the uuid must be provided and must match the service Path argument
- the created and lastModified attributes must both be provided untouched in respect of the last read; the server must verify they both match the stored entity data before changing the lastModified, if they don’t a conflict must be raised (409)

The Response status code should be

➔ 200 on success
➔ 4xx on client error, malformed json / missing parameters / unauthorized etc
➔ 5xx on server error

The Response body of an error should contain a json detailing the issue for debugging purposes.

Delete Entity

API to delete a single entity.

HTTP Verb / Path

DELETE /puppies/${uuid}

The Body of the request should be empty.

The Response status code should be

➔ 200 on success
➔ 404 when the uuid doesn’t match any entity
➔ 4xx on client error
➔ 5xx on server error

The Response body of an error should contain a json detailing the issue for debugging purposes

List Entity

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 769643

API to read all entities.

HTTP Verb / Path

GET /puppies

The Body of the request should be empty. The body of the response should contain the list of entities, see below the topic of Pagination and Encapsulation.

The Response status code should be

- ➔ 200 on success
- ➔ 4xx on client error
- ➔ 5xx on server error

The Response body of an error should contain a json detailing the issue for debugging purposes.

Partial Update (Patch) - Optional

API to partially update an existing entity.

HTTP Verb / Path:

PATCH /puppies/${uuid}

The Body of the request contain a JSON with a set of instructions to modify the entity. Each instruction should provide a way for the server to verify there are no conflicts (for example the update of a string can provide the old and the new string).

The Response status code should be

- ➔ 200 on success
- ➔ 4xx on client error, malformed json / missing parameters / unauthorized etc
- ➔ 5xx on server error

The Response body of an error should contain a json detailing the issue for debugging purposes.

The PATCH should be optional and implemented only when needed: for example, when the standard update could lead to too many conflicts.

Relationship

After CRUD the next big piece of a REST API are relationship. This section contain a proposal on how to handle relationships.

Relationship 1-N

Neosperience propose to manage relationships at API level as **sub-entities**. This can be an abstraction over the actual database.

Suppose two type of entities for which we already wrote a CRUD: “puppies” and “humans”. Both entities have their own CRUD unrelated to each other.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 769643

We want to create a relationship to know which human own the puppies. That's a 1 (human) - N (puppies) relationship cause a puppy cannot be owned by more than 2 humans.

Neosperience propose to model the relationship with API paths like these:

```
/humans/${uuid}/puppies
/puppies/${uuid}/ownership
```

specifically those two paths are two different ways to express the same relationship using a CRUD-like interface:

```
GET /humans/${uuid}/puppies
```

→ list of puppies uuid owned by the human along with any relationship attributes (if any), *optionally* it can **expand puppies** entities data along with the relationship properties

```
POST /humans/${uuid}/puppies
POST /puppies/${uuid}/ownership
```

→ add a relationship, in the body relationship attribute needed, as bare minimum it should containing `puppyUuid` and `humanUuid`, these two should be equivalent

```
PUT /humans/${uuid}/puppies/${puppyUuid}
PUT /puppies/${uuid}/ownership
```

→ edit the relationship attributes, if there are no relationship attributes this service is not needed, these two should be equivalent

```
DELETE /humans/${uuid}/puppies/${puppyUuid}
DELETE /puppies/${uuid}/ownership
```

→ delete a relationship, in this case remove the puppy ownership, these two should be equivalent

```
GET /humans/${uuid}/puppy/${puppyUuid}
GET /puppies/${uuid}/ownership
```

→read the relationship attributes of a particular puppy owned by the human, these two should be equivalent

This way of modeling the relationship makes it possible and easy to add relationship attributes that do not belong on any of the two entities, for example the date the human has received ownership of a puppy is not an human attribute nor a puppy attribute but an information of the relationship.

Another, more classic, example of a 1-N relationship is a product / category relationship where a product can only be of one category but a category can contains many products; a relationship attribute could be the ordering of the product in the category.

Relationship N-N

Again, let's think of two entities: puppies and toys. These time the relationship is bidirectional, a puppy can own multiple toys and a toy can be owned by multiple puppies.

The relationship *can* have its own properties and fields, for example we could store the *liking score* of the toy or whatever information that is not part of either the puppy or the toy but it is an information specific to the relationship.

Neosperience propose again to think of the relationship as a sub-entity with paths like these:

```
/puppies/${uuid}/toys
/puppies/${uuid}/toys/${toyUuid}
/toys/${uuid}/puppies
/toys/${uuid}/puppies/${puppyUuid}
```

This time the relationship is bidirectional and the API could look something like this:

```
GET /puppies/${uuid}/toys
```

→ list of objects of the relation and their attributes (if any), *optionally* it can **expand toys** entities data along with the relationship properties

```
GET /toys/${uuid}/puppies
```

→ list of objects of the relation and their attributes (if any), *optionally* it can **expand puppies** entities data along with the relationship properties

```
POST /puppies/${uuid}/toys/${toyUuid}
POST /toys/${uuid}/puppies/${puppyUuid}
```

→ add a relationship, in the body relationship attribute needed containing, as a bare minimum, puppyUuid and toyUuid

```
PUT /puppies/${uuid}/toys/${toyUuid}
PUT /toys/${uuid}/puppies/${puppyUuid}
```

→ edit the relationship attributes; if there are no attributes specific to the relationship this service is not needed

```
DELETE /puppies/${uuid}/toys/${uuid}
DELETE /toys/${uuid}/puppies/${puppyUuid}
```

→ delete a relationship

```
GET /puppies/${uuid}/toys/${uuid}
GET /toys/${uuid}/puppies/${puppyUuid}
```

→read the relationship attributes

The relationship can be equivalent in both directions.

Another, more classic example of an N-N relationship is the one between Teams and Players:

```
/teams/${uuid}/players ↔/players/${uuid}/teams
```

Authentication / Authorization

Every service of the NESTORE platform should require authorization to be executed.

We propose to adopt **OAuth 2.0** for every service: every HTTP request should require an **Authorization header** and verify it before being executed: an OAuth token should be mandatory to perform any request, the only exception being the OAuth2 service to obtain the token. These services could either provide a generic token for a non-logged in user or a specific one for a logged in user.

Services with no user data can require a generic authorization token. Services linked to an user account should instead require the logged in user token. Specific server-side only tokens could be implemented for server-side mechanism and help desk functionality.

Every service of the platform should adopt the same Authorization model with a single-sign-on architecture.

A status code 401 Unauthorized should be returned by any request performed without an Authorization header or with an invalid token (expired, invalidated or malformed).

A status code 403 Forbidden should instead be returned if the token is valid but do not have permission to perform an operation or access a resource. For example the token for user A shouldn't be allowed to read or modify resources of user B.

The authorization token could also be used to automatically filter resources as needed.

Every client must implement an automatic handling of OAuth2 token refresh upon receiving a 401 status code by obtaining a new token.

Request parameters (filtering, ordering, expansion, actions)

Sometimes common operation on data exposed by a CRUD services can be needed like

- **filtering** a list of entities

  Examples:
  ```
  GET /puppies?breed=cat,dog
  GET /puppies?owner=${humanUuid}
  GET /humans?withPuppiesOnly=true
  ```
- **ordering** a list of entities in a specific way

  Examples:
  ```
  GET /humans?orderBy=name asc,age desc
  GET /puppies?orderByNumberOfToyes=desc
  ```
- **expanding** a relationship to avoid multiple calls

  Examples:
  ```
  GET /humans/${uuid}?expandPuppies=true
  GET /puppies?expand=toys,owner
  ```
- **actions** with any particular effects on one or more entities and / or relationship

  Examples:
  ```
  POST /puppies/${uuid}/stealRandomToy
  ```
  → steal a random toy from another puppy of the same human etc...

These are all powerful and undeniably useful features in many occasions but can be expensive to implement and maintain. Specially combining them.

The NESTORE proposal is to implement these kind of features only when needed and in the way that makes more sense for the specific need and constraints.

Broadcast Changes

To ease up integration between services and different partners every CRUD and relationship services should broadcast a modification event on a channel. Every service interested in a particular change can listen to that channel and react to modification.

This makes it easy to add push or socket notifications of updates on entities and handle "cascade" modification in the platform. For example, if an entity is deleted all relationship using that entity could automatically update without the specific entity service having to manually perform the change.

Documentation

Every API should be documented but writing documentation is a tedious and time-consuming process. Following these guidelines lift much of the need of writing a documentation for each API but it is not enough.

The NESTORE proposal is to:

- Document JSON Exchange formats for entities, relationships, errors using JSON-SCHEMA, the same schema can also be used for validation inside the service with many libraries (http://json-schema.org/)

- Services API should be documented using SWAGGER (https://swagger.io/)

- Any peculiarity of the API that do not exactly follow the guidelines or it's not immediately obvious reading the API has to be documented with a short manually written documentation

- Along with the documentation POSTMAN export for each service should be exported and versioned with the documentation to make it easier for everyone to start experimenting with a service without writing a single line of code (https://www.getpostman.com/)

Pagination and Encapsulation

When a list of entities or relationship to entities can get too some kind of Pagination may be needed.

The proposal is simply to use a standard query string request format:

❏ `limit`: max number of item of a single page
❏ `offset`: offset from 0 to start the list from

The response body array should encapsulate some information like the `total` number of items along with a repetition of the `limit` and `offset` requested.

Encapsulation can also be needed on relationship. For example, showing the list of puppies uuids associated to a human could not be enough, encapsulating the full puppy associated with each uuid can be helpful to maintain relationship specific attributes external to the puppy entity.

Security

Every Service should run over secure SSL (TSL) in HTTPS.

The European GDPR regulation requires any platform managing sensible user data to do everything possible to protect that data.

For this reason, along with the common best practice for security we recommend:

➔ using encrypted databases
➔ never mix in the same table / database data that can lead to identify an user and sensible data (like health information), just use a reference to the user id

The GDPR also require every user to be able to require and obtain all the data the platform collected about him for download, this is easier to be done if every data for an user is associated to the user with the same field name “`userUuid`” and we keep a list of entities containing user data.

Finally, GDPR require a feature for which an user can ask for the deletion of all his data at any time, NESTORE recommend to keep a shared list among partners of all services containing user data to make the job of writing a script to purge an user from the database completely or collect all its data as easy as possible.

Caching and Compression

Clients can use the `If-Modified-Since` HTTP header with the *`LastModified`* caching http headers to avoid redownloading an entity JSON if it did not change. Since we required for each entity to have a *`lastModified`* attribute this should be easy to implement.

Another great and more powerful way to manage HTTP caching is the use of ETag or Weak ETag when possible using the *`If-None-Match`* HTTP header.

It could be a good idea to compute the weak ETag and return it along with each entity, even on lists, so that a smart client could use this to the max extent when optimizing content loading.

We also advice to enable gzip compression on every Service whenever possible to speed up performances.

I18n (multi-language)

NESTORE project will need to support the following languages: English, Italian, Spanish, Catalan.

Whenever a multi-language content is present in an entity all the localized fields should be included in a specific `content` field of type array. The array should contain a json object for each locale. Each object should always contain the locale as string (example: en, it, es, ca) and all the same fields.

```
{
  "uuid": "2e6e553f-a55c-4888-9353-d987f4d9fb54",
  "non-localized-field": "...",
  "content": [
    {
      "locale": "en",
      "title": "English title",
      "image": "https://cdn.domain.com/english-image.png"
    },
    {
      "locale": "it",
      "title": "Italian title",
      "image": "https://cdn.domain.com/italian-image.png"
    },
  ]
}
```

when a locale is not available the English locale should always be the fallback, and should be mandatory.

Beyond Classic REST API

If any client need should go beyond what we can provide following these guidelines we can evaluate to build an interface on top of them using either

- Facebook GraphQL - http://graphql.org/

- Netflix Falcor - https://netflix.github.io/falcor/

Both standards solve the same problem in a different way and are widely adopted.

They permit to abstract the REST API and let the client customize the query obtaining only portion the portion of the data needed or combining multiple service call into a single one.

Facebook GraphQL

A specification for querying data logically structured as a graph (looks more like a tree actually).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

It define a query language that is designed to look like the expected returned json.

```
{
  me {
    name
  }
}
```

is expected to return the information in a JSON like this:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

The idea is that "me" should be one of the entry points provided by the graph, it could be an object with many fields and the query here define the only parameter the client is interested in is the name.

The specification also defines parameters to be passed, queries, fragments for common parts and many other things.

The idea is to have a **single endpoint** for every request needed, clients can create their own request specifying exactly what data they want as response and how to structure them.

Server side a group of resolvers is given the job of resolving those properties like "`me`" in the example below or "`me.name`", with reasonable fallback when a resolver is not defined. They work as a key / map resolving plethora of functions that should connect to existing services, database or generate data on the fly. This way multiple services can be combined to answer a single request and build a single response.

This kind of architecture can be built in front of existing, regular services, to provide the clients with the ability to build the request in the way they prefer and need without client-side logic to chain request and assemble them.

GraphQL also support an optional mechanism called ***mutation*** that allow a somewhat limited support for writing data in the same fashion.

There are libraries in many languages, server side the suggested one is Node.js using a plugin for express.

Main concerns:

- HTTP Caching: One of the drawback is that it requires providing a query through POST to obtain the data, meaning that clients can't rely on http caching. There's an option to using GET with the query in query string but that doesn't solve the issue anyway. Caching have to be handled manually.
- GraphQL do not provide any guidelines on how to handle API versioning, the strong opinion they have is that if the client can specify the data so precisely evolving the API without breaking existing clients should be a lot easier and, in the worst case, require a custom resolver server side to provide support on older versions of the clients.
- No binary support for uploading or downloading.
- No control over the exact query clients are using, meaning the graph, once released, become the API contract and need to be maintained indefinitely.
- Requires a little bit of extra knowledge, but it is honestly quick to learn and well documented.

Netflix Falcor

A javascript library to map API services to a graph model.

The full distributed database of a set of REST APIs can be seen as a big graph of nodes connected with each other where nodes can be repeated in the graph.

The basic idea of Falcor is to create the infrastructure to mirror and sync that graph between client and server allowing the client to navigate and read / write the model as if it had the json locally and take care of the synchronization automatically.

To achieve that the JSON is being converted from a tree to a graph where every entity is being mapped and referenced with a symbolic link that is resolved internally by the library.

Server side a routing resolve every entities, accessing data maskerate a series of redirections between parts of the graph.

Caching is possible relying on HTTP cause in reality the calls are all GET and PUT / POST, which if REST is truly RESTful are idempotent operations.

Main concerns:

- Not as widely adopted as GraphQL.
- Requires a client side library to do the work of keeping the JSON Graph synced with remote data, Netflix briefly released a java version of the client library which has later been taken down; only javascript is officially supported.
- Server side only Javascript is officially supported.

There's no obvious way to dynamically query the Graph Model.

Comparison

Both GraphQL and Falcor try to solve the same problem: decoupling client and server giving the client the flexibility it needs without the need of the server to implement complex and costly to maintain custom filtering, aggregations, ordering.

They both put the data in the center describing it as some kind of Graph but the way the two tackle the issue is very different: while falcor try to keep in sync a partial graph between server and client using libraries and internal redirections graphQL resolve the redirections server side and return objects leaving to the client the job of figuring out how to handle caching.

From the performance perspective Falcor is faster out of the box because of the build-in caching and it is easier to pick up and start using but it has strong limitations, the biggest one being reliant on a client-side library which is not provided for other languages then javascript. This automatically rule out the option for non-web clients.

On the other side GraphQL is a bit harder to pick up and requires more thinking to set up properly. It leaves the caching management to the clients by just providing a generic guide line on using uuids for entities to build a local cache. But it doesn't really need a client library and have many more options as languages and technology for implementing it server side.

The GraphQL specification itself is superior to Falcor because it has build-in typing and introspection, which allow for automatic tooling if needed and gives a little more flexibility to the developer both client and server side. It is easier to think of GraphQL as a standard, while Falcor is more a library or a tool.

In the end it comes down to comparing the two big issues:

- Falcor has no multiplatform support

- GraphQL strips away HTTP Caching from clients and requires them to handle it manually

Aside from other differences I think these two are the main one to consider when picking a technology.

9.3 LogMeal API

LogMeal is an API (Application Programming Interface) developed by engineers and researchers from the University of Barcelona, aimed at satisfying various needs directly related to the automatic analysis of foods, both small and large companies and people for individual use from an image.

More technical details and examples:

- LogMeal's API documentation

- Updated version.

Company Sign Up

First of all, sign up your company sending your company name and email. This has to be done once only. A token will be generated and sent to your email (cToken).

The url for accessing this service is http://www.logmeal.ml:8088/companySignUp

Requirements:

- POST request
- Headers:

```
{'Content-type':'application/json'}
```

- Parameters sent in data, in json format :

```
{'name':your_company_name, 'email':your_company_email}
```

Parser Response

HTTP Status Code	Description
200	OK, company user created, token generated and sent by email
210	Company already signed up with generated token. Call /forgotCToken to recover it
211	Not a valid email
212	Company name already used
213	Company email already used
400	Invalid syntax
404	The specified URL was not found or couldn't be retrieved

User Sign Up

With the cToken generated you can add users to your company. This user will have access to image recognition information. A token for this user will be generated and returned in response, let's call it uToken.

Url is http://www.logmeal.ml:8088/userSignUp

Requirements:

- POST request
- Headers:
```
{'Content-type':'application/json', 'Authorization':'Bearer '+cToken}
```
- Parameters sent in data, in json format :
```
{'username':your_username}
```

Parser Response

HTTP Status Code	Description
200	cToken is valid, company user is created, token is generated and sent in response (JSON format)
210	Username already exists in LogMeal's database
400	Invalid syntax
404	The specified URL was not found or couldn't be retrieved

If Status Code is 200, uToken is returned in response. This uToken must be saved and used in next API calls.

Get information

Updating returned information, adding an image id and dish type ('food', 'non_food', 'drinks' or 'ingredients') and removing the tag 'ingredients'. Recipe will be returned in another method.

The food type 'ingredients' correspond to raw food (like 'carrot', 'pepper', etc). If the dish type recognized is 'no food', the method will not return more information about recognition.

If the dish type recognized is 'food', method will return the top 3 food families (in previous version it returns all food family probabilities). The list of possible food families is : noodles_pasta, soup, vegetable_fruit, seafood, egg, meat, rice, dessert, fried_food, bread, dairy_products.

Response example

If dish type is food

```
print response.json()
{'imageId':4, 'dishType':'food', 'foodFamily': {'tops': ['noodles_pasta',
'soup', 'vegetable_fruit'], 'probs': [0.9995937943458557,
0.00037020735908299685, 2.5715842639328912e-05]}, 'foodRecognition':
```

```
{'tops': ['spaghetti_bolognese', 'curry_noodles_with_mushrooms',
'pasta_with_pesto', 'ratatouille', 'celery_salad'], 'probs':
[0.971264660358429, 0.023766646161675453, 0.0023865797556936740,
0.0013138295616954565, 0.00037995769525878l3]}}
```

If dish type is drinks

```
print response.json()
{'imageId':4, 'dishType':'drinks', 'drinksRecognition': {'tops':
['coffee', 'cava', 'beer', 'sangria', 'cider'], 'probs':
[0.971264660358429, 0.023766646161675453, 0.002386579755693674,
0.0013138295616954565, 0.00037995769525878l3]}}
```

If dish type is ingredients

```
print response.json()
{'imageId':4, 'dishType':'ingredients', 'ingredientsRecognition': {'tops':
['broccoli', 'cucumber', 'lettuce', 'apple', 'green peas'], 'probs':
[0.971264660358429, 0.023766646161675453, 0.002386579755693674,
0.0013138295616954565, 0.00037995769525878l3]}}
```

If dish type is non_food

```
print response.json()
```

{'imageId':4, 'dishType':'non_food'}

9.4 ZivaCare

In this annex are presented details relating zivacare.

Latest documentation can be accessed to https://docs.zivacare.com

SyncEngine

As part of the ZivaCare system, SyncEngine is responsible with the synchronization of user's data between ZivaCare and various third-party data providers. The synchronization is done by invoking agents (which are provider specific), using a RESTful Web Service interface, based on a specific and configurable schedule.

On each run, it extracts a list of users from the ZivaCare database and runs the synchronization for each of them. The synchronization consists in calling the corresponding agent for each external provider the user has granted access to ZivaCare. The call is done using a REST API implemented by the agent (with the proper arguments extracted from database). It is the agent's responsibility to decide if something must be done (synchronization is needed) or the existing information is up to date. The agent will have to notify the SyncEngine about its status and operations. If the SyncEngine detects some incidents (agents are not performing well or are constantly raising errors), a notification needs to be sent to the operators for further investigation.

The main system components are illustrated in Figure 42 Component Diagram for ZivaCare SyncEngine System.

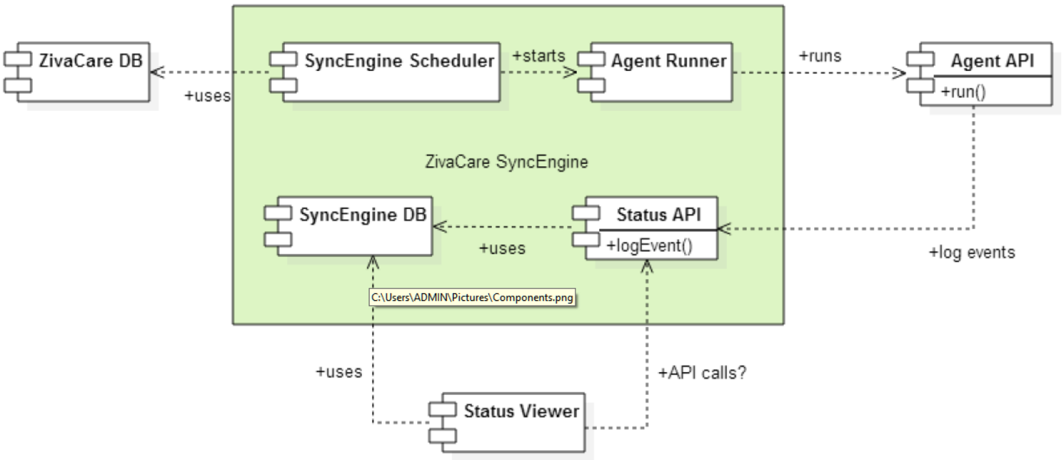


Figure 42 Component Diagram for ZivaCare SyncEngine System

The SyncEngine Scheduler is responsible with scheduling synchronization activities with a frequency set by the operator. At the appropriate moments in time, synchronizations are performed. Obviously a single synchronization (for a given user and data source) can run at any time. A synchronization process consists in the following steps:

- Extract users from the ZivaCare DB.
- For each user obtain a list of external providers that the engine is authorized to synchronize.
- For each provider start an Agent Runner (which issues a REST request to the external agent). The runners can be started in parallel (because they are addressing different providers).

- Wait for all requests to be complete (successful or with errors, doesn't matter).

- Process next user in the list.

The Agent Runner receives the parameters (keys/credentials) to invoke the agent performing the synchronization. The invoking is done using a REST API. The runner is then waiting for the answer from the agent (either a successful result or an error).

Status API is a very simple REST API exposing a log resource to be used by an external agent. The agent should log at least start, stop and result status.

SyncEngine DB is a local database for storing system status (runs, agent events, current operations, history results etc). It's directly used by the Status API to record agents' events and by scheduler and agent runner to record their status.

Status Viewer is a simple web page showing the status of the system.

Authorization

ZivaCare provides a single authentication point and it operates with the OAuth2 authorization protocol. The implemented OAuth2 mechanism is presented in Figure 43 The ZivaCare authorization process.

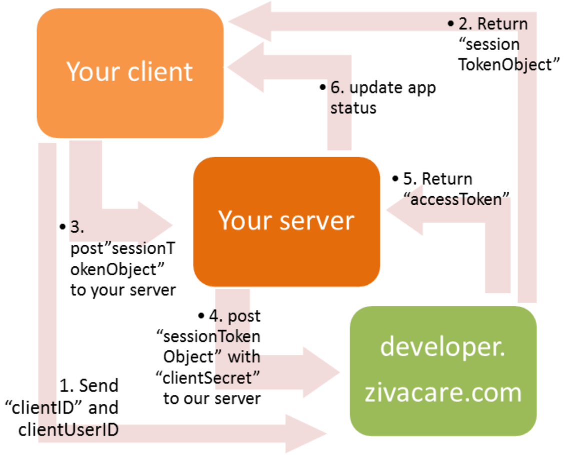

Your client
• 2. Return "session TokenObject"
• 6. update app status
Your server
• 5. Return "accessToken"
• 3. post"sessionTokenObject" to your server
• 4. post "sessionToken Object" with "clientSecret" to our server
developer. zivacare.com
1. Send "clientID" and clientUserID

Figure 43 The ZivaCare authorization process

The ZivaCare authorization mechanism consists of the following steps:

1. Client will launch the Connect popup with two configuration parameters: clientId and clientUserId.

2. Once a user pressed the "Finish Connecting Data" button you will receive a sessionTokenObject.

3. You will post the" SessionToken Object" to your server application.

4. You will add your unique clientSecret to the "session TokenObject" and POST it to Ziva API.

5. Ziva API will return accessToken and publicToken.

With such unique access token, an application registered in the ZivaCare system can access a user's health data on her/his behalf.

9.5 NESTORE End User Portal

The NESTORE Web Portal for End Users will be developed using Liferay (Liferay, 2018). This is an open source Social Content Management System. Its key features are: personalized online experience, social collaboration, content management and responsive design. They are all valuable for the NESTORE platform. The offered web content can be personalized at user level, and the users can be part of social communities where they can collaborate, share opinions and documents. Moreover, Liferay web sites are mobile-ready.

The Liferay architecture is presented in Figure 44 Liferay architecture. It is organized on three layers: persistence, services and frontend.

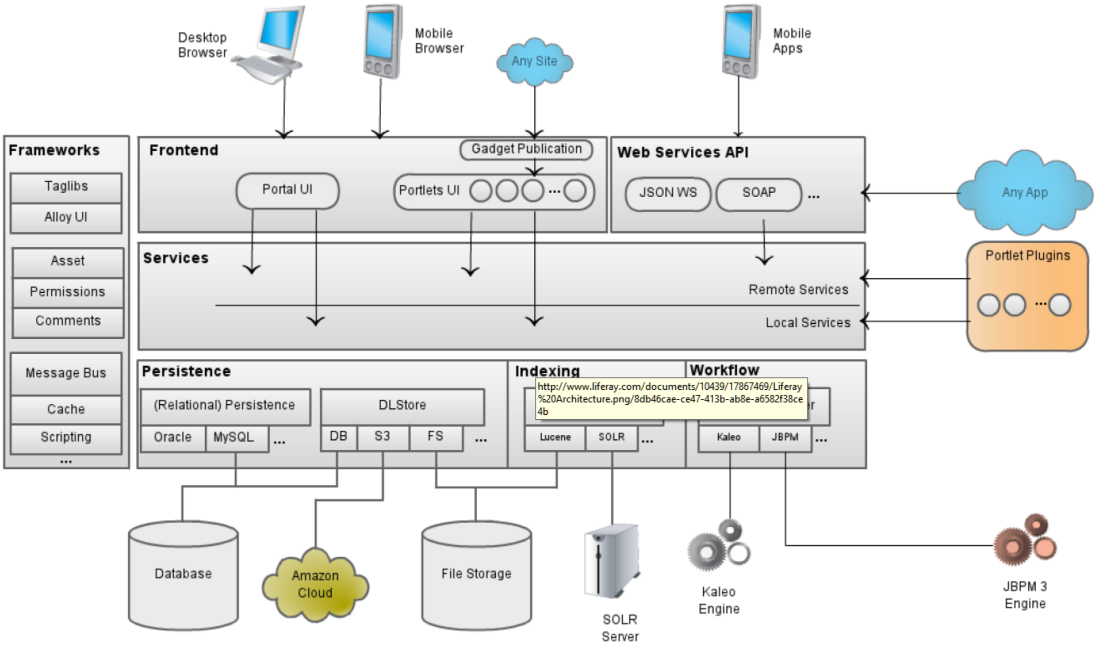


Figure 44 Liferay architecture

Liferay aggregates a lot of frameworks, which means it offers a lot of functionality. At the persistence level, virtually any Relational Database Management System is supported. The services layer holds the business logic. The services can be local or remote, secured or unsecured (typically the first is used only internally), and they are of two types: portal and portlet. Portal services allow the management of organizations, users, sites, pages, documents, workflows, etc. A portlet is a pluggable user interface software component. Through portlets the Liferay Portal can be extended and customized. NESTORE specific functionalities will be developed using this Liferay framework extension point. Any portlet can also expose services, like any other Liferay existing module. The services can be exposed as SOAP or REST web services. This is important because it allows the development of NESTORE mobile applications that can use them.

The portlets User Interface can be integrated in the Liferay frontend. The Graphical User Interface can be developed with HTML, CSS, JavaScript, JSP, JSF, Velocity and Vaadin. Liferay also supports themes.

The users will be able to interact with all NESTORE services. For example, they will be able to view their health data, acquired from NESTORE and/or third-party sensors and devices.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

9.6 NoSQL Investigation

NoSQL, meaning "not only SQL", is a movement encouraging developers and business people to open their minds and consider new possibilities beyond the classic relational approach to data persistence. But a novel data modelling technique but it does not provide sufficient justification for replacing a well-established and well understood data platform; there must also be an immediate and very significant practical benefit. This motivation must be searched in the form of a set of use cases and data patterns whose performance improves by one or more orders of magnitude when implemented in one of the considered NoSQL databases. One compelling reason for choosing a graph database is the performance increase when dealing with connected data versus relational databases, where join-intensive query performance deteriorates as the dataset gets bigger. With a graph database performance tends to remain relatively constant, even as the dataset grows. This is because queries are localized to a portion of the graph. As a result, the execution time for each query is proportional only to the size of the part of the graph traversed to satisfy that query, rather than the size of the overall graph. Another reason is the model's flexibility, to avoid modelling our domain in exhaustive detail ahead of time. NoSQL database are mainly schema-less and can be though as additive, meaning we can add new kinds of relationships, new columns, new nodes, new labels, and new subgraphs to an existing structure without disturbing existing queries and application functionality.

MONGODB

MongoDB is a document database, it is ranked #4 as general database and #1 in the document database category. It is a mature database [see Figure 45 Magic Quadrant for Operational Database Management Systems - Source: Gartner (October 12, 2015)], with a commercial company behind its development, also offering commercial 24x7 paid support. The drawbacks are in the relationships with other documents (the suggested method is to manually manage them) and the storage of large binary data (the maximum document size is 16 MB, for file storage it is suggested to use an additional layer called GridFS that splits the data in small chunks). It has strong support for the major programming languages.

NEO4J

Neo4j is a graph database that give focus on relationships. Its data model is based on nodes and relationships, and both can have properties. It is not well suited to represent other kind of data, and it is often used in together with other databases. It has strong support for Java programming language (can be also embedded into an application) or can be accessed by REST services. It is available both in open source version and commercially licensed version with some additional feature, but it has no support plan.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

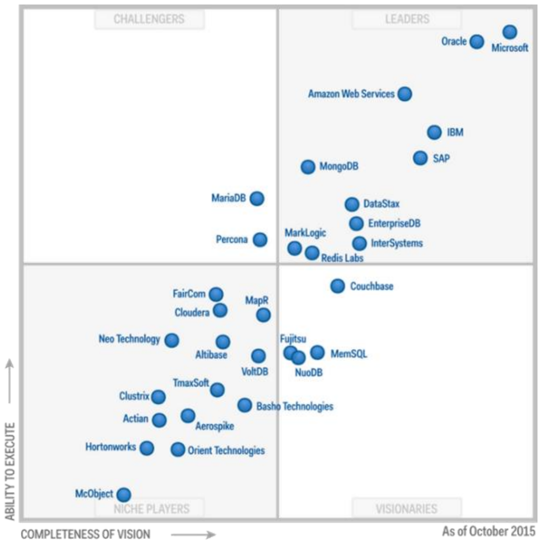


Figure 45 Magic Quadrant for Operational Database Management Systems - Source: Gartner (October 12, 2015)

ORIENTDB

It is a multi-model database, meaning it support Graph, Document, Key/Value, and Object models. Supporting both embedded documents and true relationships (not just manual references like MongoDB), it provides a flexible and powerful support to model connected data. It supports SQL and REST service. It can be embedded in Java applications, but it also supports deployment in cloud environment (Multi-Master + Sharded architecture). It is available both in open source version and commercially licensed version with paid 24x7 support service.

APACHE CASSANDRA

It is a wide column database with emphasis on its distributed architecture, providing high availability on commodity servers with no single point of failure. It was initially developed at Facebook and from February 2010 it is a top-level Apache project. There is a commercially supported version maintained by Datastax with 24x7 support, certified software updates and hot fixes.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

9.7 NESTORE WoT Agent

WoT: the Web of Things (WoT) is a term used to describe approaches, software architectural styles and programming patterns that allow real-world objects to be part of the World Wide Web. Similarly to what the Web (Application Layer) is to the Internet (Network Layer), the Web of Things provides an Application Layer that simplifies the creation of Internet of Things applications.

Deployment of the IoT can be realized around the wotAgent (Web of Things Agent) that can be implemented at different levels.(Figure 20 IoT deployment using wotAgent)

- On the private cloud – then sensors must be technically able to send data to the cloud wotAgent
- At the level of sensors network – then the wotAgent will be implemented on a local gateway device
- On mobile device – the mobile device is responsible to acquire data from other specialized sensors or devices and the mobile wotAgent is transferring data to the cloud

The Mobile wotAgent will run on the mobile device inside a standalone mobile App composed of these layers (Figure 46 Mobile wotAgent App Layers):

- UI Layer: not strictly part of the wotAgent will be the human interface for the Nestore User, a thin simple interface allowing the user to login to the Nestore Cloud, register sensors and manage the synchronization preferences (how often, when etc.).
- Cloud Sync and Authentication Layer: the core of the wotAgent will be responsible of writing sensors data to the Nestore Cloud and authenticate with it to register each sensors; this layer will also operate the Sensors Interface.
- Sensors Interface Layer: or sensors driver, will be responsible for establishing the bluetooth communication with sensors and fetching the data.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

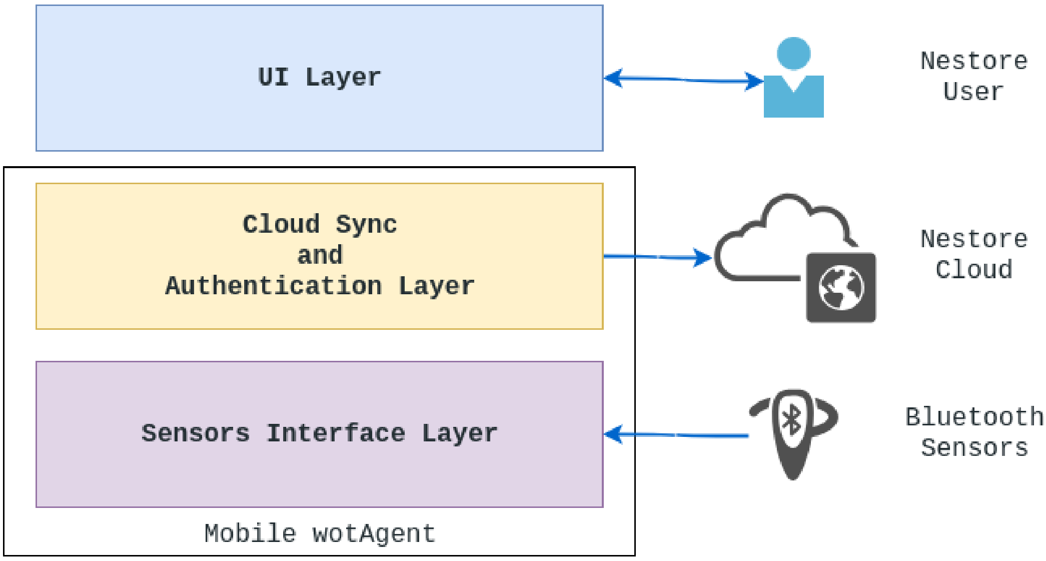


Figure 46 Mobile wotAgent App Layers

The UI Layer will be a thin layer allowing the user to operate the wotAgent and it will not be described in this document.

Due to the requirements of the hardware communications this App and all its layers will be developed in the native code of the platform.

Sensors Interface Layer

The sensors will be collecting data inside an internal, limited memory that will have to be regularly collected through bluetooth by the wotAgent.

The Sensors Interface Layer will manage this communication with sensors devices abstracting the process to the upper layer through a Sensor Driver SDK. This will be a passive layer: it will expose a set of operations and wait for the upper layer to trigger them:

- Scan: search for sensors devices in range and obtain some basic information on them (ex. the device identifier).
- Status: obtain status information from the device (ex. battery levels, current setup, etc...).
- Setup: send configuration information to a sensor (ex. gender, weight, height of the user).
- Fetch: obtain the data currently hold by a sensor and convert it into a native data structure.
- Purge: free up the internal memory of the device, usually performed after a successful fetch.

Cloud Sync

The Nestore Cloud REST services will be used by the Cloud layer to push sensors data to the Cloud and to get Authorization tokens to associate each sensor data to a Nestore User. There are, in fact, two process performed by this layer: a classical auth/login flow and the data synchronization.

This layer will be an active one with scheduled background processes that, after a proper initial setup, will operate even without the user interaction: the user will have to login to the Nestore Cloud, a token will be generated and will be used for all successive HTTPS requests allowing the Cloud to correctly associate each to the user.

A sensor registration process will be the next step of the setup: involving a SCAN for sensors and bluetooth pairing with them for the user, eventually associating a name to it. Each sensors will then have to be configured (SETUP) with the needed information that could be provided by the user directly through the UI or obtained from the Cloud User Details as needed.

The final step of the setup for the wotAgent will be scheduling the synchronization interval with the sensors: these will require to define how often or when to wake up the device and collect it's data. It will also be possible to trigger the synchronization manually.

During the synchronization the Sensors Layer will be operated, the data collected and pushed / written to the cloud using REST Services API: during this phase a conversion between the native data structure produced by each sensor will be converted into JSON format understood by the NestoreCloud and sent through a secure HTTPS connection.

Real time vs Batch synchronization

The optimal solution from the Nestore Coach perspective would be to receive all the sensors data in real-time and immediately push it to the cloud.

However, all the operations performed by this layer are constrained by the device limit in battery power and network resources and a real time synchronization would quickly drain the mobile device battery causing a bad user experience for the user and ultimately completely stop sending data to the Cloud.

Moreover Sensors devices may go offline unexpectedly for various reasons: out of battery, out of range, device malfunction. All these conditions calls for a tolerance in the process of collecting and synchronizing data to the cloud.

It is imperative for the wotAgent to look and find for a compromise in battery consumption and data freshness along with reliability and fault tolerance. For this reason all the data will be synchronized in batches from the sensors instead of continuously.

To allow the Nestore user to monitor the situation the wotAgent will expose to the UI Layer the state of the synchronization, roughly exposing these states:

- IDLE: no synchronization is going on
- READING: collecting data from the sensor
- WAITING_NETWORK: device is ready locally, waiting for network to sync it in the cloud
- SYNCING: writing data to the cloud
- ERROR_DEVICE: communication with device failed
- ERROR_CLOUD: communication with the cloud failed

Other states or more precise informations that we can’t anticipate right now may be also exposed to the UI.

9.8 Log System

The Logging section for the Nestore Project is implemented in PostgreSQL with TimescaleDB. It consists of an API developed with PostgREST. The following is a step by step documentation on how to use it.

The log payload is represented as follow:

```
{
"userid": "123456",
"applicationName": "logMeal",
"applicationUID": "ID to identify the thread or specific instance",
“applicationContext”: “JSON like data from the application”,
“applicationAction”: “exception/event/state/http”,
“applicationActionDetails”: “JSON with Action Details i.e. API call”
"actionHttpMethod": "GET/POST/PUT/DELETE",
"createdate": "2018-09-05T11:34:01.785474+03:00"
}
```

Authentication

For security reasons, the API has several clearance levels (roles with permissions). Before the actual application logs can be registered and/or retrieved, you must login with a user and password to obtain an Access Token generated by the log system. This Token will be used afterwards for a GET and/or POST operation on the logs entity.

The following is an example of a request and response for the Authentication process:

Request
endpoint: /rpc/login
body:

```
        {
                "email": "whatever@test.user",
                "pass": "ThisIsAPassword"
        }
```

Response body:

```
        [{
                "token": "thisisanauthtoken"
        }]
```

Save Logs

To store the log data, there are 2 POST methods:

Save with params
request endpoint: /rpc/p_logs_params
request body:

```
        {
“Payload”
        }
```

Save with JSON

request endpoint: /rpc/p_logs
request body:

```
[{
"logs_line": { payload }
}]
```

On both these requests you must set a Header with the Token.
Key: Authorization
Value: Bearer thisisanauthtoken

Retrieve Logs

To get a set of logs, there is only one endpoint that can be called. Below you can see a simple request and response. After this example, you can find a list of specific operators used for more complex queries.

request endpoint: /v_logs
response:

```
{
Array of Payload
}
```

On both these requests you must set a Header with the Token.

Key: Authorization
Value: Bearer thisisanauthtoken

Postgrest filtering

Commands to use after API call for Horizontal filtering:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Abbreviation	Meaning	PostgreSQL Equivalent
eq	equals	=
gt	greater than	>
gte	greater than or equal	>=
lt	less than	<
lte	less than or equal	<=
neq	not equal	<> or !=
like	LIKE operator (use * in place of %)	LIKE
ilike	ILIKE operator (use * in place of %)	ILIKE
in	one of a list of values e.g. ?a=in.(1,2,3) – also supports commas in quoted strings like ?a=in.("hi,there","yes,you")	IN
is	checking for exact equality (null,true,false)	IS
fts	Full-Text Search using to_tsquery	@@
plfts	Full-Text Search using plainto_tsquery	@@
phfts	Full-Text Search using phraseto_tsquery	@@
cs	contains e.g. ?tags=cs.{example, new}	@>
cd	contained in e.g. ?values=cd.{1,2,3}	<@
ov	overlap (have points in common), e.g. ?period=ov.[2017-01-01,2017-06-30]	&&
sl	strictly left of, e.g. ?range=sl.(1,10)	<<
sr	strictly right of	>>
nxr	does not extend to the right of, e.g. ?range=nxr.(1,10)	&<
nxl	does not extend to the left of	&>
adj	is adjacent to, e.g. ?range=adj.(1,10)	-|-
not	negates another operator, see below	

Application can be filter result rows by adding conditions on columns, each condition a query string parameter. For instance, to return people aged under 13 years old:
GET /people?age=lt.13 HTTP/1.1

Multiple parameters can be logically conjoined by:
GET /people?age=gte.18&student=is.true HTTP/1.1

Multiple parameters can be logically disjoined by:
GET /people?or=(age.gte.14,age.lte.18) HTTP/1.1

Complex logic can also be applied:
GET /people?and=(grade.gte.90,student.is.true,or(age.gte.14,age.is.null)) HTTP/1.1