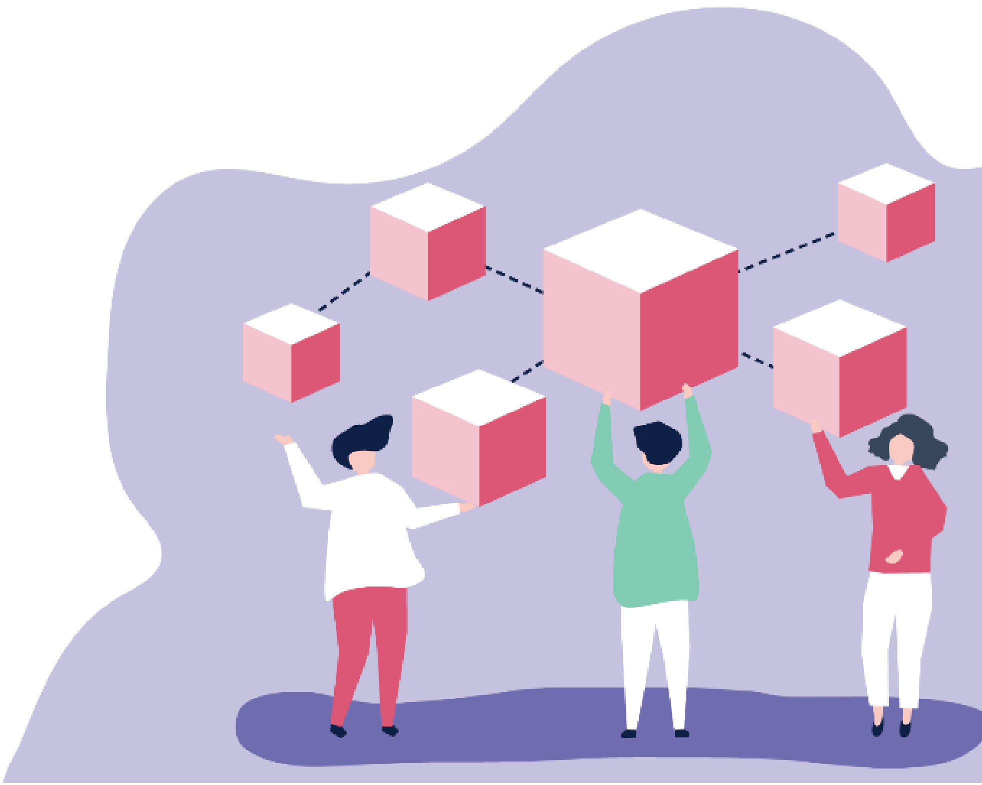
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 857223

GATE KEEPER

D4.2 Thing Management System

Deliverable No.	D4.2	**Due Date**	30/09/2020
Description	Platform cluster implementation and integration		
Type	Other	**Dissemination Level**	PU
Work Package No.	WP4	**Work Package Title**	GATEKEEPER Things Management Infrastructure & Development
Version	1.0	**Status**	Final



Authors

Name and surname	Partner name	e-mail
Eugenio Gaeta	UPM	eugenio.gaeta@lst.tfo.upm.es
Álvaro Belmar	UPM	abelmar@lst.tfo.upm.es
Domenico Martino	ENG	Domenico.Martino@eng.it
Alejandro Medrano	UPM	amedrano@lst.tfo.upm.es
Jose Javier Serrano	UPM	jserrano@lst.tfo.upm.es

History

Date	Version	Change
28/07/2020	0.1	Initial draft
31/08/2020	0.2	Working Version
30/09/2020	1.0	Final Version

Key data

Keywords	Gatekeeper, Web of Things, Digital Twin, Thing Description, FHIR, JSON-LD
Lead Editor	Eugenio Gaeta (UPM)
Internal Reviewer(s)	Imad Ahmed (Medisantè), Eleni Georga (UoL)

Abstract

This deliverable reports on the progress of T4.2 with the main goal of designing the GATEKEEPER Thing Management System for providing discoverability and access to the components of the GATEKEEPER platform following the Web of Thing standard (T3.3).

The deliverable describes the initial version of the Thing Management System that is used to build the first version of the Gatekeeper platform that includes the Thing Management System as entry point of the API infrastructure, the Gatekeeper Trust Authority (T4.5) that provides authentication and authorization for the usage and the Gatekeeper API and the Data Federation Framework (T4.4) that provides the early version of the Gatekeeper Healthcare semantic model (T3.4 and T3.5).

This deliverable is a live document that will be updated in a second version by M24.

Statement of originality

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

Table of contents

Abbreviations

Table 1: List of abbreviations

API	Application Programming Interface
BSON	Binary JSON
GTA	Gatekeeper Trust Authority
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
RDBMS	Relational Database Management System
SAREF	Smart Appliances REFerence (SAREF) ontology
SOAP	Simple Object Access Protocol
SPA	Single Page Application
SSL	Secure Sockets Layer
TD	Thing Description
TMS	Thing Management System
WoT	Web of Things
WWW	World Wide Web
XML	Extensible Markup Language

List of tables

List of figures

1 Introduction

This deliverable describes the Thing Management System (TMS) that is one of the core components of the Gatekeeper platform. It is based on the experience that UPM acquired working on a Web of Things (WoT) home gateway (described in the deliverable D3.4) that is used in the Smart House Living Lab[1] at UPM facilities.

This gateway provides standard interoperability and secure communication based on WoT Description, SAREF ontology and OAuth2.0 to everyone that wants to interact with the devices of the Smart Home Living lab.

The TMS is a similar component that extends these features (interoperability, security, ontologies) on the healthcare domain considering the semantic models not only devices but also services, data, and platforms. The idea behind the TMS is to provide access to general assets of the Gatekeeper platform by using the standard WoT and ontologies that can be used to describe in a standard way device, data and any other assets.

Furthermore, through the interaction of the TMS and the Gatekeeper Trust Authority we are going to add a fundamental functionality that is the certification of a component that is to join the Gatekeeper platform.

More in detail this deliverable describes the concept behind the Thing Management System and how it is positioned within the Gatekeeper architecture (chapter 2), the technologies used for its development and how it is implemented with different microservices that interact with each other (chapter 3), the procedures for the integration of general components, GTA and data federation framework (chapter 4) and how it is deployed in development and production environments on a cloud infrastructure that grant isolation, scalability and resilience.

[1] Smart House Living Lab, https://www.lst.tfo.upm.es/smart-house/ , Last access September 2020

2 Thing Management System Version 1

2.1 Position of the TMS into Gatekeeper architecture

The Thing Management System (TMS) is one of the core components of the Gatekeeper platform described into deliverable D3.2. It is focused on the management of the things that belong to the GATEKEEPER platform, providing an entry point of the platform where all the Gatekeeper things are registered.

It can be seen as an API gateway harmonized with a broker service, that provides description of the Gatekeeper things with Things Description (TD) as described by the standard WoT defined by the W3C[2]. The WoT exposes sensors, actuators and related services as software objects independently of the underlying protocols and data formats, as a means to counter the fragmentation of the IoT. JSON-LD is used for machine interpretable descriptions of things in terms of the object properties, actions and events, and associated metadata for semantics, communication and security.

The broker service is based on an intermediary architecture that is a WoT architecture component implemented by a servient.

A servient, in the field of WoT terminology, is the software stack that implements the WoT building blocks. A Servient can host and expose Things and/or host Consumers that consume Things. Servients can support multiple Protocol Bindings to enable interaction with different IoT platforms.

An Intermediary is located between a Thing and its Consumers, performing the roles of both a Consumer (to the Thing) and a Thing (to the Consumers). In an Intermediary, a Servient software stack contains the representations of both a Consumer (Consumed Thing) and a Thing (Exposed Thing)[3]. The concept of intermediary is shown in Figure 1.

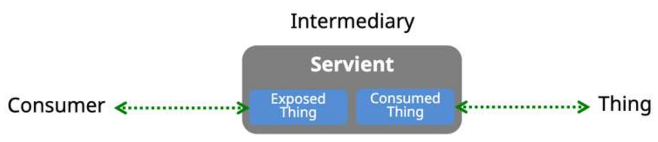


Figure 1 – Servient as an Intermediary

The TMS interacts with the Gatekeeper Trust Authority (GTA) T4.5 for authentication and authorization of platform users in order to provide the necessary rights for Thing consumption.

In this first release of the Gatekeeper platform there is an integration among TMS, Gatekeeper Trust Authority (GTA) and Data Federation Framework.

[2] Web of Things, https://www.w3.org/TR/wot-thing-description/ , Last access September 2020

[3] Intermediary Web of Thing Architecture, https://www.w3.org/TR/wot-architecture/ , Last access September 2020

Referring to the figure of the Gatekeeper architecture shown in the deliverable D3.2 (Figure 2), the TMS is the entry point of the Gatekeeper platform. It provides:

- Descriptions of the Gatekeeper things (data, devices, services, assets, etc..). These descriptions, based on the standard WoT, will be provided though a broker service.
- Access to the Gatekeeper things and/or federated external things. Through an API gateway the endpoints associated to every registered TD will be accessible.
- Access to the Gatekeeper Data Space. The main TD available through the TMS will be the Data Federation Framework that defines the common semantic model for the healthcare data space within the Gatekeeper platform.
- Integration with the Authentication, Authorization and Auditing system. The TMS manages access and security of the Gatekeeper platform interacting with the services provided by the GTA.

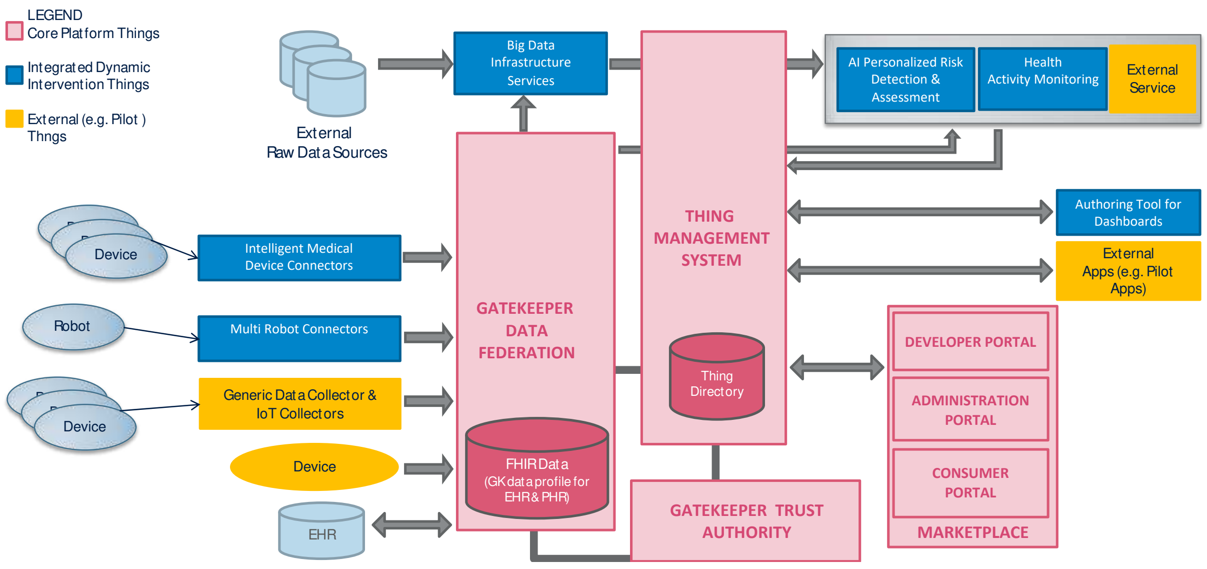

LEGEND
Core Platform Things
Integrated Dynamic Intervention Things
External (e.g. Pilot) Thngs
External Raw Data Sources
Big Data Infrastructure Services
THING MANAGEMENT SYSTEM
Thing Directory
AI Personalized Risk Detection & Assessment
Health Activity Monitoring
External Service
Authoring Tool for Dashboards
External Apps (e.g. Pilot Apps)
Device
Intelligent Medical Device Connectors
Robot
Multi Robot Connectors
Device
Generic Data Collector & IoT Collectors
Device
EHR
GATEKEEPER DATA FEDERATION
FHIR Data (GK data profile for EHR & PHR)
GATEKEEPER TRUST AUTHORITY
DEVELOPER PORTAL
ADMINISTRATION PORTAL
CONSUMER PORTAL
MARKETPLACE

Figure 2 – Gatekeeper architecture

2.2 TMS inner architecture

Figure 3 presents the internal architecture of the TMS and the interaction foreseen with the GTA.

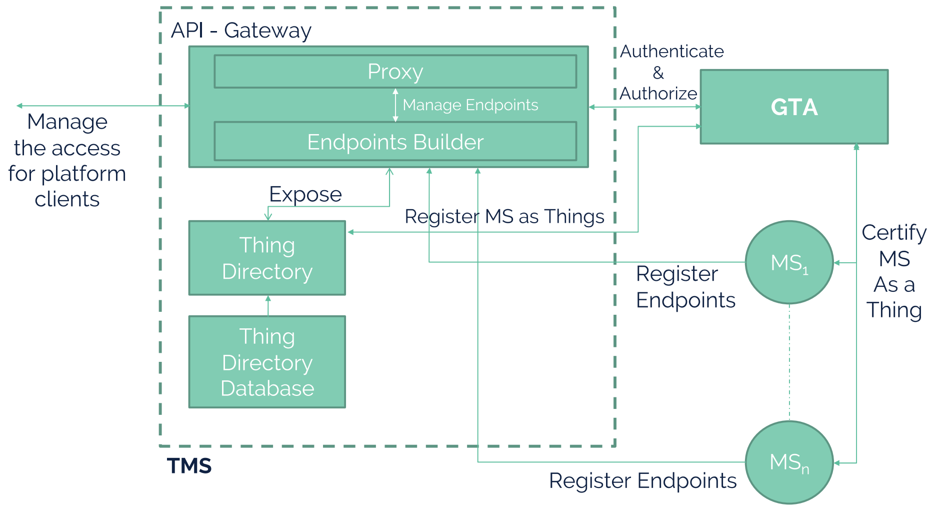


Figure 3 – TMS architecture

Several microservices are integrated together in order to provide the TMS functionalities. They also provide interaction with other external components.

The TMS consists of the following internal microservices:

- The API gateway representing the single point of access of the entire Gatekeeper platform.
- The Thing Directory service representing the broker service that provides the description of Gatekeeper thing in the WoT format.
- The Thing Directory database representing the database that stores the necessary data in order to build and provide the TDs of the Gatekeeper assets.

2.2.1 Thing directory database

TD describes the metadata and interfaces of Things, where a Thing is an abstraction of a physical or virtual entity that provides interactions to and participates in the WoT. TDs provide a set of interactions based on a small vocabulary that makes possible both to integrate diverse devices and to allow diverse applications to interoperate. TDs, by default, are encoded in JSON format that also allows JSON-LD processing. The latter provides a powerful foundation to represent knowledge about Things in a machine-understandable way. A TD instance can be hosted by the Thing itself or hosted externally when a Thing has resource restrictions (e.g., limited memory space) or when a WoT-compatible legacy device is retrofitted with a TD.

The WoT TD is a central building block in the W3C WoT and can be considered as the entry point of a Thing (much like the index.html of a Web site). A TD instance has four main components: textual metadata about the Thing itself, a set of Interaction Affordances that indicate how the Thing can be used, schemas for the data exchanged with the Thing for machine-understandability, and, finally, Web links to express any formal or informal relationship to other Things or documents on the Web.

TDs are formatted as JSON documents extended with JSON-LD contexts. Within the TMS such documents are served by the Thing Directory. The Thing Directory reads and writes TDs in a database, such database is the Thing Directory database.

Due to the nature of the TD the most efficient way for storing and retrieving TDs is using document-oriented databases. These databases, also known as document stores, are used to manage semi-structured data. This data does not adhere to a fixed structure but it forms its own structure. The information can be ordered using markers within the semi-structured data. Due to the lack of a defined structure, this data is not suitable for relational databases since its information cannot be arranged in tables.

Document databases focus on storage and access methods optimized for documents as opposed to rows or records in an RDBMS. The data model is a set of collections of documents that contain key-value collections. In a document store, the values can be nested documents or lists, as well as scalar values. The nesting aspect is one important differentiator with the advanced key-value stores we just presented. The attribute names are not predefined in a global schema, but rather are dynamically defined for each document at runtime. Moreover, unlike RDBMS tuples, a wide range of values are possible. A document stores data in tree-like structures and requires the data to be stored in a format understood by the database. In theory, this storage format can be XML, JSON, Binary JSON (BSON), or just about anything, as long as the database can understand the document's internal structure.[4]

Several systems are available in this category, the most popular and used in production environments is MongoDB.

2.2.2 Thing directory

The TMS offers a Thing Directory as a core microservice. This allows authorized clients to search for JSON-LD based descriptions for things exposed as part of the Gatekeeper marketplace of services.

Early work on the WoT emphasised the use of HTTP as the primary Internet protocol for accessing sensors and actuators using REST based APIs. Subsequent work focused on abstracting things in terms of software objects with properties, actions and events, decoupling client applications from the details of how these objects are connected to IoT devices. This considerably reduces the effort needed to build services across heterogeneous ecosystems, each with their own IoT standards. Also recognizing that the amount of different competing IoT standards will be with us for some time to come, and so, we need to find a solution. The WoT counters this fragmentation with an over-arching abstraction layer.

W3C's WoT builds upon an extensive suite of standards for the Semantic Web in which things are associated with HTTP based identifiers for use with the Resource Description Framework (RDF) for graph-based metadata. In April 2020, W3C released a standard for using JSON-LD for the WoT. JSON-LD is a serialisation of RDF in terms of the popular JavaScript Object Notation (JSON). Gatekeeper is using this standard as the basis for a marketplace of services relating to healthcare.

[4] RDF Database Systems, Triples Storage and SPARQL Query Processing, 2015, Pages 9-40. Chapter Two - Database Management Systems, https://doi.org/10.1016/B978-0-12-799957-9.00002-X

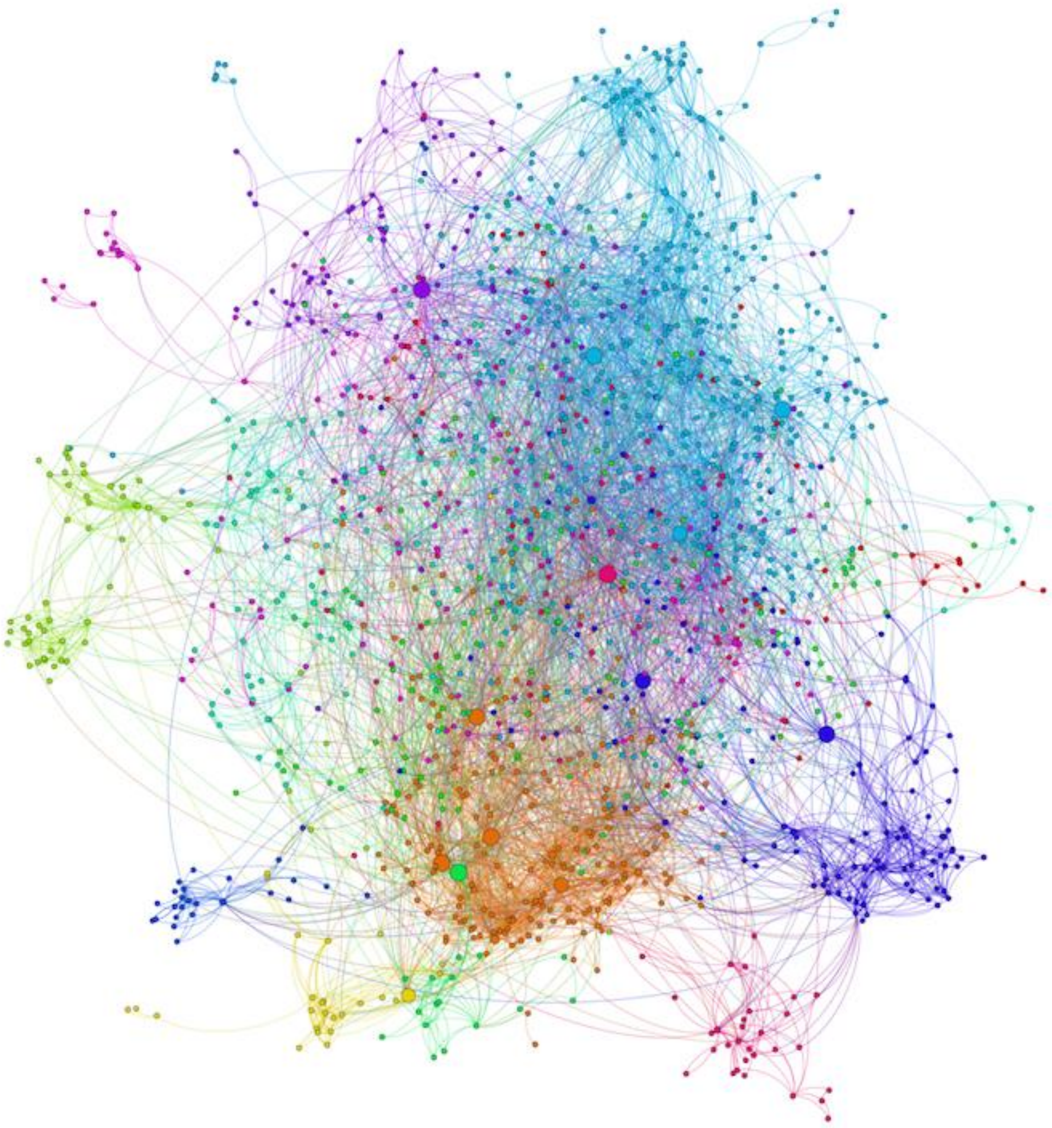


Figure 4 – Gatekeeper conceptual representation into WoT universe

Figure 4 shows the conceptual representation of Gatekeeper within the WoT universe that will be implemented by the Thing Directory. Following the figure, it is possible to imagine the Gatekeeper platform as a subset of things of the WoT universe that have a common property, in the figure represented with the same colour, for instance the turquoise one.

Regarding the real property, Gatekeeper will create a trusted island of things that will differ from the rest of universe of things because they will be certified by a certification authority. The Gatekeeper Certification Authority (GTA) is another core component of Gatekeeper platform and is described in the deliverable D4.5.

2.2.3 API Gateway

In a microservices architecture that implements a digital platform, scalability and resilience are key aspects that must be supported. The client apps usually need to consume functionality from many microservices, if that consumption is performed directly, the client has to handle multiple calls to microservice endpoints and authorize directly the consumers. If an application is an entire platform, as the Gatekeeper case, has many microservices, handling so many endpoints from the client apps can be a nightmare; so how do the clients of a microservices-based application access the individual services?

In this case, having an intermediate level or an intermediary (Gateway) can solve the issue (Figure 5). Without an API Gateway, the client apps must send requests directly to the microservices owners and that raises problems, such as:

- Coupling: Without the API Gateway pattern, client apps are coupled to the internal microservices. The client apps need to know how the multiple areas of the application are decomposed in microservices. When evolving and refactoring the internal microservices, those actions impact maintenance because they cause breaking changes to the client apps due to the direct reference to the internal microservices from the client apps. Client apps need to be updated frequently, making the solution harder to evolve.
- Too many round trips: A single page/screen in the client app might require several calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency. Aggregation handled in an intermediate level could improve the performance and user experience for the client app.
- Security issues: Without a gateway, all the microservices must be exposed to the "external world", making the attack surface larger than if you hide internal microservices that are not directly used by the client apps. The smaller the attack surface is, the more secure your application can be.
- Cross-cutting concerns: Each publicly published microservice must handle concerns such as authorization and SSL. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified.[5]

[5] The API gateway pattern versus the Direct client-to-microservice communication, https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern , Last Access September 2020.

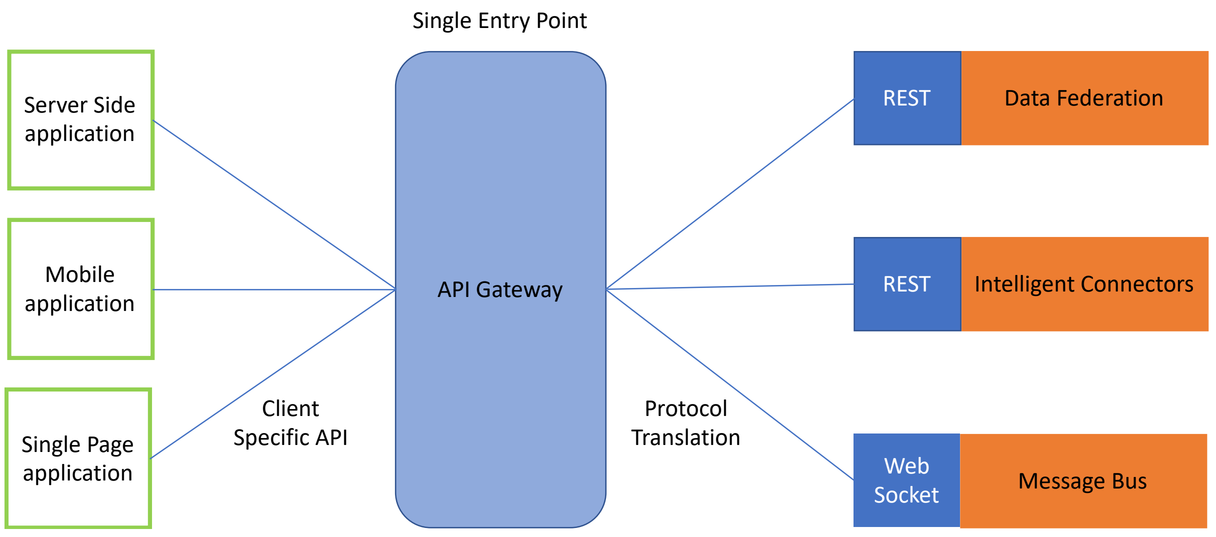


Figure 5 – API Gateway pattern in microservices architecture

Within Gatekeeper, by using container and orchestration technologies in conjunction with an API Gateway it is possible to offer multiple features. This API Gateway configuration provides very important features described by the following design patterns:

- **Reverse proxy**. The API Gateway offers a gateway routing to redirect or route requests (usually HTTP requests) to the endpoints of the internal microservices. The gateway provides a single endpoint or URL for the client apps and then internally maps the requests to a group of internal microservices. This routing feature helps to decouple the client apps from the microservices, but it is also convenient when modernizing a monolithic API by sitting the API Gateway in between the monolithic API and the client apps. In this configuration, new APIs can be added as new microservices while still using the legacy monolithic API until it is split into many microservices in the future.
- **Requests aggregation**. As part of the gateway multiple client requests (usually HTTP requests) targeting multiple internal microservices into a single client request can be aggregated. This pattern is especially convenient when a client page/screen needs information from several microservices. With this approach, the client app sends a single request to the API Gateway that dispatches several requests to the internal microservices and then aggregates the results and sends everything back to the client app. The main benefit and goal of this design pattern is to reduce chattiness between the client apps and the backend API, which is especially important for remote apps out of the data centre where the microservices live, like mobile apps or requests coming from SPA apps.
- **Gateway offloading.** One of the core features offered by API Gateway is offloading functionality from individual microservices to the gateway. This pattern simplifies the implementation of each microservice by consolidating cross-cutting concerns into one tier. This is especially convenient for specialized features that can be complex to implement properly in every internal microservice, such:
  - Authentication and authorization
  - Service discovery integration
  - Response caching

- Retry policies, circuit breaker, and QoS
- Rate limiting and throttling
- Load balancing
- Logging, tracing, correlation
- Headers, query strings, and claims transformation
- IP whitelisting.

3 TMS Components implementation

As already described in the previous section, the Thing Management System is composed of three main components: The Thing Directory, its database, and the API Gateway. In this section, it will be shown how each internal component is implemented.

3.1 Thing Directory

The Thing Directory works as a broker service for the API that exist within the TMS. Its main function is to provide the TDs of the APIs that are integrated with the system. The source code for this component is accessible within the repository: https://gitlab.lst.tfo.upm.es/gatekeeper/cluster-demo/cluster-kubernetes/-/tree/master/thing-directory .

When accessing the Thing Directory for the first time it would provide the user with its own TD in order to navigate through it functions. In it we will be able to see how to access the API's TDs stored in the database, and how to authenticate ourselves to access them.

This component is implemented with the IBM Loopback framework v4[6]. Loopback 4 is a highly-extensible, open-source Node.js[7] framework for:

- Building dynamic end-to-end REST APIs.
- Access data from major relational databases, MongoDB, SOAP and REST APIs.
- Incorporate model relationships and access controls for complex APIs.
- Separate components for file storage, third-party login, and OAuth 2.0.

Since Loopback is based on ExpressJS[8] LoopBack middleware, it is the same as Express middleware. However, LoopBack adds the concept of phases, to clearly define the order in which middleware is called. Using phases helps to avoid ordering issues that can occur with standard Express middleware[9].

LoopBack generalizes backend services such as databases, REST APIs, SOAP web services, and storage services as data sources. Data sources are backed by connectors that then communicate directly with the database or other back-end service. Applications do not use connectors directly, rather they go through data sources.

However, the standard used for the TDs was established by the W3C and it includes the standard JSON-LD that uses special characters such "@" in some of the keys. Loopback misinterpreted it as a reference and not as a valid key. The solution was to modify the key for the model to support such special characters.

```
{
    "@context": ["https://www.w3.org/2019/wot/td/v1"],
    "title": "Broker service",
```

[6] Loopback 4, https://loopback.io/doc/en/lb4/, Last Access September 2020.

[7] NodeJS, https://nodejs.org/en/, Last Access September 2020.

[8] ExpressJS Framework, https://expressjs.com/, Last Access September 2020.

[9] Loopback Middleware, https://loopback.io/doc/en/lb3/Defining-middleware.html, Last Access September 2020.

```
    "description": "Broker service para la gestion de multiples recursos",
    "base": "http://192.168.23.131:32015",
    "security": ["oauth2_sc"],
    "securityDefinitions": {
        "oauth2_sc": {
            "scheme": "oauth2",
            "authorizationUrl": "http://192.168.23.131:32017/auth/realms/Fhir-
test/account",
            "flow": "code"
        }
    },
    "properties": {
        "info": {
            "title": "Information about broker service",
            "description": "Informacion sobre el broker service",
            "output": {
                "@type": "Thing"
            },
            "forms": [
                {
                    "op": "readproperty",
                    "href": "http://192.168.23.131:32015",
                    "htv:methodName": "GET"
                }
            ]
        },
        "servicesList": {
            "title": "Services list",
            "description": "Lista de servicios",
            "@type": "jsonschema:ArraySchema",
            "output": {
                "@type": "Thing"
            },
            "forms": [
                {
                    "op": "readproperty",
                    "href": "http://192.168.23.131:32015/b/broker",
                    "htv:methodName": "GET",
                    "security": ["oauth2_sc"]
                }
            ]
        },
        "servicesCount": {
            "title": "Services count",
            "description": "Servicios disponibles",
            "output": {
                "type": "xsd:integer"
            },
            "forms": [
                {
                    "op": "readproperty",
                    "href": "http://192.168.23.131:32015/b/broker/count",
                    "htv:methodName": "GET",
                    "security": ["oauth2_sc"]
                }
            ]
        },
        "getServiceById": {
            "title": "Service by id",
            "description": "Obtener un servicio segun su id",
            "uriVariables": {
                "id": {
                        "type": "string"
                    }
                },
                "output": {
                    "@type": "Thing"
                },
                "forms": [
                    {
                        "op": "readproperty",
                        "href": "http://192.168.23.131:32015/b/broker/{id}}",
                        "htv:methodName": "GET",
                        "security": ["oauth2_sc"]
                    }
                ]
            }
        },
        "actions": {
            "auth": {
                "title": "Authentication",
                "description": "Access this endpoint to authenticate",
                "forms": [
                    {
                        "op": "invokeaction",
                        "href": "http://192.168.23.131:32017/auth/realms/Fhir-
test/account",
                        "htv:methodName": "GET"
                    }
                ]
            },
    }
}
```

The TD of the Thing Directory focuses on guiding the user through the available services. For that, first of all, in the metadata of the TD is defined the context, linking it though the JSON-LD protocol to the guidelines of the W3C TD. In that section, the security is also defined. In “SecurityDefinitions” key, a list of the security methods used in the TD is provided, in this case, there is only one type of security protocol, that being the OAuth2. In “security” key a list of the names of the security definitions is provided. Furthermore, under the “properties” key the main actions of the Thing Directory are explained, along with their endpoints. Only one of these properties is not under security, which is the “info” property, it is the root of the Thing Directory and being the one providing this TD. Inside the actions tree, only one action is defined, informing the user of the method to authenticate inside the Thing Directory.

The thing directory is linked with a MongoDB database that stores all the TDs of the Gatekeeper platform.

LoopBack has a concept that is called a Data Source[10] for managing data connections, which generally consists of a named configuration for an instance of the Connector that represents data on an external system.

A DataSource has, by default, the database connector configuration inside the file `${dataSource.dataSourceName}.datasource.json`. As our main goal is to offer support so

[10] https://loopback.io/doc/en/lb4/DataSources.html, last access October 2019

that the configurations may be provided through environment variables, using the file format ".json" makes this task impracticable. That way, it's needed to create a configuration file in Typescript that consists on exporting a simple object, containing all configuration that were provided in a static way by the ".json" file.

```
const db_config: object = {
    name: "mongo",
    connector: 'mongodb',
    url: process.env.MONGODB_URI || 'mongodb://127.0.0.1:27017/quest-service',
    useNewUrlParser: true
};

export { db_config }
```

We can observe that the URI for connection with the database is received through the environment variable MONGODB_URI, otherwise the default value will be used. That being done, modify the class `${dataSource.dataSourceName}.datasource.ts` which can be used to override the default DataSource behavior programmatically to not use the connector configuration stored in the file .json, but to use the Typescript file created previously containing all the configuration needed for the database connector.

```
import { juggler } from '@loopback/repository';
import { db_config } from './mongo.datasource.config';

export class MongoDataSource extends juggler.DataSource {
  static dataSourceName = 'mongo';

  constructor(
    dsConfig: object = db_config,
  ) {
    super(dsConfig);
  }
}
```

For this example, only the MongoDB URI was made available for configuration by environment variable, however, other settings could also be easily made available. The idea is that everything that needs to be configured later is available by environment variable. It is important to note that the name of the environment variables does not matter, we only recommend that suggestive names be used for a better understanding.

3.2 Thing Directory Database

The Thing Directory database will be a MongoDB instance configured to be used with the Loopback application that implements the Thing Directory as described in the previous section. Inside the database, the TDs will be stored, to be later retrieved by the Thing Directory. The database schema is aligned with the source code contained inside the model elements of the LoopBack application: https://gitlab.lst.tfo.upm.es/gatekeeper/cluster-demo/cluster-kubernetes/-/tree/master/thing-directory/src/models .

MongoDB[11] was chosen for its easiness when working with unstructured types of data, in our case it helps to store the JSON information of the resources contained within the Thing Directory, without the need of further configuration, structure management and format conversion. The document data model is a powerful way to store and retrieve data that allows developers to move fast. MongoDB's horizontal, scale-out architecture can support huge volumes of both data and traffic.

In conventional relational databases, a field must exist for each piece of information—and in every entry. If the information is not available, the cell is kept empty, but it must still exist. Document-oriented databases are much more flexible: the structure of individual documents does not have to be consistent. Even large volumes of unstructured data can be accommodated in the database. Plus, it is easier to integrate new information. While in the case of a relational database a new information criterion must be added to all datasets, new information only needs to be included in just a few datasets in a document store. The additional content can be added to further documents, but it is not required. Moreover, with document stores the information is not distributed over multiple linked tables. Everything is contained in a single location, and this can result in better performance. However, this speed advantage is only realized in document databases provided that one does not attempt to use relational elements: references do not really suit the concept of document stores.[12]

Furthermore, the database schema is linked with the Thing Directory service thought the WoT TD Information Model[13]. This model is referred within the service that has a direct object mapping within the object stored within the database. The lack of format conversion improves the performance against any other data base technologies where a format conversion (e. g. tables into JSON as in the case of SQL based technologies) is needed.

3.3 API Gateway

The source code for this component is accessible within the repository: https://gitlab.lst.tfo.upm.es/gatekeeper/cluster-demo/cluster-kubernetes/-/tree/master/gateway .

According to the specifications of the Gatekeeper architecture the internal microservices are not exposed to the outside world being the responsibility of the API Gateway to expose the interfaces of each microservice. Thus, it is necessary that each microservice is registered. The Gateway API of the Gatekeeper platform relies on Express Gateway, a microservice Gateway API built into Express.js[14].

[11] Mongo DB, https://www.mongodb.com/ , Last Access September 2020.

[12] Benefit of document oriented dbs, https://www.ionos.com/digitalguide/hosting/technical-matters/document-database/ , Last Access September 2020.

[13] Web of Things Information Model, https://www.w3.org/TR/wot-thing-description/#sec-vocabulary-definition , Last Access September 2020.

[14] Express Gateway, https://www.express-gateway.io , Last Access September 2020.

This framework allows to define the proxy rules to redirect the user to the desired APIs, behind an authorization wall, in order to ensure that the user has the access level. Although the authentication is managed by the Gatekeeper Trust Authority (GTA), the TMS API Gateway oversees defining the authorization rules to access each API, with the possibility of integrating user roles and scopes.

In development, the only endpoint available without login in was the root of the Thing Directory, which answered with its TD, providing the relevant instructions to authenticate. Once authenticated the user could access the rest of the development cluster.

The advantage of Express Gateway (built on top of Express.js) is the ability to expand its functionalities with plugins. The development of the plugins is closely related to the development of middleware in Express.js.

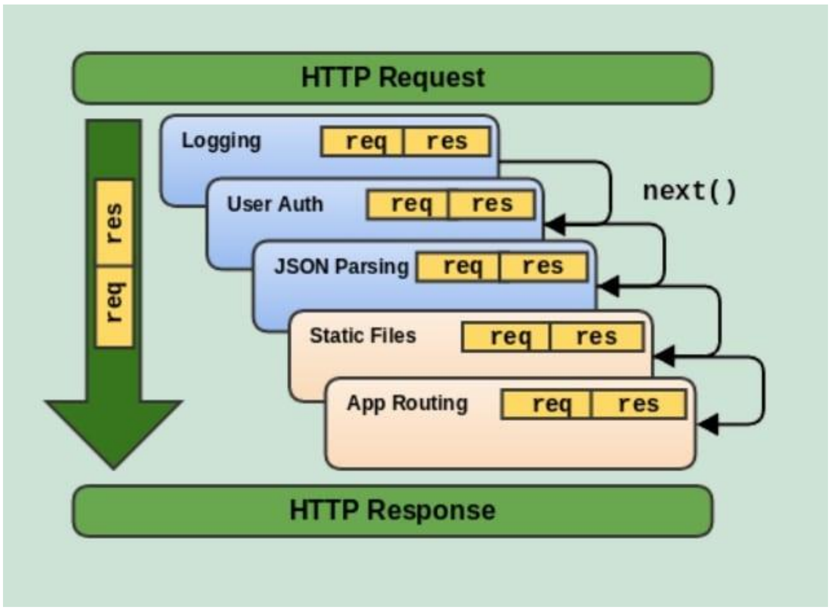


Figure 6 – Express middleware architecture

As Figure 6 shows, an Express middleware is a piece of code that can be injected between the reception of an HTTP request and the delivery of an HTTP response.

With the same logic, an Express Gateway plugin includes a set of actions that are programmed and injected in the pipeline of the service and are executed in order.

```yaml
pipelines:
  thing-directory:
    apiEndpoints:
      - thing-directory
    policies:
      - custom-log:
      - keycloak-protect:
      - proxy:
          - action:
              serviceEndpoint: thing-directory
              changeOrigin: true
```

As seen in the code above, the plugins are specified under the policies key, they can also be programmed as routes and conditions. In the example it is defined two custom policy plugins, one is dedicated to logging (custom-log), and the other one is authentication and authorization (keycloak-protect). The advantage of this type of plugins is that, as they work as a middleware, we can inspect and modify the request before it is sent to the server. The proxy policy is inherent to Express Gateway and is the responsible component of redirecting the requests. As they are executed in order, the first one to execute would be the log, the next one the authentication, and finally the redirection.

4 Component interaction and integrations

Apart from these microservices, the TMS version 1 needs a strong interaction with the GTA and with the service owners that are part of the Gatekeeper ecosystem. One of this service is the Data Federation Framework which builds the Gatekeeper healthcare data space (T4.4) in agreement with the semantic models defined into T3.4 and the FHIR implementation guide coming from T3.5.

The general approach for components integration into TMS is through endpoints registration with pipelines including custom policies into Express Gateway and registration of the corresponding TD into the Thing Directory. Custom policies will be implemented with a plugin.

This mechanism allows a high degree of flexibility if requirements were to change and very little interaction among components owners.

4.1 Generic approach for integration based on Thing Description

In order to register a new microservice described by a valid TD, within the TMS, there are basically three steps to be done in the configuration file of Express Gateway and one additional step in the Thing Directory service:

1. The API endpoints registration (API Gateway).
2. The service endpoints registration (API Gateway).
3. The pipeline registration (API Gateway).
4. The registration of TD into Thing Directory service.

The API endpoints to be registered into the API Gateway are described in the key "form" of the interaction patterns (actions, events, properties) of the TD.

4.1.1 API Endpoints Registration

Endpoints, also known as URLs, are the way Express Gateway exposes the microservices. It is through endpoints that clients will make requests to the Gatekeeper platform. In the following code snippet example, the registration of three sets of API endpoints is shown. One of them is a generic FHIR server, another is the Thing Directory service and the last one is a thing directory info that is like info entry point of the system. This entry point (root directory) is needed because the Thing Directory is protected, while the root is publicly open and describe how to access the TD of the Thing Directory. From the entry point, the user can authenticate himself and then access the internal part of the Thing Directory, where all the TD are available.

```
apiEndpoints:
  thing-directory-info:
    host: '*'
    paths: ['/']
  thing-directory:
    host: '*'
    paths: ['/b/broker', '/b/broker/*']
  fhir-server:
    host: '*'
    paths: ['/hapi-fhir-jpaserver', '/hapi-fhir-jpaserver/*']
```

Among the available options for the entry points registration, there are:

- **host** – Sets the hostname to accept the request. Its value is a string and will be compared to the "HOST" header of the request.
- **paths** – Defines the paths of the routes that must follow the route patterns of Express.js[15], that way the wildcard pattern is supported. Its value can be a string or an array of strings
- **methods** – Defines the HTTP methods required to accept requests. Values are defined in an array.
- **scopes** – Sets the scopes required to access the resource. Values are defined in an array.

For more information regarding endpoints, see the official documentation of Express Gateway[16].

4.1.2 Service Endpoints registration

Service endpoints are used to define the URLs of each microservice, through which Express Gateway performs the proxy process. A service Endpoint must have an URL with a single string or an array of strings that can be used as load balancing.

In the code snippet below, we can see the URL definitions for more two microservices: Thing Directory and FHIR server. An important point is that URLs can receive values through environment variables, a detail that makes it possible to configure the Gateway in Docker Compose files.

```
serviceEndpoints:
  thing-directory:
    url: 'http://broker:3000'
  fhir-server:
    url: 'http://fhir:8082'
```

Among the available options for serviceEndpoints we have:

- **url** – Defines the protocol and hostname that will be used for proxy requests.
- **urls** – Defines the protocol and hostname that will be used for proxy requests with load balancing.
- **proxyOptions** – Sets options that will be used in the proxy policy.

[15] Express Gateway Router, https://expressjs.com/en/4x/api.html#router, , Last access September 2020

[16] Express Gateway apiEndpoints, https://www.express-gateway.io/docs/configuration/gateway.config.yml/apiEndpoints, Last access September 2020

For more information, see the official Express Gateway documentation about serviceEndpoints[17].

4.1.3 Pipelines registration

Pipelines specify the main Express Gateway operations, linking the endpoints and entities, through the request and response flow. The code snippet below shows some pipelines included in the platform. Here it can be seen how the link between the API entry points, and the service endpoints work. They are complemented by the plugins specified in Section 3.3, as it can be seen although the logging plugin policy is integrated into all of the pipelines, the authentication plugin policy is not, this is caused by the need to have the root of the Thing Directory open for an initial description. Moreover, this further explains the need to have to different API endpoint registrations for the Thing Directory in Section 4.1.1.

```yaml
pipelines:
  thing-directory:
    apiEndpoints:
      - thing-directory
    policies:
      - custom-log:
      - keycloak-protect:
      - proxy:
          - action:
              serviceEndpoint: thing-directory
              changeOrigin: true
  thing-directory-info:
    apiEndpoints:
      - thing-directory-info
    policies:
      - custom-log:
      - proxy:
          - action:
              serviceEndpoint: thing-directory
              changeOrigin: true
  endpot:
    apiEndpoints:
      - fhir-server
    policies:
      - custom-log:
      - keycloak-protect:
      - proxy:
          - action:
              serviceEndpoint: fhir-server
              changeOrigin: true
```

Among the available options for a pipeline, we have:

[17] Express Gateway Pipelines registration, https://www.express-gateway.io/docs/configuration/gateway.config.yml/serviceEndpoints, last access September 2020.

- **condition** – Defines a rule that must be satisfied to trigger the corresponding action. This functionality can be expanded further with a plugin.
- **action** – Defines the parameters that will be used in the action.

Pipelines and plugins are used to match requirements that come from component owners (e. g. endpoints outside Gatekeeper platform) of from the TD itself (e. g. authorization though OAuth2.0).

For details on defining pipelines in Express Gateway and all possible configurations, see the official documentation[18].

4.1.4 Alignment of TD into Thing Directory service

Through previous steps, the endpoints of a TD have been registered and are available eventually though authentication as described in the TD itself. However, the registration into the gateway is not enough for public availability of the service.

In order to solve this issue, the last step is needed. It foresees that the TD must be stored into the Thing Directory Database. After that, the TD is available in the Thing Directory service and is publicly available for any client.

4.2 Interaction and integration with Data Federation

One example of TD based on TD Information Model is the FHIR server TD, that is a simulation in the development environment of the Data Federation. In fact as for Data Federation and Integration component design (see T4.4) a FHIR server (per pilot) will be adopted to hold data sent by different data sources (IoT and remote EHR) once their original format has been converted to the GK FHIR Profile.

```
{
    "@context": [
      "https://www.w3.org/2019/wot/td/v1",
      {
          "xsd": "http://www.w3.org/2001/XMLSchema#",
          "td": "https://www.w3.org/2019/wot/td#",
          "FHIRServer":  {"@id": "td:Thing"},
          "conformance": {"@id": "xsd:anyURI"}
      }
    ],
    "@type":"FHIRServer",
    "title": "Gatekeeper pilot x FHIR server",
    "description" : "A FHIR server implementation",
    "security": ["oauth2_sc"],
    "securityDefinitions": {
        "oauth2_sc": {
```

[18] Express Gateway pipeline, https://www.express-gateway.io/docs/configuration/gateway.config.yml/pipelines, last access September 2020.

```
            "scheme": "oauth2",
            "authorizationUrl": "http://192.168.23.131:32017/auth/realms/Fhir-test/account",
            "flow": "code"
        }
    },
    "conformance": "http://192.168.23.131:32021/hapi-fhir-jpaserver/fhir/metadata"
  }
```

This TD defines the actions needed to access the FHIR Gatekeeper server. The “security” and “securityDefinitions” keys are the same that in the Thing Directory’s TD, defining the OAuth2 protocol and providing the endpoint of authorization.

Finally, a key to the endpoint containing the CapabilityStatement Resource of the FHIR server is provided. In fact, a FHIR capability statement describes the operations exposed by the system and is used to inform external system about functionalities and technical specification of a specific FHIR server. It contains the list of the FHIR resources provided by the server and for each resource the possible operations are reported (e.g. read, delete, update, etc.). Moreover, for each resource a set of search parameters is described which represent the attributes to involve in specific queries. In the figures below a logic organization of the CapabilityStatement and, following, the web interface of an HAPI server FHIR.

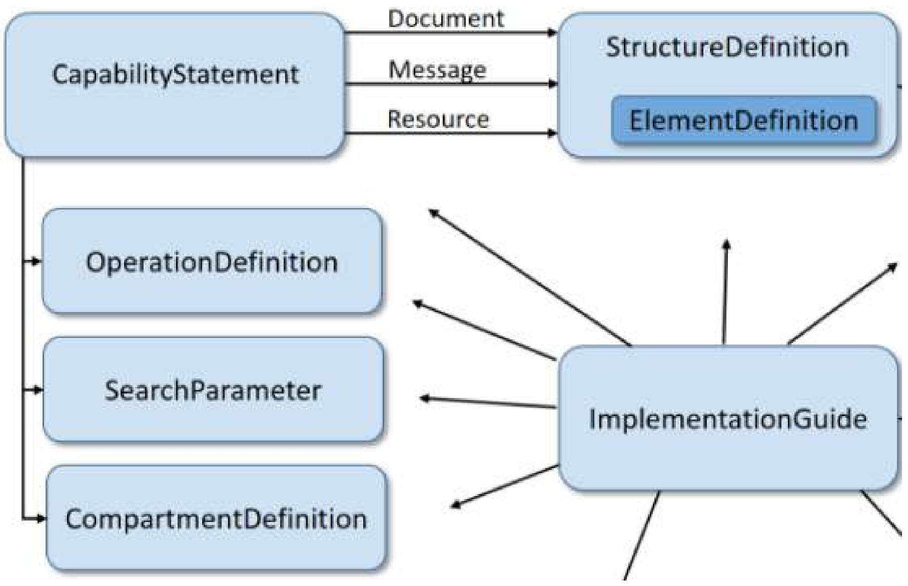
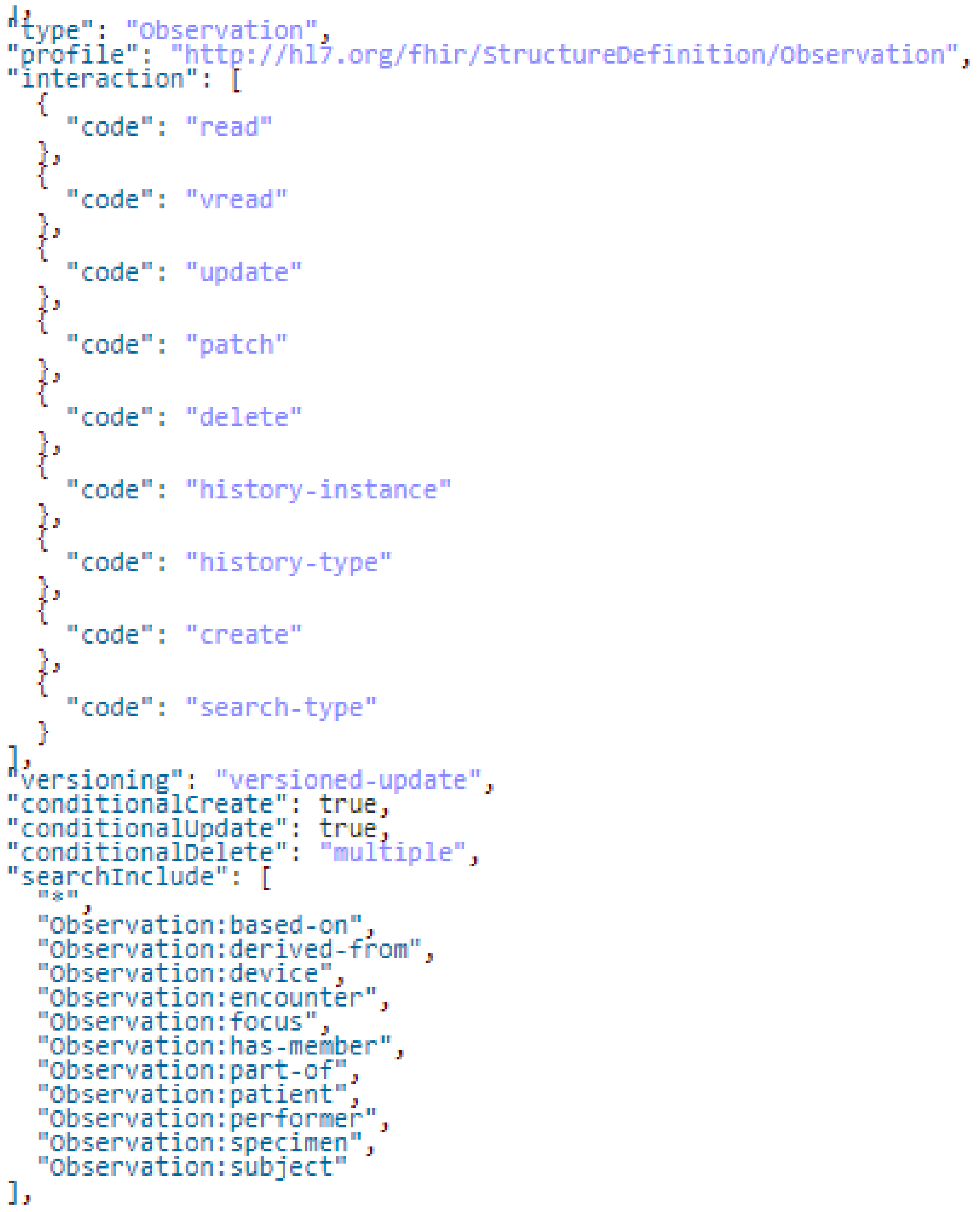


Figure 7 - CapabilityStatement logical view



```
},
"type": "Observation",
"profile": "http://hl7.org/fhir/StructureDefinition/Observation",
"interaction": [
  {
    "code": "read"
  },
  {
    "code": "vread"
  },
  {
    "code": "update"
  },
  {
    "code": "patch"
  },
  {
    "code": "delete"
  },
  {
    "code": "history-instance"
  },
  {
    "code": "history-type"
  },
  {
    "code": "create"
  },
  {
    "code": "search-type"
  }
],
"versioning": "versioned-update",
"conditionalCreate": true,
"conditionalUpdate": true,
"conditionalDelete": "multiple",
"searchInclude": [
  "*",
  "Observation:based-on",
  "Observation:derived-from",
  "Observation:device",
  "Observation:encounter",
  "Observation:focus",
  "Observation:has-member",
  "Observation:part-of",
  "Observation:patient",
  "Observation:performer",
  "Observation:specimen",
  "Observation:subject"
],
```

Figure 8 - CapabilityStatement web page: observations section

All endpoints described into the CapabilityStatement resources are registered into the API gateway. In this way, starting from the TMS TD is possible to navigate through every FHIR service.

4.3 Interaction and integration with GTA

TMS and GTA cooperate for managing users, audit/track activities and validating things within the platform. From the point of view of TMS, the GTA is providing the following services that need to be integrated:

- Thing Action Tracking/Audit, every time an endpoint is called the TMS uses the GTA audit service in order to track and log activities.

- User Management Module, the GTA is exposing a set of APIs for the management of Gatekeeper users. By using such APIs, the TMS asks to the GTA for user authorization in order to allow a Gatekeeper valid user to access the things he is authorized for.
- Certification Authority, when an authorized user wants to add a new thing into the Gatekeeper platform, he has to certify the associated TD with the GTA. In this context, the TMS is the man in the middle of the process.

4.3.1 Integration of Thing Action Tracking

The Thing Action Tracking will be integrated at 2 different levels: core Gatekeeper component and Gatekeeper Things.

Its integration as a GATEKEEPER core component implies the development of a plugin that defines an action tracking policy. Such policy will integrate the Thing Action Tracking API in a way that any relevant call to sensible API endpoints, associated to a TD registered into the TMS, will be logged within the Thing Action Tracking module associated to the Gatekeeper platform.

On the other hand, when the Thing Action Tracking is integrated as Gatekeeper Thing it follows the process described in Subsection 4.1. In this case, the Thing Action Tracking is a service that is available for developers that are using Gatekeeper platform.

4.3.2 Integration of User Management Module

For the integration of the GTA inside the TMS, we are using Keycloak as a handler. This is included inside the Gateway through a plugin that executes a policy whenever a request is made to a pipeline that has the policy enabled.

```
pipelines:
  thing-directory:
    apiEndpoints:
      - thing-directory
    policies:
      - custom-log:
      - keycloak-protect:
      - proxy:
          - action:
              serviceEndpoint: thing-directory
              changeOrigin: true
```

When the plugin is installed inside Express Gateway, a set of parameters are needed for the plugin to correctly work. As can be seen in the code snippet below, it is necessary to insert the *auth-server-url*, this is because the authorization and authentication will not be run from the gateway itself.

```
plugins:
  express-gateway-keycloak:
    package: express-gateway-keycloak
    keycloakConfig:
      realm: Fhir-test
      auth-server-url: "http://192.168.23.131:32017/auth"
      ssl-required: external
      resource: fhir-broker
      public-client: true
      confidential-port: 0
```

When a request to access a resource comes to the TMS, it will go to the Gateway which will ask the GTA through the plugin if the user is authorized/authenticated, if the response is positive, it will let the request go through.

4.3.3 Integration of Certification Authority

The integration of the Certification Authority will be performed with a plugin implementation that will be integrated into the TMS. The details and specification of its functionalities will be addressed in the next version of the TMS.

5 TMS Deployment environments

The cluster's deployment framework is Kubernetes, allowing to scale and maintain the TMS without interfering with other services.

Kubernetes was chosen because of the resilience that it naturally provides. It allows replicating the services and resources of the cluster without the need of any downtime improving the scalability of the cluster. Moreover, in the case one of those replicas were to malfunction or go down it would create new ones automatically, reducing the downtime in case of an outage. As Kubernetes operates as a master-slave configuration, in case a change in the physical hardware is needed, it would also allow making the change seamlessly.

The source code for this component is accessible within the repository: https://gitlab.lst.tfo.upm.es/gatekeeper/cluster-demo/cluster-kubernetes

5.1 Development environment

The development environment at UPM premises follows the schema in Figure 9.

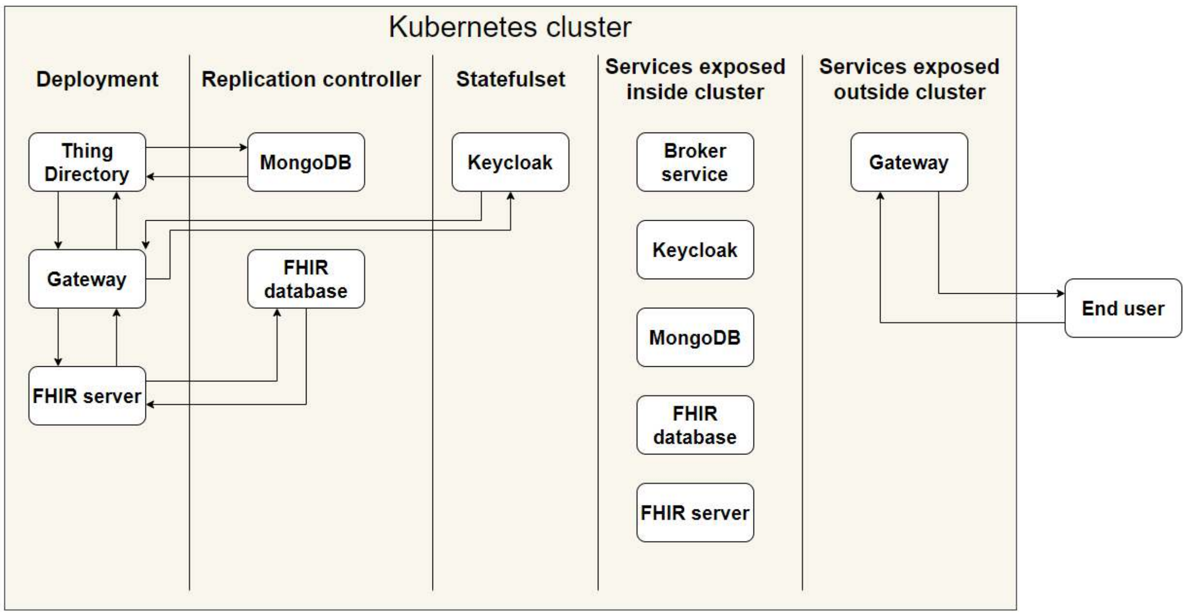


Figure 9 – Kubernetes deployment schema in development environment

Due to COVID-19, it was not possible to have access to an advance infrastructure to test the deployment on. As such, the development and production testing were done internally at UPM, limiting the characteristics available to use.

5.1.1 Thing Directory

The Thing Directory, as mentioned in Section 3.1, is a Loopback application. To deploy it to a Kubernetes environment, first we must compile the application as a Docker image. Once done that, we can now freely deploy it inside a cluster.

For the testing environment, the Thing Directory uses a *Deployment* configuration with 4 replicas, meaning it has no state, or in other words, persistence. However, that is not problem because as mentioned in 3.2, another component of the cluster is going to be the Thing Directory Database. In our case a MongoDB instance.

The MongoDB is deployed as a *ReplicationController* and is exposed inside the cluster with a *NodePort* service so that is accessible for the Thing Directory. The Thing Directory is also exposed by a *NodePort* service so that is only accessible inside the cluster.

5.1.2 Gateway

The Gateway as seen in Section 3.3 is implemented using Express Gateway, which is a NodeJS application. Because of that, the steps needed to integrate it with Kubernetes are the same with those in the case of the Thing Directory.

As the Gateway only has two configuration files, and those are compiled along with the application when generating the image, it does not have a need for persistence. The type chosen for integration is a *Deployment.*

A plugin for the Gateway was also written. The purpose of this plugin is to log everything that happens inside the Gateway. It can be configured as to where to store the data, and to what data to store.

The Gateway is the last part to be loaded into the cluster as the endpoints of the different services must be known. Furthermore, it is exposed by service to the outside world, it is the only component of the TMS visible to the outside.

5.1.3 Authorization and authentication

The authorization and authentication are managed by a Keycloak[19] instance and the Gateway configuration files. Keycloak is an authentication framework, allowing us to easily integrate users and rules, being also OAuth2 compliant.

The authorization inside the cluster is managed by the Gateway by defining rules and access permissions to the routes defined inside the component. One example of this is in the development stage there were only two components being redirected by the Gateway, namely the Thing Directory and the Data Federation Framework. To access the Data Federation Framework the user must be authenticated. However, in the case of the Thing Directory there were two rules, one of them allowed unauthorized access to the root of the Thing Directory, which contains the TD of the service. The other one restricted access to users authenticated to the rest of the Thing Directory, such as the list of resources served, or theirs' TDs.

The authentication was managed by the Keycloak instance, which provided an endpoint for login purposes. As Keycloak has the need to store data, the deployment type chosen

[19] Keycloak, https://www.keycloak.org/, Las Access September 2020.

was *Statefulset.* As with the other components, Keycloak is only exposed inside the cluster.

5.1.4 Resources

In development, the only resource implemented in the cluster is the Data Federation Framework. As the Data Federation Framework has the need to persist its data, an instance of MySQL is integrated along with the application.

MySQL is deployed using a *Statefulset* in a similar way to Keycloak and exposed only to the cluster. In a similar way to the Thing Directory, the Data Federation Framework is deployed as a *Deployment*, and connected to the database by the Kubernetes service. The Data Federation Framework is only exposed inside the cluster.

5.2 Production environment

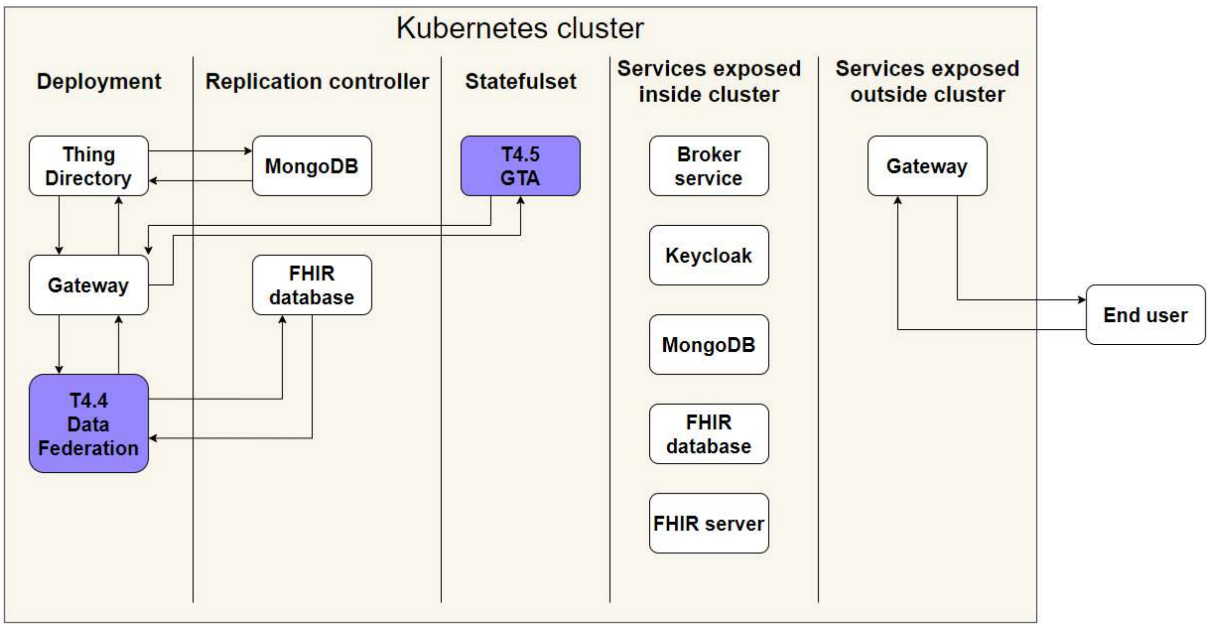


Figure 10 – Kubernetes deployment schema in production environment

The production environment is less complex than the development one (Figure 10). In production, components like Keycloak and FHIR server are managed and deployed by TMS (T4.5.) and Data Federation Framework (T4.4.) respectively.

The main two differences between the development environment and the production environment are the Security and Resource components. While the security component in the development phase was filled using a standalone Keycloak server, in the production environment it would be replaced by the Gatekeeper Trust Authority (GTA). Moreover, for the test on the resources provided by the Thing Directory we used the HAPI Starter FHIR server[20], in the production phase it will be replaced by the Data Federation Framework. One more difference that can't be seen in Figure 7 is that in the development phase, new resources and services were added directly to the Thing Directory, without any kind of validation. However, in the production environment, the task of adding new resources and services to the system will be gathered through the GTA.

[20] HAPI-FHIR Starter Project, https://github.com/hapifhir/hapi-fhir-jpaserver-starter, last access September 2020.

6 Conclusion

This deliverable is providing the initial description of the Thing Management System as well as the description of how the overall GATEKEEPER platform will work. It includes references to the source code of the projects that implements the microservices that are part of the component.

It describes how things will be integrated within the GATEKEEPER platform and made accessible to a developer, allowing the navigation through them in a secure, scalable and isolated infrastructure.

Due to the COVID situation, the demonstration in a cloud environment will be provided in the next version of the deliverable (due at M24) as well as the detailed description of the integration of the validation and certification process of a thing.