Collaborative Network for Industry, Manufacturing, Business and Logistics in Europe

D5.1

Advanced Platform Infrastructure

Project Acronym	NIMBLE	
Project Title	Collaboration Network for Industry, Manufacturing, Business and Logistics in Europe	
Project Number	723810	
Work Package	WP5	Value-Added Business Services for the NIMBLE Platform
Lead Beneficiary	IBM	
Editor	Benny Mandler	IBM
Reviewers	Wernher Behrend	SRFG
Dissemination Level	PU	
Contractual Delivery Date	31/01/2020	
Actual Delivery Date	31/01/2020	
Version	V1.0	

Abstract

The NIMBLE project aims to perform research leading to the development of a cloud platform specifically targeted to supply chain relationships and logistics. Core capabilities will enable firms to register, publish machine-readable catalogues for products and services, search for suitable supply chain partners, negotiate contracts and supply logistics, and develop private and secure information exchange channels between firms. The intention is to support a federation of such NIMBLE instances, all providing a set of core services, and each potentially specifically tailored to a different aspect (regional, sectorial, topical, etc.).

The main goal of this document is to present the advanced platform infrastructure, which supports added value services on top of the existing core services. The document starts from a description of the underlying platform building blocks, proceeds to describe the manner in which several such individual instances can be made to collaborate in a federation, and finally presents two of the envisioned advanced services to support scenarios in blockchain based supply chain.

Document History

Version	Date	Comments
V0.1	20/09/2019	Initial populated version
V0.2	20/10/2019	First initial internal review
V0.3	15/12/2019	First version for review
V0.4	13/01/2020	Second version incorporating review comments
V1.0	15/01/2020	Final version

Table of Contents

List of Figures

List of Tables

Acronyms

Table 1: Acronyms

Acronym	Meaning
ACL	Access Control
API	Application Programming Interface
B2B	Business to Business
ELK	ElasticSearch, Logstash, Kibana
EPCIS	Electronic Product Code Information System
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
IIoT/ Industrie 4.0	Industrial Internet of Things
IoT	Internet of Things
K8s	Kubernetes
NIMBLE	Collaboration Network for Industry, Manufacturing, Business and Logistics in Europe
PaaS	Platform as a Service
REST	Representational State Transfer
RFID	Radio Frequency Identification
SDK	Software Development Kit
SaaS	Software as a Service
SDN	Software Defined Network
SOA	Service Oriented Architecture
TLS	Transport Layer Security
VM	Virtual Machine
WP	Work Package
XaaS	Everything as a Service

Glossary

- Camunda – BPMN workflow engine
- ElasticSearch - high-performance indexing and search system that can be integrated with the CouchBase scalable data back-end.
- EPCIS – standard for tracking events
- Eureka (Netflix) - Service registry and discovery component
- Ingress - management of access to the services inside a k8s cluster, from entities outside the cluster. Can provide load balancing, SSL termination and name-based virtual hosting
- ISTIO - An open platform to connect, manage, and secure microservices. https://istio.io/
- Jenkins – automate software development stages
- Keycloak - identity and access control package
- Kibana - Data visualization for Elasticsearch based data
- Kubernetes – a platform for managing containerized workloads
- Microservices – an architectural paradigm in which an application is comprised of a collection of loosely coupled services
- Logstash – Collect, parse and transform logs
- OAuth2 – authorization framework
- Platform-as-a-Service - a cloud computing concept which offers the developer and deployer of cloud-based applications the infrastructure, both HW and middleware, needed for creating and deploying successfully such applications on a cloud environment.
- REST – A standard and interface for the operation of web services
- RFID – object tagging
- Software as a Service (SaaS) - software delivery model in which software is provided on a subscription basis mostly hosted on a cloud and follow a pay-as-you-go model
- SPARQL – an RDF query language for semantic linked data
- SPARK – Open, distributed, real-time streams processing engine; used within the NIMBLE data management component.
- User Account and Authentication – The Cloud Identity Management component.

- YAML - human-readable configuration format

Introduction

The roadmap of the NIMBLE project called for an early establishment of a set of core services running as a unified platform, comprising of all open source components that provide the essential capabilities that each NMBLE instance would require. The current document describes the later stages of the platform evolution beyond the core services.

In section 1 the description starts with the technological background of advanced hosting and deployment of the NIMBLE platform. It provides a description of the technology and processes used to achieve a large degree of automation in the platform deployment, resulting in a horizontally scalable set of components and supporting cloud services. Section 2 elaborates on the manner in which the federation of several NIMBLE instances is achieved. Section 3 introduces the blockchain technology and specifically its application in supply chain scenarios.

1 Advanced platform

The advanced version of the NIMBLE underlying platform is tailored to host and connect all the core services and in addition to help integrate value added services that are developed on top of the platform. The platform is designed for and hosted in a secure cloud environment.

A microservices approach was adopted to handle the extreme complexity of the project. This approach ensures autonomy to the different components, to the extent possible, while enabling proper interaction between different components. This approach is especially suited for a distributed development effort, such as is the case in this project, with different groups developing different components independently in different parts of Europe. There is a need to keep such flexibility on the one hand while being able to provide a coherent single running platform on the other hand. Thus, the interfaces for interaction between components are agreed upon but the internal development of each component is left to the group working on it. For example, each component can be developed using a different programming language as long as the interfaces between components are respected. Thus, a popular mode of communication between components is using REST interfaces, which was adopted by NIMBLE. Components usually find each other with the help of a discovery service.

This architectural mode leads to a highly distributed yet not siloed design and development process. Such a design caters to the scale required of the platform with each component being stateless, and able to scale horizontally based on changing load.

For the deployment and orchestration medium the NIMBLE platform relies on Kubernetes (https://kubernetes.io/), which is a container management and orchestration run-time, hosted on a public cloud. Kubernetes enables horizontal scalability of internal components running within it. Kubernetes (aka K8s) is an open-source system for the management of containerized microservices forming applications. It is scalable, efficient, and robust.

1.1 Micro-Services as a guiding principle

Microservices is an architecture form and methodology for breaking up a large complex system into small units, each fulfilling a small, unique, well-defined task, in an independent manner (see Figure 1). Each such microservice can be deployed separately and interacts with its peer microservices using standard communication protocols, such as REST interfaces. Such an

approach eases the task of long-term maintenance and evolution of a large system, when compared to a monolithic system. In addition, internal changes within a microservice do not disrupt any other system component as long as the external contract of the microservice is adhered to. Moreover, each component can be implemented in a different programming language, using different middleware. Thus, collaborative distributed development is made easier.

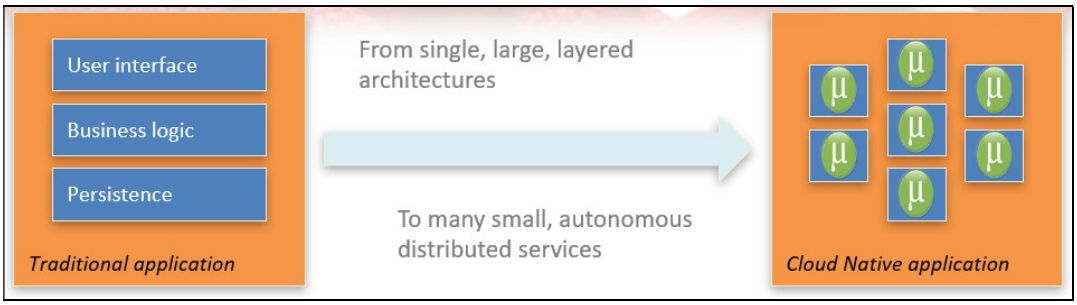

Figure 1: Microservices approach

Such an architecture is distributed in nature thus there needs to be a way for one microservice to find other microservices it needs to interact with, or for the orchestrating application to locate a specific microservice. For that purpose, a service discovery component is deployed. Such a component serves as a central registry of available microservices, responding to queries about the location of a certain microservice. In our deployment, the Eureka[1] service is used for that purpose. In addition, as the NIMBLE components are deployed within a Kubernetes cluster, the built-in Kubernetes Ingress gateway (reverse proxy), is used as well to directly invoke the desired microservice by name.

In Figure 2 we can further see a complete deployment of a NIMBLE instance, comprising of the core services, each running as an individual microservice within a Kubernetes cluster. In addition, cloud-based services are provided in the form of messaging and different kinds of storage to be used by the different microservices for communicating and for storing and retrieving their state.

[1] https://github.com/Netflix/eureka

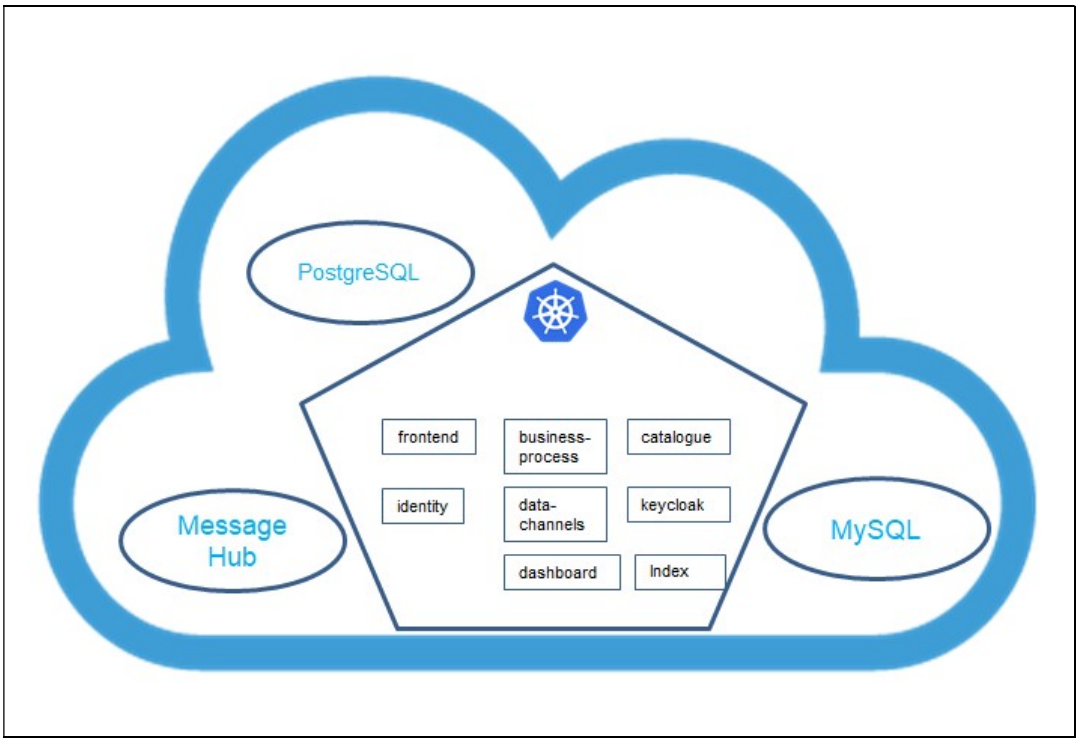

Figure 2: Platform deployment

Every external entity that communicates with the platform or the entities hosted in it needs to go through a routing layer. The chosen Kubernetes routing layer (reverse proxy) is Ingress, which acts as a router and load balancer for the cluster. This router is in charge of maintaining the mapping between the web address provided to external users and the real physical location in which the respective microservice resides. In addition, in the NIMBLE platform Ingress serves as a gateway to the microservices deployed within the cluster. It acts as the public entry point to the Microservice, providing routing, filtering, and load balancing capabilities. Ingress manages access to internal cluster components from the external world. It accepts external calls and based on the requested path it knows which internal service to invoke, further knowing which IP:port is used to serve each service request (see Figure 3).

Figure 3: NIMBLE components as Kubernetes (K8s) services

In a microservices environment, the activity status of different components needs to be checked to ensure that services are up and ready to respond. As the system grows, more microservices are deployed with more instances per each microservice, and the probability for a component to be unresponsive goes up. The Kubernetes based deployment has provisions for the "health checking" of enclosed microservices, including the ability to restart a failing microservice using Kubernetes liveness probe. In a Kubernetes installation failed components will be restarted by the orchestration manager based on the component's stated restartPolicy. The default policy is to always restart a failed component. In addition, horizontal scalability can be achieved by having Kubernetes dynamically adjust the amount of running instances of a microservice based on the load. Internal orchestration mechanisms will further load balance incoming requests among different instances of the same microservice.

Communication among applications is performed directly via REST calls. Logging is performed using the ELK stack[2] (**E**lasticSearch, **L**ogstash, **K**ibana) or through the cloud's native logging tools. A Service Configuration is set up via a combination of Spring Cloud Config[3], and a central Git repository.

Finally, security and access control are the responsibility of the Identity management component which administrates identities on the platform. Keycloak[4] is used as the authentication server for the OAuth2 and OpenID Connect standards. Within the NIMBLE platform KeyCloak is mostly used for security and role-based access control. KeyCloak is an Open Source identity and access management component that can be combined with software packages to provide a higher level of security guarantees.

Figure 2 provides a high-level view of the NIMBLE deployment scheme. As can be seen in the figure all microservices are deployed within a Kubernetes cluster running on a public cloud. Since microservices are stateless by design they use external sources for data management and communication. Thus, a set of cloud-provided services is used by the different microservices. The cloud services include several kinds of data management and storage systems (different

[2] https://www.elastic.co/webinars/introduction-elk-stack
[3] https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html
[4] http://www.keycloak.org/

databases), and a messaging infrastructure. The combination of microservices running in a Kubernetes environment, using cloud services creates a fully horizontally scalable system.

The NIMBLE platform is composed of multiple microservices and is delivered to users according to the Software as a Service (SaaS) paradigm. A recent methodology for building SaaS applications is called *twelve-factor*[5], which defines twelve guidelines for building and running applications in the cloud. Due to their affinity to cloud environments, they are often called *cloud-native* applications. NIMBLE has adopted and was inspired by the twelve factors paradigm.

1.2 NIMBLE Run-time

The NIMBLE platform consists of a set of core services, which are supported by every NIMBLE instance, and a set of optional advanced services. Platform core components represent the main aspects which are required to take a "standard" cloud environment and make it NIMBLE compliant. They provide the core services which are the heart of a NIMBLE platform instance. All services are hosted and deployed on the NIMBLE cloud-based run-time.

The NIMBLE platform distributed run-time hosts all entities needed for the complete operation of the platform, including NIMBLE core and advanced services. In addition, it shall connect all the platform-supplied middleware services such as the discovery and data related cloud services.

The NIMBLE platform run-time consists of a specially customized PaaS cloud, geared towards B2B scenarios. Thus, the daily operations are supported by a cloud environment automating many of the tasks of running such a platform in a scalable and secure manner. This B2B PaaS coordinates and provides interfaces and hooks to additional system components, such as the front-end, to complete the cycle of aiding the users with fulfilling their tasks.

In general, NIMBLE follows a strict separation between applications and services. Applications are stateless microservices, which implement NIMBLE internal business logic. On the other hand, services are used to store certain state and data in a persistent manner. Therefore, applications can easily be scaled horizontally without potential data inconsistencies arising.

An internal cloud service bus for communication is integrated within the cloud run-time environment to provide internal communication and notifications among NIMBLE components.

Applications are deployed within docker containers on a K8s cluster. While several such containers may be run on the same virtual machines (VM). The VMs in which the applications run are stateless and are managed by Kubernetes itself. Thus, an application that needs to save data uses a data storage service such as a database, or NoSQL store. Cloud services are used by platform instances; access to these services is provided using specific secrets which are available inside the K8s cluster and are loaded at run-time to the deployed microservices.

1.2.1 NIMBLE Components Deployment

NIMBLE platform deployment starts from a generic cloud infrastructure deployment, including base services such as security. This deployment is mostly a PaaS which is ready to host generic cloud service such as storage and can host the NIMBLE platform specific services. Once the bare cloud infrastructure is in place the necessary services need to be deployed, namely for storage, communication, etc. Certain services may be deployed directly from the cloud catalogue of services while other less generic services need to be deployed separately, such as

[5] https://12factor.net/

discovery, and gateway, monitoring. Next, the cloud service bus using cloud provided messaging services is deployed. At the last stage, the NIMBLE applications are deployed, their Kubernetes related configuration manages the actual deployment and access information, and in addition they may register with a service discovery component such that the connection between different applications can be established.

1.3 Cloud services used by the platform

The NIMBLE cloud-based deployment is divided between the individual stateless microservices which are deployed on a Kubernetes cluster running in a cloud environment and cloud-based services which provide generic capabilities which can be used by the individual microservices. In this section we describe the most significant cloud services used by NIMBLE components.

1.3.1 Data management

The data management component deals mainly with data storage and different flavours of retrieval. These encompass different databases (relational and NoSQL), as well as file systems. Moreover, data-based notifications can be supported by connecting between cloud-based communication and data services.

First and foremost, these services enable ingesting heterogeneous data types into the platform. Based on the requirements of the entity connecting a data source, data flowing into the platform can take several routes. The simple one is for incoming data to be stored for potential use in the future for many purposes, including offline analytics. Offline analytics enables the analysis of offline data of different types. Furthermore, data can be passed through a real-time pipe for performing transformations on the data including filtering, potentially targeting notification, and finally storage of transformed data items.

A list of tables in the main NIMBLE DB can be seen in Figure 4

```
postgres=# \l
                                  List of databases
      Name        |  Owner   | Encoding |  Collate   |   Ctype    |
------------------+----------+----------+------------+------------+
 binarycontentdb  | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 businessprocessdb | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 camundadb        | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 catalogcategorydb | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 catalogsyncdb    | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 datachanneldb    | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 identitydb       | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 keycloak         | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 modalmldb        | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 postgres         | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 template0        | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
                  |          |          |            |            |
 template1        | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
                  |          |          |            |            |
 trustdb          | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 ubldb            | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
(14 rows)

postgres=#
```

Figure 4: Tables in the main NIMBLE DB

The following microservices make use of a PostgreSQL DB: Identity, business process, catalogue, data channel, trust, and keycloak. The Track & Trace component makes use of a MongoDB instance hosted on the cloud.

1.3.2 Communication Bus

The main purpose of this component is to provide generic communication capabilities among different NIMBLE components and participants. It can be used to send messages and notifications among components or to share information among various entities. The dominant paradigm is the publish/ subscribe pattern leading to event-based communication among collaborating partners by registering interest in particular events. Event-based communication is often used in microservices based architectures as it decouples producers and consumers in terms of location and time (asynchronous communication). Using this paradigm producers and consumers do not have to be aware of each other and need only to agree on the topic via which they are going to communicate, and the message format of the agreed upon topic. NIMBLE employs open source tools such as Apache Kafka for the Messaging and Communication Framework. The message bus is deployed as a service over IBM's public cloud.

The communication bus will be configured, upon deployment, with the necessary set of topics as agreed upon between the different components. In addition, the message structure of each message in each topic will be agreed upon and documented by the cooperating components. The communication bus needs to support the number of different topics required for a NIMBLE installation, along with the associated aggregated throughput in all topics.

The communication bus is realized by using an instance of a MessageHub service, deployed in IBM's cloud. The back-end is based on a Kafka cluster, and the interaction with the service is realized using standard Kafka clients.

The following microservices make use of the communication bus: Identity, business process, catalogue, and trust. Microservices communicate with each other to exchange information which is pertinent to more than a single microservice, for example when company or trust related information was changed.

1.4 State-of-the-art System development deployment and integration

As mentioned above, due to the distributed approach of the design and implementation of the NIMBLE platform, in which different partners develop independently and maintain ownership over different components of the system, combined with the complexity of the platform, which is comprised of many different components, we opted to follow a microservices approach to make the entire process of design, implementation, and integration more manageable, in a distributed manner.

1.4.1 Technical infrastructure

After having presented the high-level design for microservices support and guiding principles of the NIMBLE platform, we turn to deployment and hosting issues, focusing on the provisioning and operation of the infrastructure, on which the NIMBLE platform relies. This includes the internal services, business services, associated repositories and external entities.

The platform components are hosted on a Kubernetes cluster managed on, a public cloud, as can be seen in Figure 5.

Currently the cluster is composed of two worker nodes, and it may grow based on evolving system needs. The main characteristics of the worker nodes can be seen in Figure 6. The K8s cluster is divided into 3 namespaces; the one used for NIMBLE deployment is the "prod" namespace. The Prod namespace for deployed Applications operates behind Ingress (a reverse Proxy). This namespace hosts all the platform microservices.

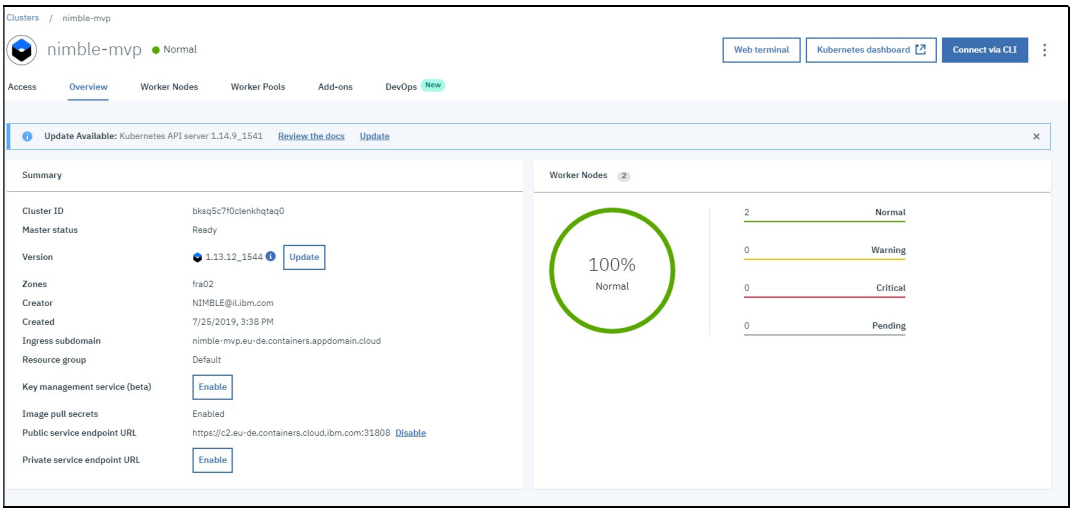

Figure 5: NIMBLE Kubernetes cluster

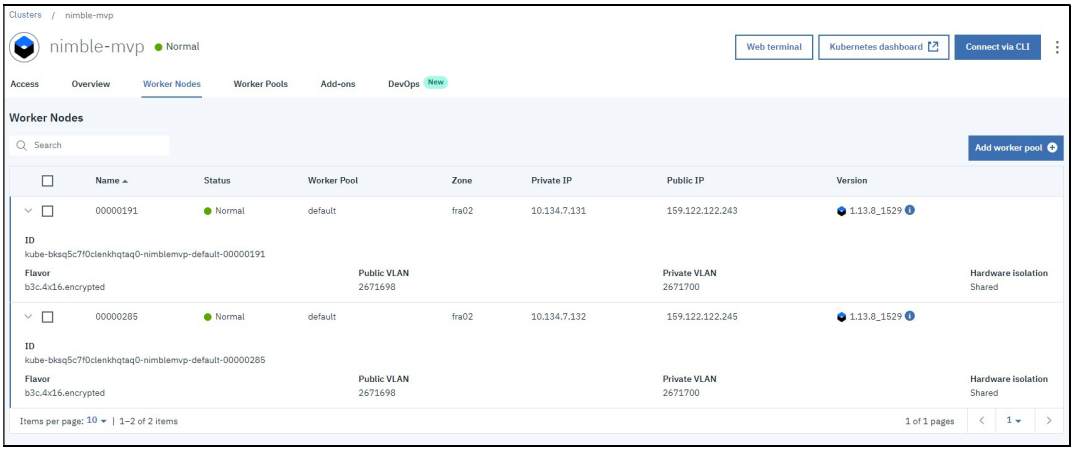

Figure 6: NIMBLE worker nodes

In addition, there are cloud services which are used by NIMBLE components as can be seen in Figure 7. As can be seen in the figure there are various instances of relational databases which are used for example by the catalog related components. There is an instance of a messaging service which is used by the data channels component. These services are hosted on IBM's public cloud and offer binding capabilities to all NIMBLE components.

Cloud Foundry services (9)

Compose for MongoDB-NIMBLE-TT	nimble-de / dev-de	Frankfurt	Compose for MongoDB	Provisioned
Compose for MySQL-jq	nimble-de / dev-de	Frankfurt	Compose for MySQL	Provisioned
Compose for PostgreSQL-Categories	nimble@iL.ibm.com / dev	London	Compose for PostgreSQL	Provisioned
Compose for PostgreSQL-y7	nimble-de / dev-de	Frankfurt	Compose for PostgreSQL	Provisioned
Log Analysis-m9	nimble-de / dev-de	Frankfurt	Cloud Foundry Service	Provisioned
Message Hub-so	nimble@iL.ibm.com / dev	London	Event Streams	Provisioned
Monitoring-oi	nimble-de / dev-de	Frankfurt	Cloud Foundry Service	Provisioned
NIMBLE-FMP-PROD	nimble-de / dev-de	Frankfurt	Compose for PostgreSQL	Provisioned
Performance Test (Suat)	nimble-de / dev-de	Frankfurt	Compose for PostgreSQL	Provisioned

Figure 7: NIMBLE cloud services

In Figure 8 we can see some of the deployed microservices, along with a measure of the resources being used by the running components.

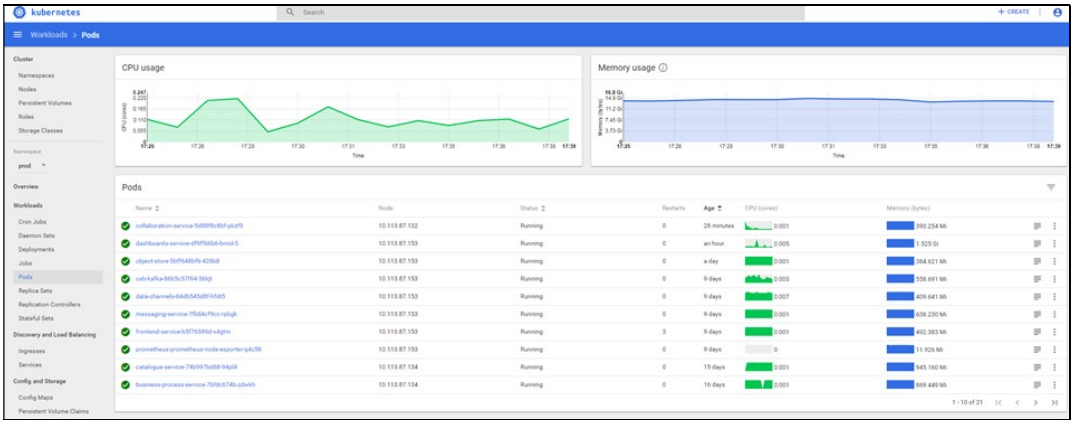

Figure 8: Deployed microservices

1.5 Integration, deployment, and the Continuous Integration (CI) toolchain

We have created a completely automatic Continuous Integration toolchain, such that changes to project code, residing in a project-wide GitHub repository (see Figure 9), result in an automatic build and deployment of a new version of the microservice within the K8s cluster.

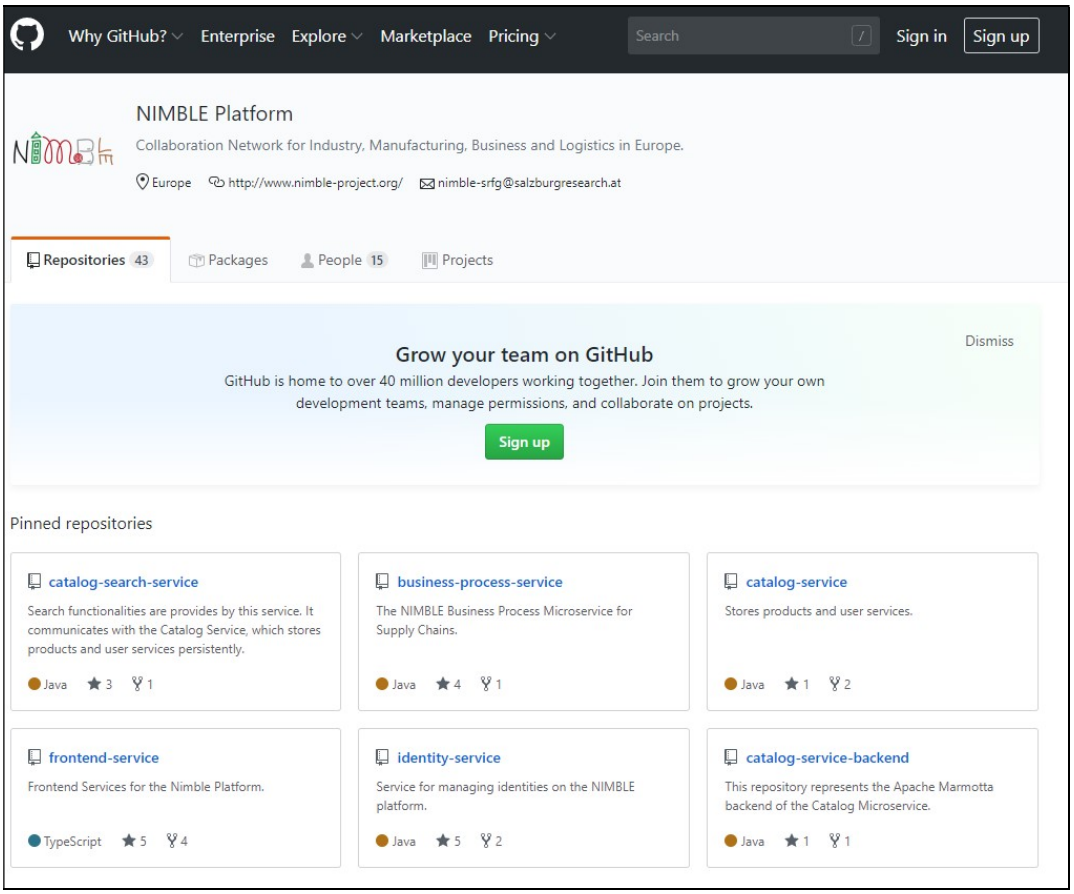

Figure 9: NIMBLE GitHub repository

The Continuous Integration (CI) environment is comprised of the following components

1. GitHub repository: all components should have a repository under the NIMBLE project (https://github.com/nimble-platform).
2. Docker – a docker image is created for each component.
3. Jenkins: build a new version of a microservice based on code commit and deploys the new version of the service to the Kubernetes cluster.
4. Kubernetes - managed cluster on which all components are deployed
    a. Requires specific Kubernetes configuration for each component

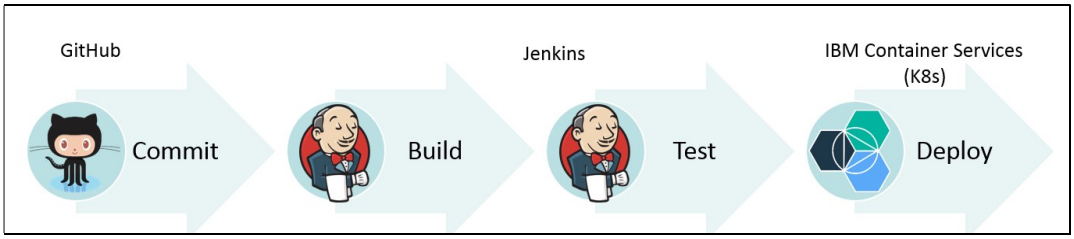

Figure 10: CI workflow

The automated workflow kicks in upon a new commit to the master branch in the project github repository. Jenkins keeps track of such changes via web hooks, and initiates the procedure as specified in the JenkinsFile. The standard procedure is to build the component using the dockerFile, and if no errors reported, to deploy the docker image on a docker container running in the Kubernetes cluster, using the specified kubernetes configuration. This process happens for every repository for which there exists an associated JenkinsFile.

A more elaborated pictorial view of the CI workflow can be seen in Figure 11.

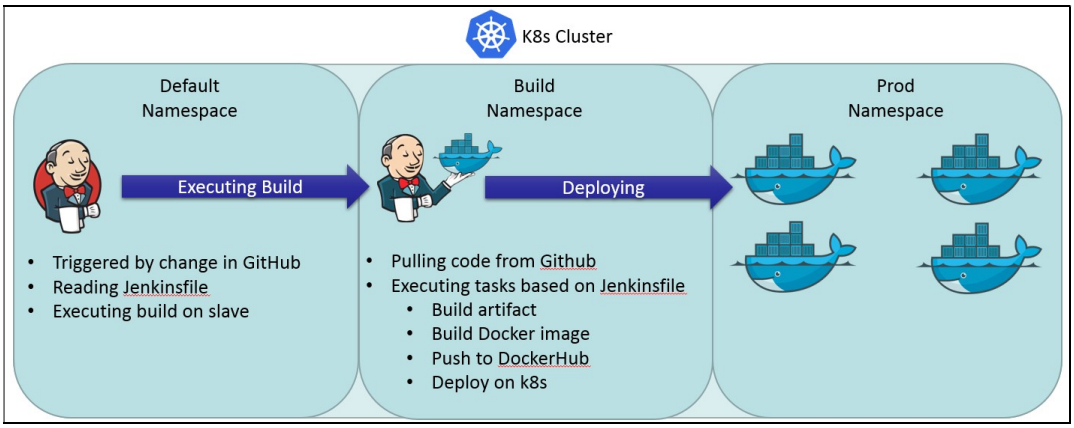

Figure 11: CI detailed

2 NIMBLE Federation

A NIMBLE instance provides a set of core services that enable end-users to engage in business collaborations. These services include, for example, the ability to publish and search product catalogues, to conduct negotiations, manage supply chains, and execute business processes. A NIMBLE provider can take the open source infrastructure and bundle it with sectorial, regional or functional added value services and launch a new platform instance. Such specializations may take place at the industry level, namely adding specific capabilities for a specific industry, or at a regional level for addressing specific requirements of a specific country or geographical region.

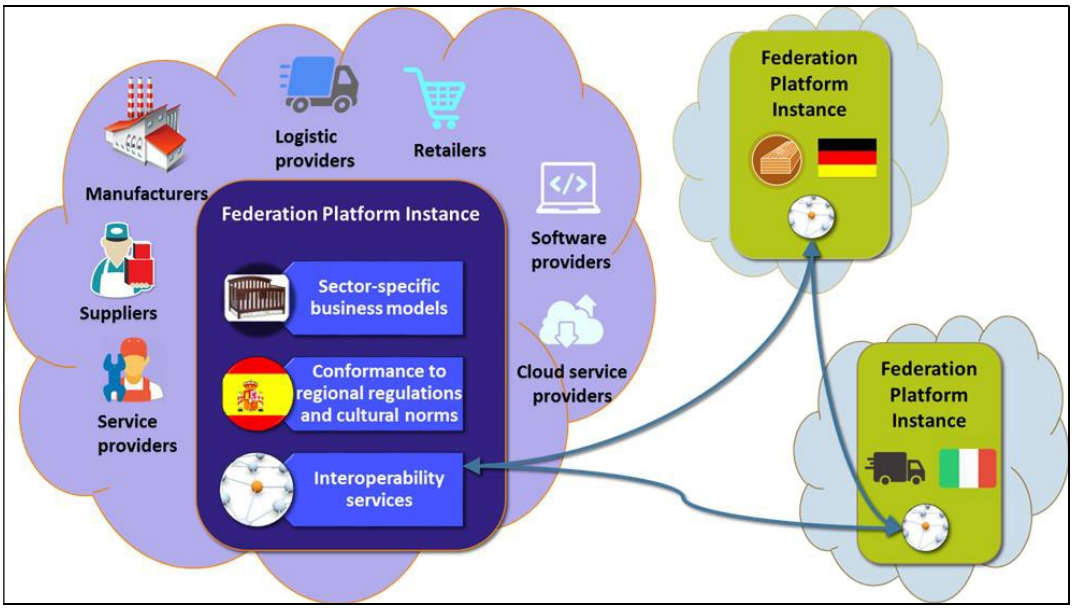

Figure 12: A set of specialized NIMBLE instances, federated for collaboration

The vision is to enable a federation of NIMBLE instances, such that end users belonging to different NIMBLE instances may engage in B2B operations.

The federated platform concept enables the specialization of a fixed set of capabilities for requirements stemming from different sectors or regions. NIMBLE aspires to a federated yet interoperable eco-system of platforms that provide B2B connectivity. Such a common, yet federated infrastructure opens the door for multiple platform providers, with a diverse set of platform instances that still can collaborate.

2.1 Federation Requirements

In addition to the major requirements from every NIMBLE instance, federation presents several additional requirements.

- The ability to discover and find additional NIMBLE instances to collaborate with.
- The ability to invoke the core services provided by one NIMBLE instance by end-users of another NIMBLE instance, for example to search for a specific product across various NIMBLE instances.
- The ability of each end user to select whether a service, catalogue, or product is exposed outside the scope of its local NIMBLE instance
- The ability of an end-user to decide whether an action on his behalf would be local or federated. For example, conduct a search operation only on the local instance or in all instances available via the federation.
- The ability of the administrator or governing body of a NIMBLE instance to decide with which other instances to federate.

2.2 Architecture High Level View

The main elements of the federation architecture are the Federation Core Service and the federation Delegate. The Federation Core Services" are placed outside of all participating NIMBLE instances and contain general management support for the federation. The Federation Delegate is placed at the edge of each NIMBLE instance representing that instance in the federation.

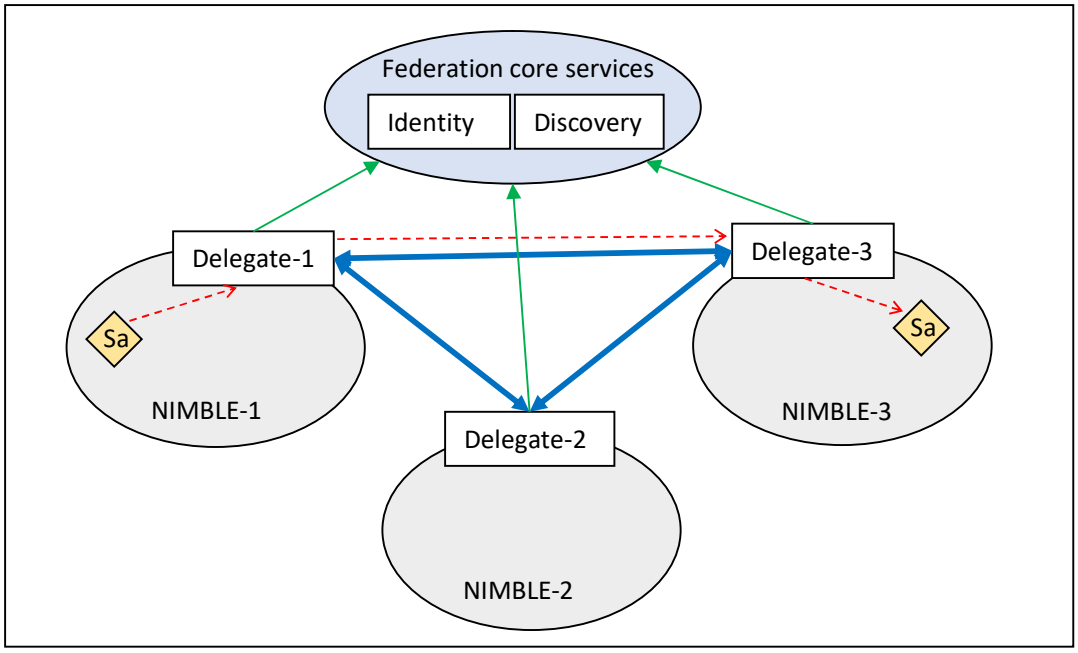

Figure 13: Federation high level view

The Federation Delegate is an entity identified, registered and deployed within the instance it belongs to. It is also identified and registered in the Federation Core Services. It controls the communication and collaboration patterns between local entities (users and services) and external instances. The delegate obtains information from the federation core services concerning its counterparts in other NIMBLE instances and proceeds to invoke the respective operations directly on the delegate belonging to the nimble instance it wishes to collaborate with.

The federation core services allow the delegates of different instances to discover and find each other and engage in secure communication and safe transactions. The federation core services use as building blocks components that exist in a NIMBLE instance; for example, the identity service and elements of the security mechanisms are reused, and therefore allow the delegates to perform role-based access control on other delegates seeking to invoke local services.

The delegate of one NIMBLE instance communicates with the delegate of a second NIMBLE instance, not directly with the local services of the second instance. The interaction is performed using a RESTful API. An end user or local service will contact its local delegate, which will contact the remote delegate of a remote NIMBLE instance, and that delegate will forward the request to its local services. Responses will be sent back, following the reverse path, to the imitating entity. Responses will be combined by the local delegate that initiated the call, and aggregated results shall be passed to the front-end to be presented to the end-user.

Security and privacy are the main reasons that we choose to place a delegate at the edge of each NIMBLE instance and allow it to communicate only with the respective delegate of a remote NIMBLE instance. This way, an internal service or entity only communicates with a local trusted component. Each instance administrator can easily configure and enforce what information stays within the instance and what is permitted to be exposed to the federation. We envision the delegate as a configurable gatekeeper that is used to protect the data of an instance, in addition to its role of connecting it to the federation. The requests flowing in the route service->delegate->delegate->service carry the bulk of the data that flows between instances, and therefore constitute the federation "data-plane". It consists mainly of API calls forwarded back and forth between instances.

The federation core services facilitate the connectivity realized by the delegates and controls what roles are granted and which access patterns are allowed. The interaction between the federation core services and the delegates can therefore be called the federation "control-plane".

This architecture is analogous to the architecture of ISTIO[6], an open platform to connect, manage, and secure microservices. Istio has an entity called "Envoy" which is analogous to our delegate, that realizes the data-plane between services and mediates the traffic between services. Istio also has a control-plane, which controls the connectivity patterns of the Envoy, much like the federation core services control the delegates. This architectural pattern is also pervasive in Software-Defined-* (e.g. software defined networks (SDN)).

2.3 Federation Components

The federation components represent the main aspects which are required to take a "standard" NIMBLE instance and allow it to federate with other NIMBLE instances. The first aspect is deploying the federation core services. The second aspect is deploying a delegate in every NIMBLE instance. The delegates are then registered to the federation core, and an administrator configures the roles of each delegate. Some extensions to the capabilities of local services are recommended, for example tagging each item, service or capability with a "scope" of whether externalizing it in a federation is permitted.

There are manual steps which are required before a federation is created. These are not technical steps but rather managerial and organizational steps. First, there needs to be an entity which is interested in establishing a federation of NIMBLE instances. That entity shall become the manager and administrator of the federation. Second, there need to be NIMBLE instances which are interested in joining the federation. Once the different entities are in place they can create a consortium which shall collaborate within a federation. The rules governing the federation consortium are out of scope of the current work and are left for the different entities who wish to collaborate to decide. A technical requirement for NIMBLE instances to join a federation is to support the APIs of the NIMBLE core services. Once the agreement is in place the federation manager can spin up the federation core services and provide the required configuration to the individual NIMBLE instances wishing to join. The individual NIMBLE instance will in turn register itself with the federation and the collaboration can begin. This process is somewhat analogous to a new company that wished to join a NIMBLE instance.

[6] ISTIO, An open platform to connect, manage, and secure microservices. https://istio.io/

2.3.1 The delegate

The delegate is a state-less component that hides the internals of a local NIMBLE instance from remote incoming traffic and mediates and routes the outgoing traffic on behalf of local entities. It has an identity (at least one, see below) in the local instance.

Local entities will be able to find the delegate using the platforms service discovery component and direct a request to it, for example a search request for discovering a certain class of products. The delegate will check the role of the entity doing the request and forward that request to a remote delegate (or delegates), while assuming the role of the requestor (role definitions should overlap in federated instances). The remote delegate will forward the request to the respective search service in its local instance, provided that the role of the requestor permits that. Responses will be sent back in the reverse path.

The API of the delegate will be a subset of the APIs of internal core services. We envision a configurable delegate, in which the instance administrator can choose which subset of API calls are supported by a delegate.

The communication between delegates – the data plane – will carry the bulk of cross NIMBLE traffic. Since the delegate is stateless, it is possible to run several delegate processes in parallel in order to address scalability concerns. As delegates are deployed within a NIMBLE instance as an additional microservice they can benefit from automatic scaling capabilities of cluster orchestrators such as Kubernetes.

2.3.2 Federation core services

The federation core services are a collection of entities that together allow NIMBLE instances to federate and end-users to collaborate. The federation core services are deployed in the same style as a NIMBLE instance – as a cloud ready collection of services within a Kubernetes cluster. Once deployed, these services will provide a set of APIs that allow the federation core to provide the following capabilities:

- Allow NIMBLE instance administrators to register their platform instance for federation.
- Allow NIMBLE instance administrators to define identities for delegates to their respective platforms.
- Allow a delegate to register and identify itself as a representative of an instance.
- Allow delegates to search for other instances and discover the delegates that represent them.
- Provide security mechanisms that allow delegates to securely communicate.
- Provide security mechanisms that allow delegates to exert role-base access control on (1) the type of operations they are willing to accept from internal entities and forward to remote instances, and (2) the type of operations they are willing to accept from remote instances and forward to internal entities.

As much as possible, the federation core services will be implemented using building blocks that are already in use in a NIMBLE instance. For example, we expect the Identity Service to play in the federation core a similar role as in a NIMBLE instance.

The identity service

The identity service is used to establish an identity for a NIMBLE instance and an identity (or identities) to the delegates that belong to that platform. This is analogous the way identity management within an instance establishes an identity for a company and users that belong to that company.

The discovery service

The discovery service allows delegates to discover other live delegates. A delegate connects to the discovery service and publishes its identity, affiliation, and address. As long as it remains connected it will be kept as "alive" in the discovery service. Other delegates can query the service for all live delegates. In a sense this is a membership service for delegates.

2.3.3 Extensions to local services

Services that reside within a NIMBLE instance need some minor extensions to be federation ready. At a minimum, they need to include in their description and role-based security profile whether they are permitted to engage in federation-wide activities.

2.4 Representative Flows

2.4.1 Deploy

A federation manager (administrator) deploys the federation core services: identity and discovery. It then provides the address of the federation core services API to NIMBLE instance administrators. Services are deployed in the same style as NIMBLE core services – cloud ready decoupled state entities.

2.4.2 Enrolment and Registration

As can be seen in Figure 14, a NIMBLE instance manager submits an enrolment request to the federation core, and the request is evaluated by the federation governing body or automatically. If approved the joining instance administrator generates an identity for the delegate that represents the instance. The administrator then deploys the delegate, providing it with the address of the federation core API, and proper credentials. The delegate connects to the discovery service, registers itself, and from that moment on it is discoverable as a delegate of the respective instance. Conversely, it can discover the delegates of other NIMBLE instances within the federation.

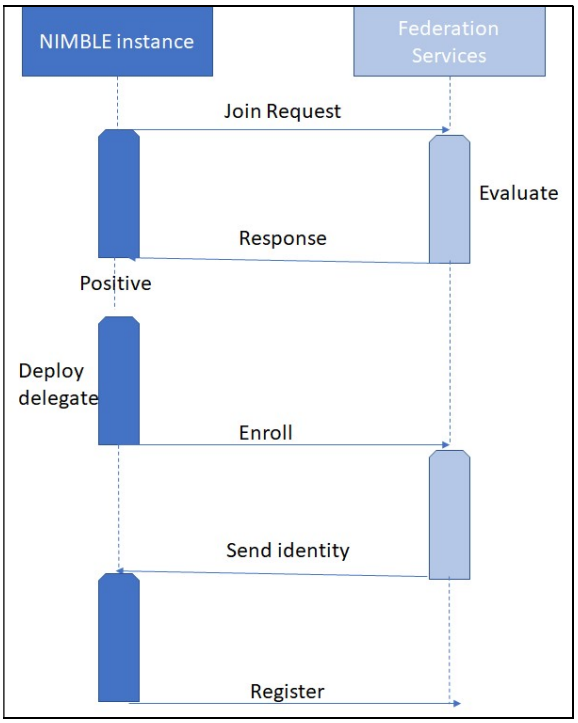

Figure 14: Join a federation

2.4.3 Search

A user within a NIMBLE instance wants to search outside its instance for a certain type of service. The user ticks the "federation" box within the search component provided by the NIMBLE front-end. The front-end will discover the local delegate using the local discovery service and will direct the search call to the local delegate. The local delegate shall turn to the local search component and invoke the corresponding query on it. In parallel, the delegate shall query the federation code services to obtain a list of all available delegates of other NIMBLE instances in the federation. Note that such a list may be cached by the local instance with periodic refreshes from the federation core services. Each delegate receiving the search request shall invoke the query on its local search component and shall send back the results to the initiating delegate. The delegate at the initiating instance shall combine all the results received, locally or remotely, shall add an identified as to the originating instance and send back the combined set of results to the local front-end.

2.4.4 Catalogue Service

Search results retrieve the basic information concerning the relevant product or services. The user can drill down to obtain more information on specific products or services, by clicking on the item of interest in the front-end. For local products the front-end turns to the catalogue service which returns the desired information. For remote products, namely products that were brought in from another NIMBLE instance there is a need to query the remote catalogue service. To realize this capability the federation aspects come into play. The call which in the local case targets the local catalogue service, turns instead to the local delegate. The delegate, using the mechanisms described above of the core federation services, locates and contacts the remote delegate service on the corresponding NIMBLE instance. The remote delegate invokes the call

on its local catalogue service and the response is sent back to the originating delegate. The originating delegate in turn sends the responses back to the local front-end to be presented to the interested end-user.

2.4.5 Business process

To achieve cross NIMBLE instances business processes, we require synchronization of the relevant state across the participating instances. There are two main options for keeping the remote instance synchronized with relevant business processes, namely using a single business process engine, or synchronizing engines on both instances (Camunda is used as the business processes engine). We opt for supporting federated business process using a single engine, at the seller's platform.

Federated business process using a single engine

Parties from two instances that wish to execute a shared business process shall do so utilizing a single underlying business process engine. The engine of the seller instance shall be used, and the counterpart will interact with the business process engine remotely via the delegate service. Within a single NIMBLE instance at each step of the business process there is a party which should send a message, and the other party which is waiting to get a message. Sending messages is done via the Camunda API, while the receiver side polls the Camunda engine to obtain new messages targeted to it. The federated case will be handled in a similar manner, having the seller side operate regularly (locally), while the buyer side will do it remotely using the previously described delegates mechanism.

After initiating the process in a particular business process engine, triggering events on it follows the same path as search requests: end-users local to the business process call it directly, whereas remote end-users call their delegate, which forwards their calls to the delegate local to the business process, which in turn invokes the business process.

Thus, the entity in the instance which is using a local Camunda engine will invoke the API directly, while the remote instance will invoke the API remotely using the delegate processes on both instances. Entities polling for information will have to poll all instances in the federation to discover the updated information concerning their active business processes across all NIMBLE instances in the federation. The NIMBLE dashboard for business processes is populated in that manner. Thus, when taking another step in a business process and sending a corresponding message, the identity of the remote counterpart NIMBLE instance is known, such that the remote call can be targeted towards one specific instance delegate. When populating the dashboard with the updated state of all business processes, that call shall be distributed to all members of the federation, using the delegate service, to obtain complete and updated information.

Federated business process using local engines

This option explored the possibility for each instance to operate just like in the single instance case by using its own local Camunda engine. For that to work we need to keep the Camunda engines of both NIMBLE instances in sync. To achieve that we would require an API call invoked on the local Camunda instance to be invoked also on the Camunda engine of the second instance by utilizing the delegates.

This option was evaluated as well but was discarded in favour of the solution described above. The main rationale behind this choice is that keeping both instances of a business process engine in synchronization, taking into consideration corner cases (such as one of them being down for a while) is a complicated task. Using the solution above reuses a lot of the work that has been done already to support federated search and catalogue service.

Invoking APIs will always be local. Once the call has been invoked, a similar one shall be invoked on the local delegate, which in turn will invoke it on the remote delegate, which in turn shall invoke it on the remote Camunda engine. Polling for retrieving the current state of business processes for an entity shall always be a local call invoked on the local Camunda engine.

2.5 Access control

The federation architecture was designed with security and flexibility in mind. We therefore choose to place a delegate at the edge of each NIMBLE instance and allow it to communicate with the respective delegate of a remote NIMBLE instance. This way, an internal service or entity only communicates with a local trusted component, and all security filters are enforced in one place. The control plane that is headed by the federation core services augments the delegates with central control of all security measures and activity patterns.

The controller and control plane architecture provide flexibility as well. Role based access control together with selective publication of capabilities imply that a NIMBLE instance administrator has complete control on what is exposed to the federation.

3 Blockchain in NIMBLE

3.1 Blockchain essentials

A blockchain revolves around the concept of a shared ledger, representing the system of record and a single source of truth for business interactions. The shared ledger, at the heart of the system, is maintained by a cluster of peer processes, belonging to different organizations, providing an append only transactions log, while guaranteeing the immutability of inserted and validated transactions. The blockchain enables a network of business partners to perform transactions across organizations without resorting to a single unified trusted authority. A blockchain transaction represents a state change or asset transfer in the ledger; transactions are governed by smart contracts, which contain the rules for transactions to be invoked and the agreed upon resulting behaviour. Blockchain provides a shared, replicated, permissioned ledger ensuring trust, provenance, immutability and finality, to replace inefficient, expensive, and vulnerable processes.

These measures together provide a level of trust among partners which is difficult to achieve otherwise in an inherently trust-less and distributed environment. Most importantly, the trust is not due to a single actor within the network, but rather it is an outcome of the collective nature and properties of the underlying technology. Transactions through the platform are recorded in a final and immutable manner by the blockchain, providing all network members with an identical and trustworthy real-time view of the state. Validated transactions in a block in a ledger cannot be modified or deleted without leaving a noticeable trail.

As can be seen in Figure 15 the shared ledger provides a real-time common and replicated view of the state of the transactions among all members of a blockchain network. This reality stands in contrast to the pre-blockchain era in which each organization held its own ledger, opening the door to inconsistencies and disputes.

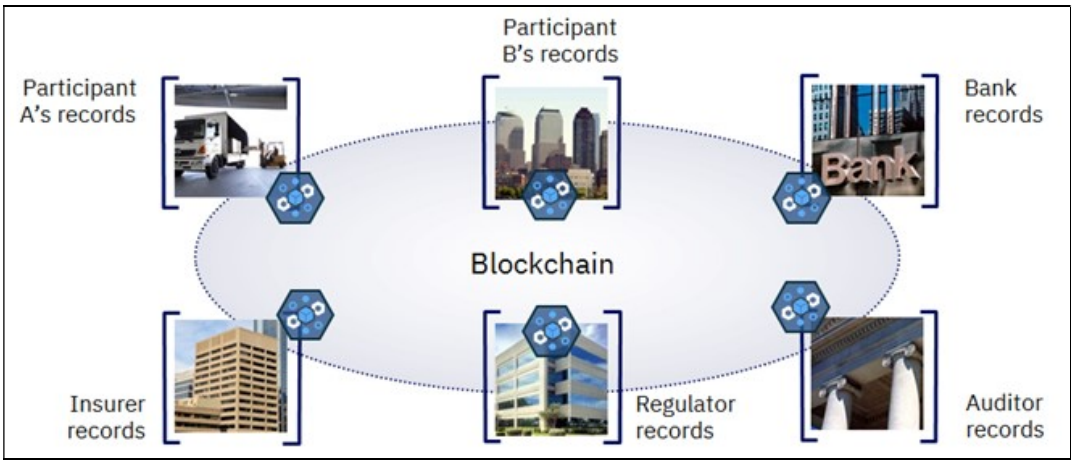

Figure 15: Blockchain's core - the shared ledger

The four cornerstones comprising the blockchain structure are a shared ledger, transaction verification by network members, smart contracts, and security & privacy measures. All these building blocks combined provide assurance for consensus, provenance, immutability, and finality. These capabilities lay the foundation for a blockchain platform for enterprises as can be seen in Figure 16.

At a high level, the system is comprised of peers belonging to different organizations, which replicate and validate the blocks comprising the ledger; an ordering service which determines the order of the transactions and publishes the corresponding blocks; and a client that interacts with the system for invoking transactions or queries. A sub-set of the peers is involved also in endorsing transactions submitted to the system; supporting consensus for inserted transactions. All entities hold verifiable security certificates issued by a Certification Authority.

Blockchain technology usage is relatively new but interest in it is growing in many fields. The first such field is the financial services arena, but more areas are exploring the usage of this technology, supply chain being in the forefront. In various analysis reports it can be seen that Banking / Financial Services and Supply Chain remain top industries for blockchain activity[7]. A lot of attention and funds are being devoted to exploring blockchain contribution to supply chain scenarios[8], both by industrial partners, as well as large IT providers, such as IBM, Oracle, and Microsoft.

[7] https://www2.deloitte.com/content/dam/Deloitte/cz/Documents/financial-services/cz-2018-deloitte-global-blockchain-survey.pdf

[8] https://www.tradelens.com/
https://www.coindesk.com/pwc-australia-port-of-brisbane-unveil-blockchain-supply-chain-pilot
https://www.zdnet.com/article/alibaba-pilots-blockchain-supply-chain-initiative-down-under/
https://www.forbes.com/sites/bernardmarr/2018/03/23/how-blockchain-will-transform-the-supply-chain-and-logistics-industry/#748fab1e5fec

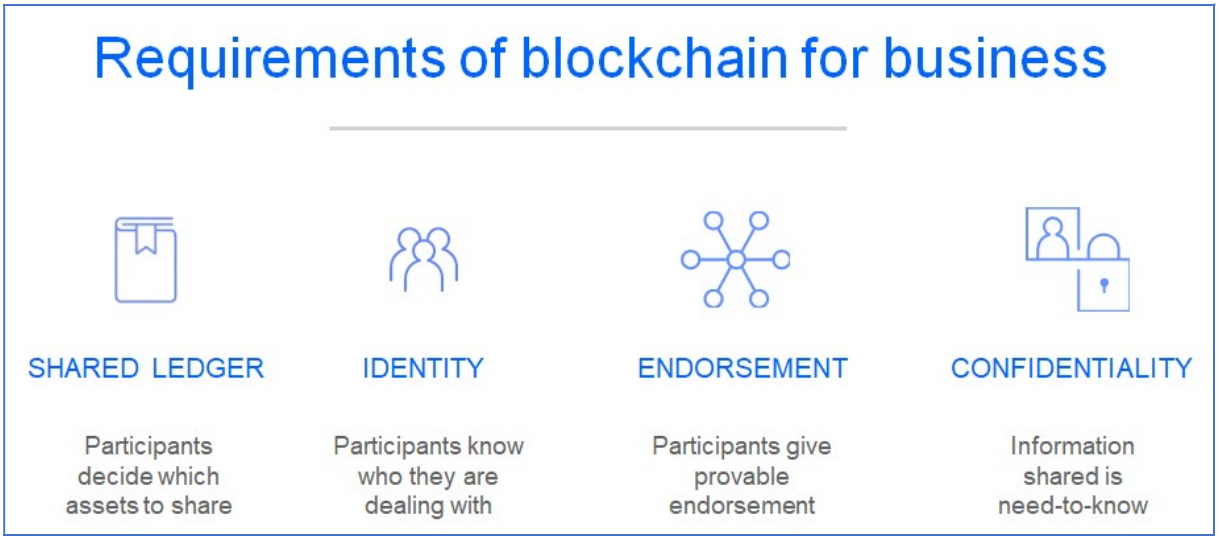

Figure 16: Blockchain essentials

3.2 Blockchain for supply chain

Blockchain solutions are prominent in business relationships which require data to be shared between different entities. Such data may be needed in real-time or close to it, or as a trace of past transactions to be used in the future. Companies involved do not, however, necessarily have trust in each other. Such relationships are prevalent in supply chain networks. The use of a blockchain based infrastructure enables parties which are a part of a supply chain relationship to leverage the technology to gain tangible benefits in important areas such as reduction in time, money, and risk. The blockchain serves as the single source of truth, which is shared among all participants, and is not controlled by a single entity.

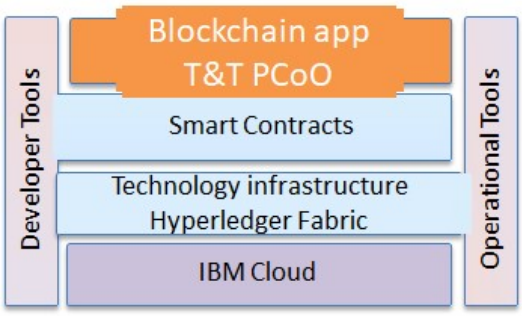

Figure 17: Technology Layers

In Figure 17, a high-level view of the technology layers involved is depicted. At the bottom resides the blockchain infrastructure itself, which consists for example of consensus mechanism, cryptographic validation, mechanisms for replication of blocks, and certificate authorities (CA). In our current implementation and deployment all these components are hosted and deployed on the IBM cloud (based on Hyperledger Fabric[9]). At the middle layer reside the elements which

[9] https://www.hyperledger.org/projects/fabric

enable developers to insert specific logic which shall be tightly coupled to a specific deployment of a blockchain. This layer encompasses the specific rules governing the interactions supported for a specific network of participants. This layer is mostly associated with and implemented by Smart Contracts (called chaincodes in Hyperledger fabric). Chaincodes interact with ledger state implementing business logic of network members. At a higher layer resides the solution or application, which serves as a mean to connect the blockchain with the business processes of companies, be it via interaction with end-users, or digital processes and devices operating on their behalf.

As a part of the vision to incorporate a blockchain based supply chain platform, the first step is to incorporate partners data of all sorts, be it machine generated data or a part of a business process involving a human in the loop. The data serves as the driving element to the agreed upon logic which resides in the blockchain level as well in the form of smart contracts. Examples as to the kind of data includes data coming from IoT devices which enables tracing the location and state of items in real-time to drive potential notification on out-of-bounds conditions affecting agreements.

Such capabilities enable a coherent and updated view of the status of the supply chain ecosystem including current stage of an item in the production floor and the associated state as can be reported by attached IoT devices; all according to the scope, rules, and conditions agreed upon among the network partners.

The blockchain infrastructure can be used to reduce the rate of disputes and errors in logistics and to enable real-time tracking of transactions in the supply chain providing elevated accuracy, security and speed; while ensuring that data and interactions are not made visible to unauthorized partners. Moreover, full traceability and provenance of business processes execution is supported by the blockchain infrastructure.

3.2.1 Advantage of Blockchain based scenarios in the area of supply chain

In general, several benefits can be obtained by applying blockchain based scenarios in the area of supply chain.

- Enhance trust in a trustless environment – providing end-to-end provenance. Blockchain based supply chain relationships can become a validated, trusted, self-executing process, supporting non-repudiation.
- Tie in fragmented and siloed systems - A shared ledger can remedy this situation by providing a unified view to all participants at the same time, which can be accessed using the same interfaces to the same underlying system. This provides a clear picture for making decisions to all involved entities.
- Minimize disputes – Having a single source of truth, verifiable and auditable, can lead to a reduced number of disputes, and a shorter time to resolution of existing disputes.
- Data integration, including IoT, can lead to greater transparency and better, more efficient, collaboration by taking actions programmatically and automatically based on incoming data. Provide the capability to track, monitor, and report the location and status, of shipments, goods, or supplies with the integration of IoT devices. Provenance of each component part in a complex system is hard to track but is of great value, especially as items can be combined or be contained. Such information may include the manufacturer, production date, batch and even the manufacturing machine program. Moreover, producers and end users require transparency on where and how their raw materials and sub-contracted products and supplies are made. Some governments

require more information about corporate supply chains, with penalties for non-compliance. In such a case blockchain enables the safe digital transfer of material and goods end to end, across the supply chain. That information includes which party had ownership to what part at what time, and what changes were performed.

- Automating contracts and processes - Terms of a contractual agreement between parties can be manifested as a smart contract running in the blockchain. For example, a buyer wants an efficient way of converting a purchase order into validated, self-executing contract updated to reflect the status of the supply.
- Differential visibility and data privacy ensure that the information is shared only among the intended partners.

3.3 Blockchain use in NIMBLE

3.3.1 Blockchain roles in the overall platform architecture

The blockchain platform is developed and deployed as an added value service to the NIMBLE platform. It consists of the blockchain network itself and the NIMBLE based applications which interact with it. The blockchain component exposes interfaces to services that the different higher-level components of the platform can use. For example, the Tracking and Tracing component uses specific REST calls which are exposed via the blockchain component. Thus, different platform components use interfaces exposed by the blockchain component in order to take advantage of the capabilities and promises of a shared ledger. The blockchain component exposes a REST interface to other platform components. The interfaces are grouped according to functionality provided, namely T&T and Preferential Certificate of Origin related interfaces. There are two main kinds of actions supported by the interfaces:

1. *Invoke smart contract transactions* – intended for NIMBLE modules to be able to invoke transactions residing in smart contracts. These transactions are invoked to change the state of an entity and record that for posterity. For example, in the T&T scenario it can represent an item passing to a new stage in the production floor.
2. *Query* - expose query capabilities to retrieve data stored in the blockchain. For example, the in the Preferential certificate of origin case, the hash of associated with a specific order can be retrieved.

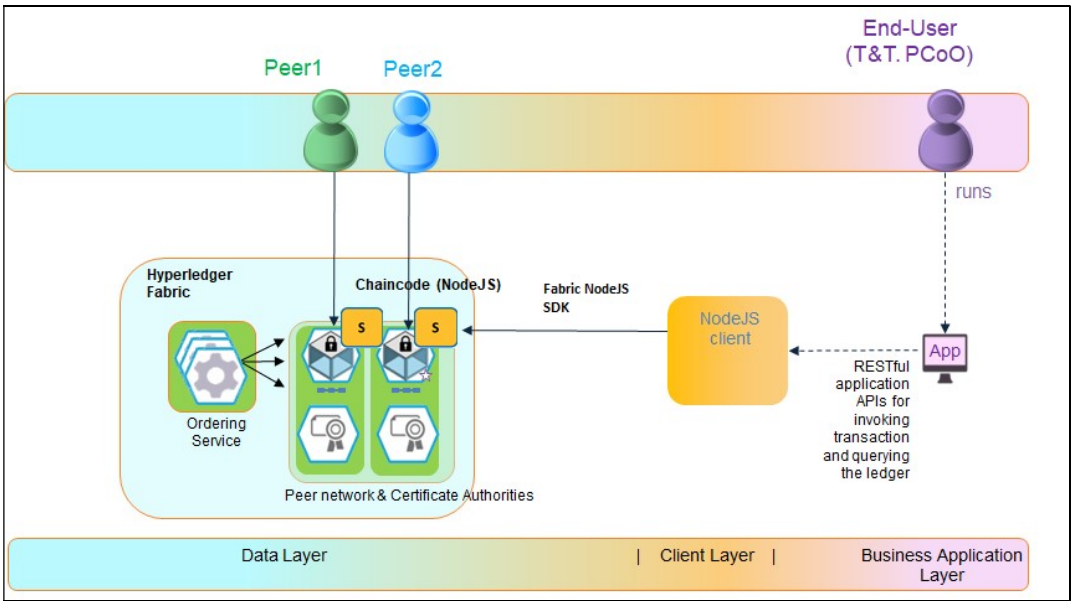

Figure 18: Embodiment within NIMBLE

As can be seen in Figure 18, the blockchain support architecture for the different NIMBLE services consists of several components. First, the blockchain network itself which currently involves a single channel per service supported. That channel hosts the chaincode which drives the interaction with the component itself. The network is further comprised of peers that obtain and validate blocks of transactions and hold a replica of the shared distributed ledger. In addition, there is a chaincode implementing the blockchain smart contract, storing and updating the state of entities and allowing querying the ledger and the associated world state for information on those entities. The chaincodes supporting the NIMBLE components are written in the Golang[10] programming language. In addition, there exists a NodeJS client embedding internally a Hyperledger Fabric NodeJS client to invoke/query the appropriate chaincodes. The NIMBLE components interacting with the blockchain invoke the blockchain client exposed methods via a set of exposed RESTful APIs, passing JSON objects whose structure is agreed upon between the components, to invoke transactions that change internal state or query for stored information. An ordering service is associated with a channel (several channels can share an ordering service) with the mission to create a total order of incoming transactions, cut blocks, and make the blocks available to the designated peers.

Mediating between the blockchain infrastructure, including the smart contracts layer and the rest of the components of NIMBLE is an application listening for incoming requests and communicating on the other side with the blockchain infrastructure. The application exposes a REST interface to the rest of the NIMBLE platform, and acts as gateway between NIMBLE on the one hand and the blockchain infrastructure on the other hand. The application communicates as well with a certificate authority (CA) in order to resolve the cryptographic material and identification of entities. The application embeds an internal blockchain client which in turn can invoke transactions and queries on the blockchain itself.

[10] https://golang.org/

3.3.2 Capabilities supported in the NIMBLE platform

This section is intended to introduce the specific use cases and tools that will make use of the blockchain technology within the NIMBLE project. As aforementioned, the Blockchain integration currently focuses on two representative scenarios, namely track and trace, and preferential certificate of origin. The NIMBLE platform shall take advantage of a blockchain based infrastructure as the provider of a trusted (in a trust-less environment) single source of truth.

The functionalities that a blockchain infrastructure can support in supply chain scenarios will be translated into plans for specific use cases to be deployed in the NIMBLE platform. Hereinafter, a short description of each supported capability is provided.

Tracking and tracing

Product identification codes turn products into unique, identifiable items. This is the basis for tracking and tracing. Tracking means to identify the current state of an item. Tracing provides the user with a history of the events that changed the item's state.

T&T receives and processes tracking events. An event has a place, time and meaning. Event data originate from one or more measurement devices that read product identification codes via Radio Frequency Identification (RFID) technology. NIMBLE applies the Electronic Product Code Information System (EPCIS) standard to systematize and name the tracking events. It supports interoperability with existing tracking and tracing infrastructures of the platform users.

Tracking and tracing is a value-added service that demonstrates the interaction between NIMBLE and a blockchain. In the specific scenario, the progress of a produced item in the shop floor shall be tracked, via RFID gates. In addition, relevant condition parameters during the production shall be tracked and stored in the blockchain using information originating in IoT sensors.

Figure 19 shows the interfaces which are supported by the blockchain application for the tracking and tracing scenario. There are interfaces available for adding information to the blockchain via transactions invocation and interfaces for querying the blockchain for specific information. The corresponding swagger can be accessed in the following link: http://161.156.70.125:5000/api/.

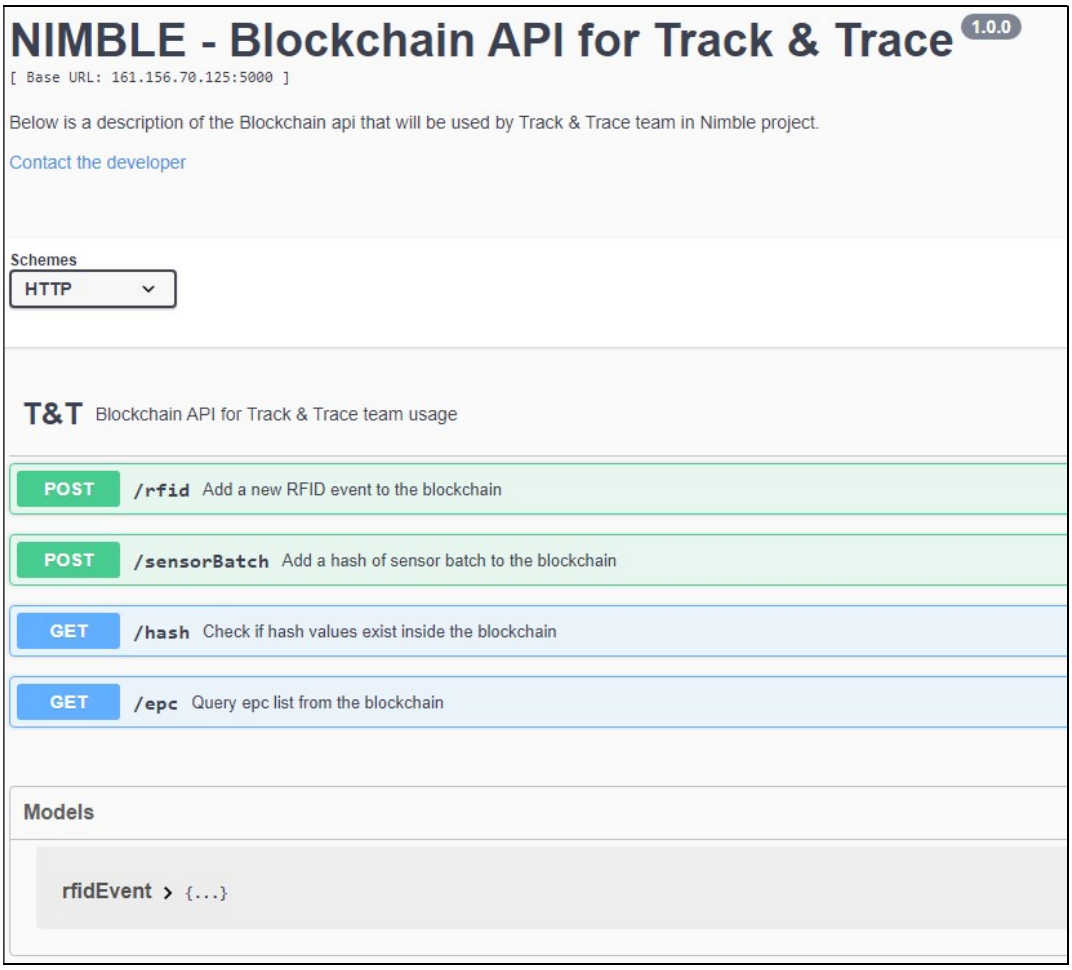

Figure 19: T&T support interfaces and data models

Figure 20 depicts the interface for adding an RFID event. This call is invoked by the T&T component when an item passes through a gate in the production floor of the factory. The information provided may include more than a single entry (see the epcList part in the figure). For each such entry there's an associated hash (shortHashForEPCList), and there's a combined hash covering all the individual entries (completeHash).

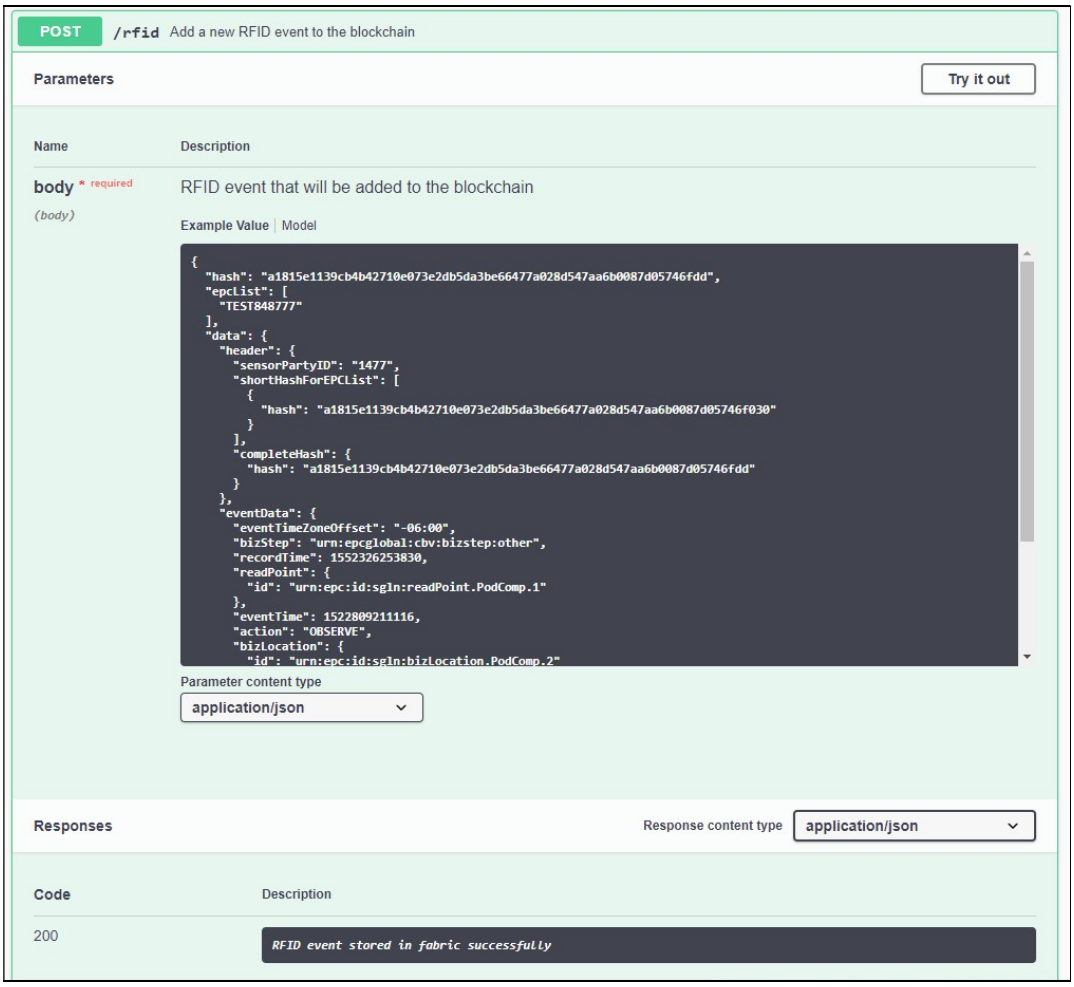

Figure 20: RFID event addition

Figure 21 depicts the addition of a hash corresponding to a batch of sensors reading. The sensors reading represents environmental conditions in the factory while the item was being produced. The actual values are stored elsewhere, in a Track & Trace data store, but upon retrieval of the individual sensors information we can retrieve the corresponding hash to ensure that the data retrieved is genuine and has not been tampered with. The readings are batched into groups and stored as such; the hash value corresponding to the batch is stored on the blockchain to ensure that data has not been tampered with.

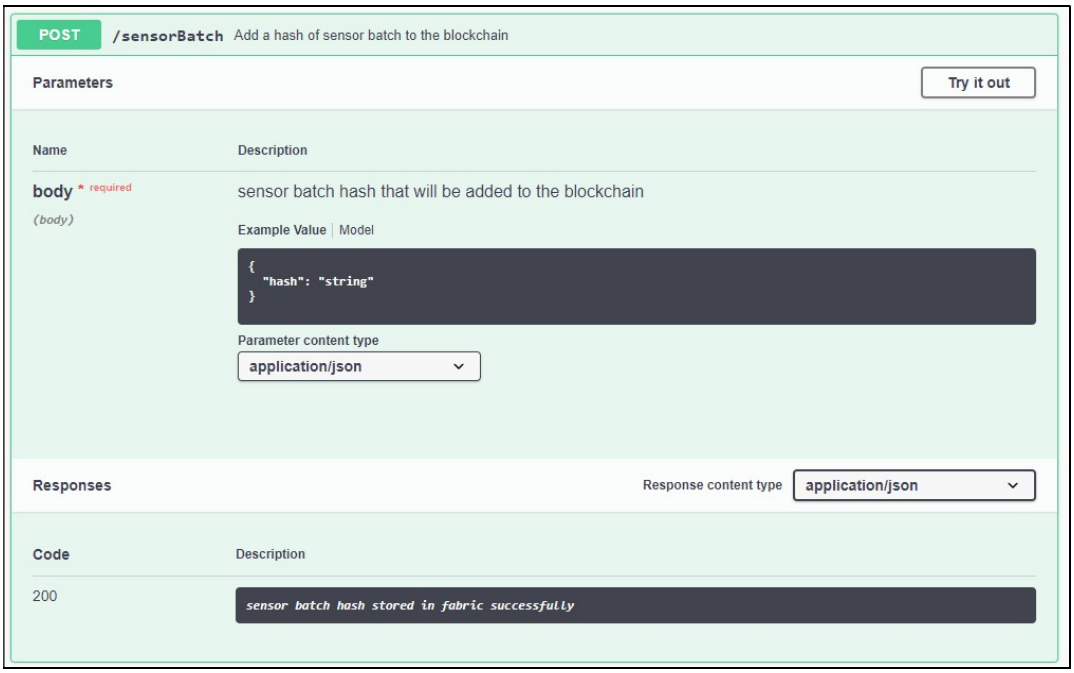

Figure 21: batch of sensors addition

Figure 22 depicts a query to retrieve hash values previously stored on eth blockchain. The query interface expects to receive as a parameter a list of hash values to be verified. Each hash string is comma separated, the first value being the corresponding event type and the second value is the hash code. This call is used at the data validation phase of the track and trace component. This phase takes place before the corresponding data leaves the system, for example to be displayed on a dashboard. Hash vales corresponding to different types of events can eb validated in one such call. The response includes validation status for each entry and as convenience value indicating whether all requested entries were validated.

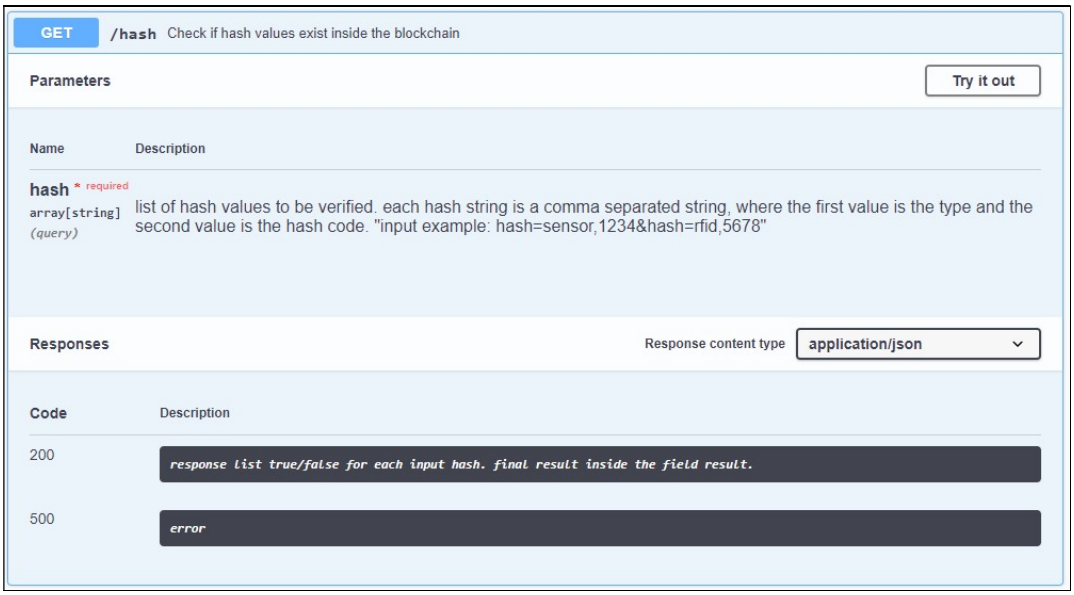

Figure 22: query for hash values

Figure 23 depicts a query to obtain historical information of a specific item that has been tracked. The response includes all the RFID events that were associated with this entry. In essence the full path the item has passed through the factory can be retrieved using this call.

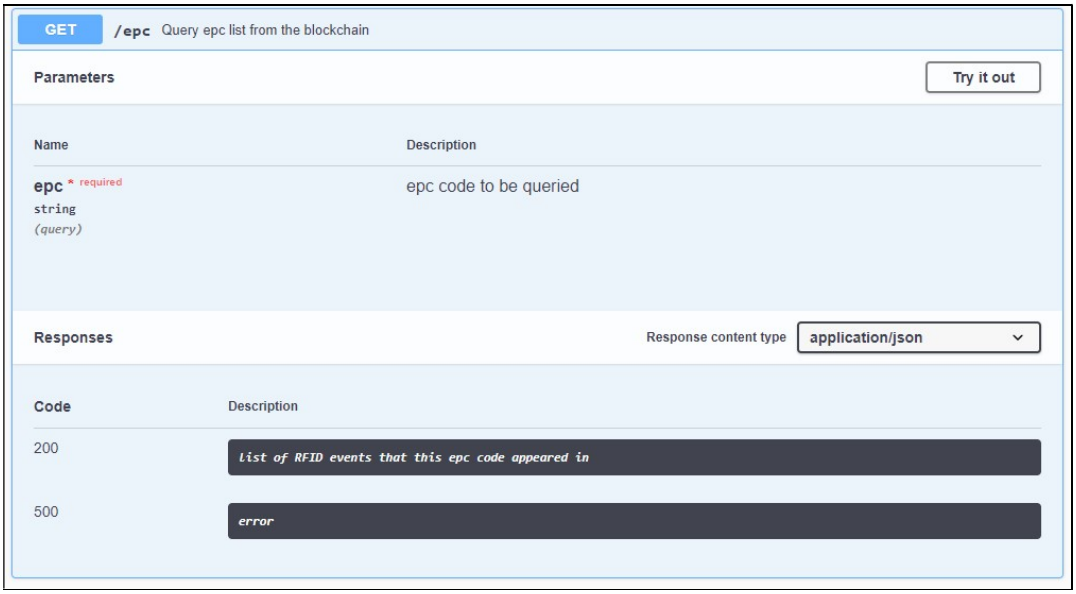

Figure 23: Query EPC history from the blockchain

Preferential Certificate of Origin

The Preferential Certificate of Origin is a statement of a producing company regarding the production of goods. The possibility of emitting a Preferential Certificate of Origin depends on factors stated in the commercial agreement between the European Commission and the country that is importing the goods. This type of certificate allows companies to reduce or avoid paying duties costs. Normally the major request is that one or more substantial transformations to the

produced good are made in a European Country. This type of certificate can be issued directly by the producer but, at least in Italy, a specific authorization by the Custom Authority must exist. The invoice is used by the customs to calculate duties and shall contains information about the Preferential Origin of the goods.

The EU has made different commercial agreements with other countries to facilitate the exchange of goods along the customs in order to improve the exportation of typical European products. These agreements, mainly, specify those products and production constraints that determine a (possible) reduction of importation duties and, thus, the final cost for the local buyer. These agreements state also the obligations of the sellers in terms of documentation required and custody of those documents. Controls are requested from remote customs and performed by local custom office on the company's premises.

The blockchain is being used to keep a record of the invoice which includes the preferential certificate of origin, including hash of the document(s) and a timestamp, such that the customs can verify that relevant documents have not been tampered with.

Figure 24Figure 19 shows the interfaces which are supported by the blockchain application for the preferential certificate of origin scenario. There are interfaces available for adding information to the blockchain via transactions invocation and interfaces for querying the blockchain for specific information. The swagger specification can be accessed via: http://161.156.70.125:7695/api/.

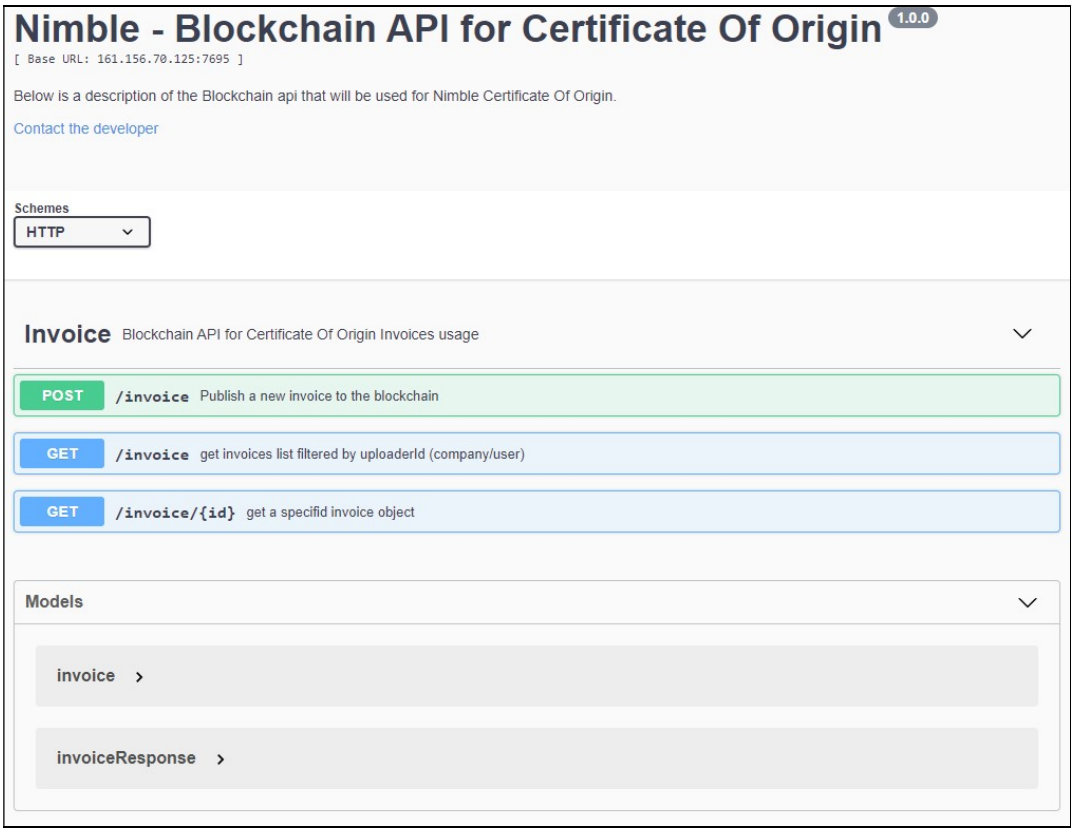

Figure 24: Certificate of origin interfaces

Figure 25 depicts the interface for adding a new invoice information to the blockchain. Information provided is the invoice ID, the corresponding hash code and an optional link to the actual document. The returned information includes the timestamp in which the information was incorporated to the blockchain.

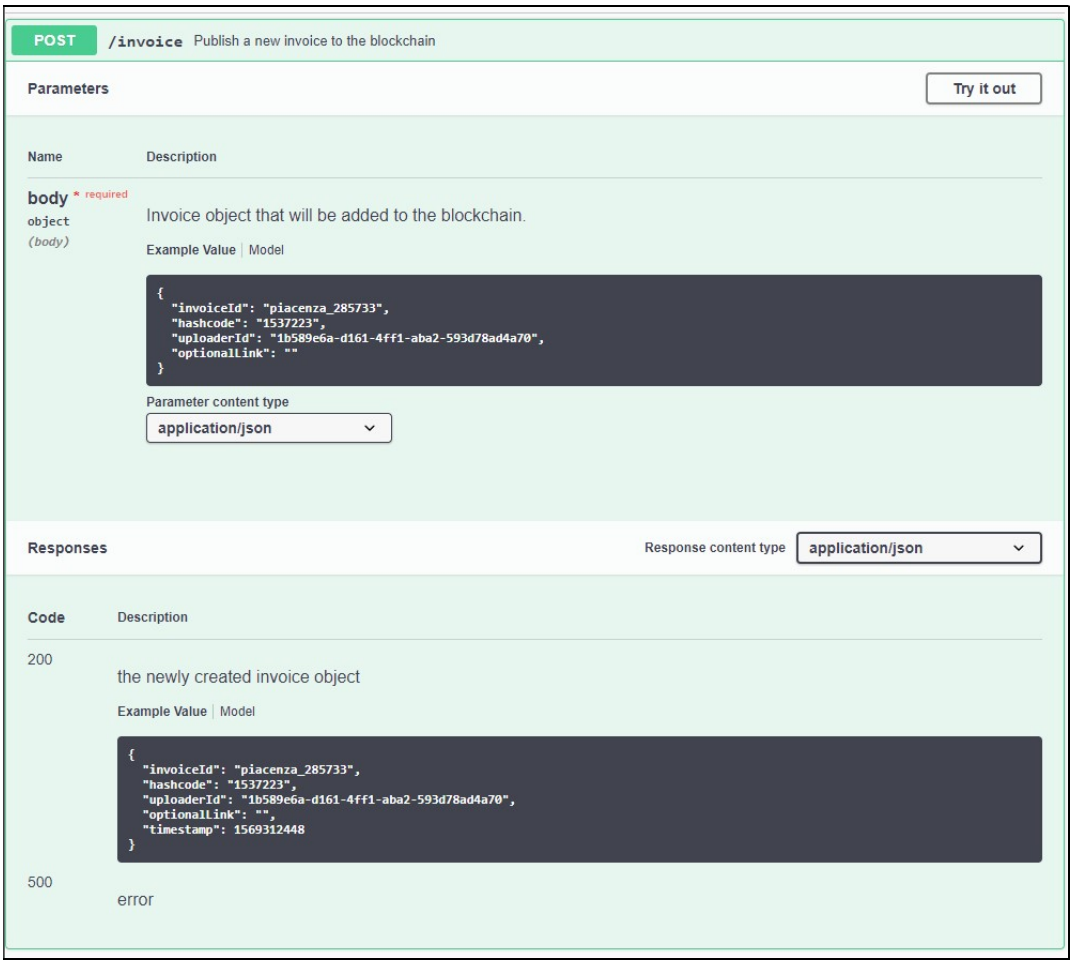

Figure 25: adding a new invoice information

Figure 26 depicts the interface for retrieving invoices entered by a specific ID.

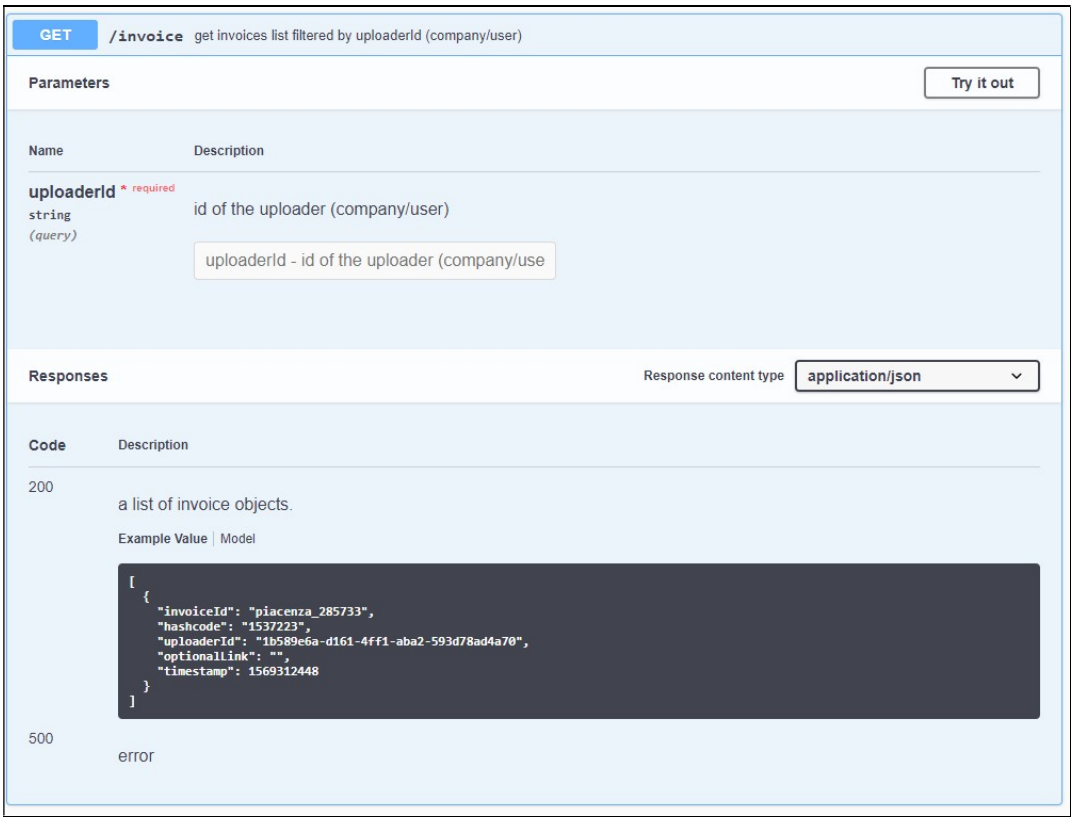

Figure 26: restore invoices added by a specific ID

Figure 27 depicts the interface for retrieving information of a specific invoice.

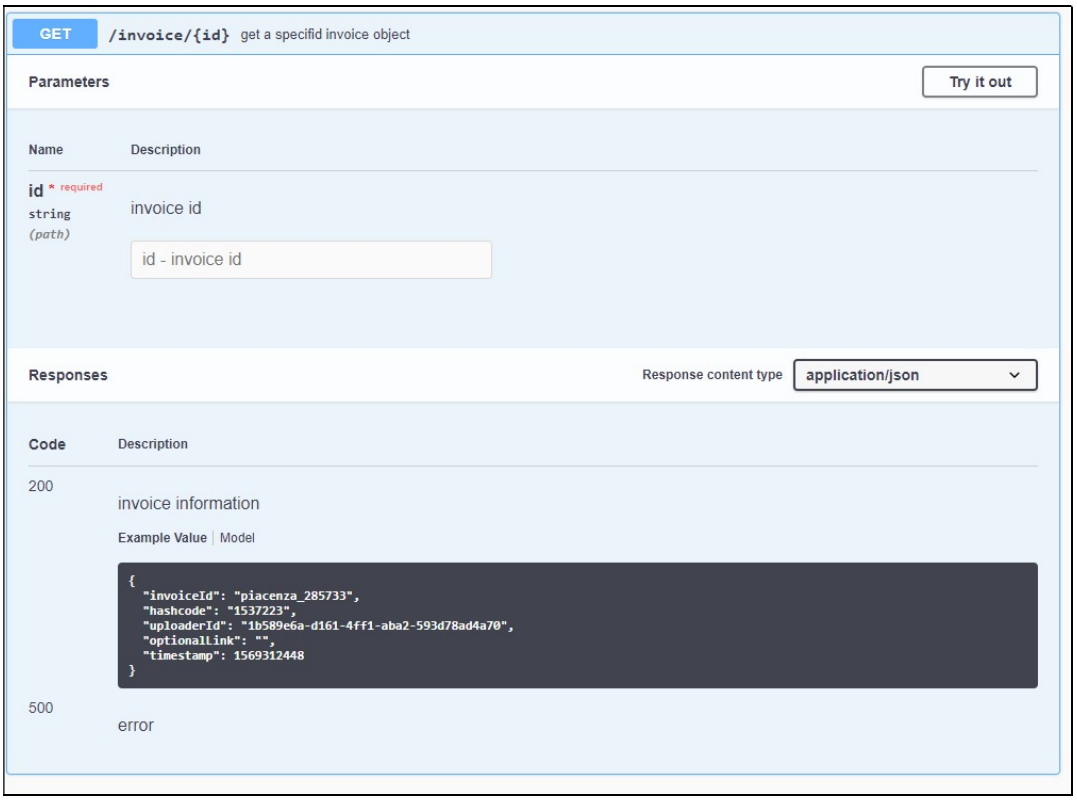

Figure 27: retrieve specific invoice information

3.4 Supporting blockchain platform architecture

Permissioned networks require the registration of entities which can participate only after being accepted. Thus, the membership of a network is known, and each action is taken on behalf of a specific entity using issued certificates. Underlying a Hyperledger Fabric network, as most permissioned networks, is the notion of a consortium, which is a group of organizations that agreed to set up a blockchain network between them, establishing the governance body and rules. As can be seen in Figure 28, most central organizations within a blockchain network will deploy (or use) a Certificate Authority (CA) on their behalf, and will contribute peer(s), which are components that endorse, validate, and hold replicas of the shared ledger. In addition, an ordering service needs to be set up by the organisations, to order the transactions, cut blocks and make them available to the peers.

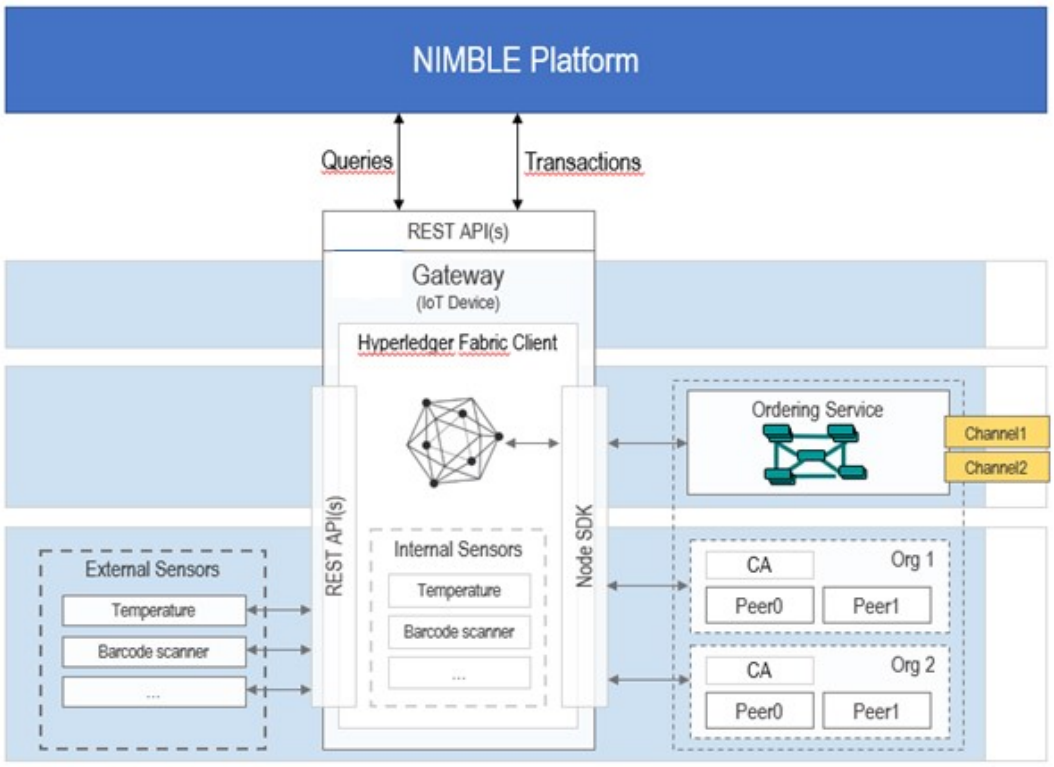

Figure 28: Blockchain network components

Transactions among members is performed in the context of a channel. A channel creates a separate ledger visible only to the organizations included in the channel.

To become a network member first each participant needs to be registered and enrolled in the network via a Certificate Authority. A user with an appropriate role (such as admin) can register additional users from his organization. Using a secret received during the registration process the new user can enroll, thus receiving the required credentials for participating in the blockchain network. Using these credentials, a user may start invoking transaction on the blockchain.

To bootstrap a network, we need to have an ordering service up, create peer processes on behalf of organizations, create channels, and distribute the appropriate cryptographic material, including the certificates required to participate in the network, to each entity. Once the backbone is in place we can install and instantiate chaincodes. We need to install a chaincode on each peer which may endorse transactions for that channel (endorsing peers are the only ones that actually execute the chaincode). The chaincode needs to be instantiated on one of the peers, to create the bond between the chaincode and the channel and run the initialization method specific to that chaincode.

Once the chaincode is in place users can start invoking transactions and queries on the blockchain channel. By using a client, the user assembles a transaction and sends it to the endorsing peers. In the NIMBLE case the application receiving REST requests from NIMBLE components performs this operation. Endorsing peers' policy is determined per chaincode and establishes the identity of potential endorsers and the conditions that have to be satisfied for a transaction to be approved. Once the client has received responses from the endorsing peers, he can evaluate whether the transaction abides by the endorsement policy and can thus go through or needs to be dropped. Endorsing peers are the only ones that actually run the chaincode (in a

simulating mode) and return the corresponding read and write sets of the transaction, namely the keys and versions of variables that were read or written by the simulated execution of the chaincode. Endorsed tractions are then sent, along with the corresponding read and write set to the ordering service. The ordering service in turn orders incoming transactions, cuts blocks, and makes the blocks available to the peers. Peers in turn obtain a new block, validate the transactions in it, and apply the write set for the transactions that have been determined to be valid.

In Figure 28 we can see the components described above in play. On the right-hand side, we can see the main components of the blockchain network itself namely the ordering service, peers, and the certificate authorities. We can further see the channels which are declared in the system associated with an ordering service and a sub-set of the peers. In the middle we can see an application which embeds a Fabric client to communicate with the blockchain while exposing a REST interface for other NIMBLE components.

The blockchain system mostly consists of three layers. First, a physical layer of deployment which includes the establishment of the network consisting of organizations, their participating servers (peers in Fabric), channels, cryptographic material, and more. A second layer includes the establishment and distribution of smart contracts (chaincode), which programmatically determine the rules and actions to be followed. These smart contracts control the state that is saved in the underlying blockchain DB. On top of these lie the business layer which connects between the external world and the underlying blockchain infrastructure. In our deployment, as in most cases, this layer consists of the programmatic core of the interactions to follow, which exposes, on the one hand, to the higher layers of external applications a RESTful interface through which the interactions with the blockchain are mediated. On the other side it includes a Blockchain client (such as the Fabric NodeSDK client), which is in charge of interacting directly with the blockchain by invoking transactions, invoking queries, and establishing call-backs. These call-backs enable an asynchronous mode of operation in which a process is notified by the blockchain network on the occurrence of events which were declared as being of interest to the application or higher layers.

Once a generic blockchain based platform infrastructure (Figure 28) has been put in place, it's possible to develop specific blockchain constructs on top of the infrastructure to provide capabilities which are specific to the NIMBLE platform hosting the blockchain network.

4 Summary

The starting point for this document is the basic NIMBLE platform developed in WP3, including the core services that each such platform instance needs to support. In this document we elaborate on extensions and advanced services which can be run on top of a NIMBLE instance to provide a flexible set of specific capabilities. In particular, we first described the NIMBLE hosting and deployment structure, followed by the manner in which a federation of several NIMBLE instances can operate. Finally, we describe the blockchain as a technology for supply chain scenarios and contextualize it with NIMBLE added value services.

Lessons learned: The early architectural choice for taking a microservices approach has proven to be helpful. Individual component were able to advance faster as they were independent, while maintaining the agreed upon external interfaces. A complementary choice was to use established middleware technologies for collaboration patterns among microservices, such that they don't even have to know about each other, including deployment location and interfaces. For example, a message bus was used to exchange information among microservices without

directly calling each other. In parallel databases were used which enabled accessing to data, for example after having been notified by the message bus of the existence of such data.

The use of Docker, an open source containerization technology, proved useful as well as it brought in flexibility to deploy services on multiple kinds of hosts, from local to cloud, using advances technologies such as docker-compose and Kubernetes.

The use of a complete CI/CD using an open source toolchain involving GitHub, Jenkins and Kubernetes proved effective as well.

A challenge was to maintain a single source code for several different kinds of deployment, attempting to manifest all the differences via run-time configuration parameters. The process works reasonably well but could be improved in the future. Similarly, the deployment of a new instance works reasonably well, but could be made more efficient in the future as well.