UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Disseny d'una solució basada en microserveis aplicada al projecte ecoMobility

Treball Fi de Grau

Grau en Enginyeria Informàtica

Autor: Antonio Tallà i Vivó

Tutor: Vicente Pelechano Ferragud
Joan Fons i Cors

Curs: 2017/2018

Agraïments

Podria escriure'n uns quants fulls agraint a totes les persones que s'han creuat amb mi al llarg d'aquestos anys però intentaré resumir-ho el màxim possible.

Gràcies a la meua família per haver-me donat la possibilitat d'existir, d'educar-me i de fer-me la personeta que soc avui dia. Hem patit moments durs però sempre hem eixit endavant.

Gràcies a la que es convertirà la meua dona aquest mes per donar-me suport al llarg d'aquestos 8 anys. Amb tu he madurat i he crescut com a individu, m'has ajudat a desenvolupar una part de mi que no canviaria per res.

Gràcies a les meues amigues de sempre Sandra, Maite, María, Irene i Belén. Està sent un molt bonic viatge al vostre costat, des del principi vaig saber que seríem amics per sempre.

Gràcies a tota la gent que he conegut a la universitat i amb els que he compartit milers de rialles, històries i aventures. "La vieja guardia" i el "Atlètic de Macrats": Mou, Marc, Alberto, Anne, Jorge i Morella. Als que després he anat fent amistat i que s'han quedat ja per sempre: Jordi, Riera, Carlos, David, Àngel, José Manuel.

D'aquestos m'agradaria agrair especialment a Alberto i a José Manuel, gràcies per creure en mi i fer-me vore lo equivocat que estic moltes vegades.

Gràcies als meus tutors Joan i Vicent per guiar-me durant la realització d'aquest treball.

Gràcies al bàsquet per fer-me aprendre tots els valors que aporta, més enllà de la derrota o la victòria. Em fas evadir-me de tot sempre que tinc un baló a les mans i em fas disfrutar més encara veient com creixen les personetes que any rere any entrene, això no te preu.

Per últim agrair als meus companys de GMV que els últims dos anys han format part de la meua vida laboral. És un gust i un plaer treballar amb gent que t'ajuda a millorar al teu voltant.

Resumen

En este trabajo expondremos una propuesta arquitectónica para convertir la arquitectura monolítica de ecoMobility a una solución orientada hacia los microservicios. Los objetivos de ecoMobility son la eficiencia energética y la reducción de emisiones de gases contaminantes provocados por el tráfico rodado que circula a través de las poblaciones.

Para poder alcanzar estos objetivos ecoMobility tiene desplegada una red de sensores en diferentes zonas de Valencia, los cuales mediante gestores de mensajería envían la información hacia la aplicación monolítica encargada de transformar el mensaje recibido "en sucio" de los sensores y reenviar la información procesada hacia los diferentes servicios que gestionan el tráfico de la zona que engloban.

Además de la reconversión de aplicación monolítica hacia una arquitectura de microservicios, implementaremos una nueva funcionalidad que modificará los límites de velocidad de las zonas controladas por los sensores. Así pues, en función de las emisiones que registren los sensores se aumentará o disminuirá la velocidad de las calles o de los tramos de carretera que existen alrededor de las ciudades.

Para el desarrollo de nuestra propuesta nos apoyaremos en Spring-boot que es una tecnología que nos permite crear microservicios de forma sencilla y que funciona una capa por encima del framework Spring, el descubridor de microservicios desarrollado por Netflix llamado Eureka y el broker RabbitMQ que nos permite implementar el protocolo asíncrono de mensajería MQTT y definir colas de trabajo asíncronas AMQP.

Palabras clave: Internet de las cosas, ciudades inteligentes, microservicios, IoT, arquitectura.

Resum

En aquest treball exposarem una proposta arquitectònica per a convertir l'arquitectura monolítica d'ecoMobility a una solució orientada als microserveis. Els objectius d'ecoMobility són l'eficiència energética i la reducció d'emissions de gasos contaminants provocats per el tràfic rodat que circula a través de les poblacions.

Per a poder abastir aquestos objectius ecoMobility té desplegada una xarxa de sensors a diferents zones de València, aquestos mitjançant gestors de missatgería envíen la informació cap a l'aplicació monolítica encarregada de transformar el missatge "en brut" rebut dels sensors i reenviar la informació processada cap als diferents serveis que gestionen el tràfic de la zona que engloben.

A més de la reconversió d'aplicació monolítica cap a una arquitectura de microserveis, implementarem una nova funcionalitat que modificarà els límits de velocitat de les zones controlades per els sensors. Així doncs, en funció de les emissions que registren els sensors s'augmentarà o es disminuirà la velocitat dels carrers o trams de carretera que envolten les ciutats.

Per al desenvolupament de la nostra proposta ens recolzarem amb Spring-boot una tecnologia que ens permet crear microserveis molt fàcilment i que funciona una capa per damunt del framework Spring, el descobridor de microserveis desenvolupat per Netflix anomenat Eureka i el broker RabbitMQ que ens permet implementar el protocol lleuger de missatgeria asíncrona MQTT i definir cues de treball asíncrones AMQP.

Paraules clau: Internet de les coses, Ciutats intel·ligents, microserveis, IoT, arquitectura.

Abstract

In this project we will present an architectural proposal to convert the monolithic architecture of ecoMobility into a microservices-oriented solution. The main objectives of ecoMobility are energy efficiency and reduction of polluting gases caused by traffic circulating in cities.

To achieve these objectives, ecoMobility have distributed sensor networks in different areas of Valencia, these sensors through asynchronous messaging send messages that contains pollution information to the monolithic application which is on charge to process this information and forward the processed information to the different services that manage the traffic of the area that they cover.

In addition to the conversion of the monolithic application to microservices-oriented solution, we implemented a new functionality that modifies the speed limits of the areas controlled by the sensors. Therefore, depending on the emissions recorded by the sensors, the speed of the streets or road sections that exist around the cities can be increased or decreased.

For the development of our proposal, we have use Spring-boot, which is a technology that allows us to create microservices in a simple way, it works a layer above of the Spring framework, the microservices discover developed by Netflix called Eureka and the broker RabbitMQ which allow us to implement the light weight messaging protocol MQTT and define AMQP asynchronous work queues.

Keywords: Internet of things, smart cities, microservices, IoT, architecture.

Taula de continguts

Contenido

Taula de figures

Taula de scripts

1. Introducció

Segons la OMS [1,2] els principals indicadors de la mala qualitat de l'aire estan directament relacionats amb les emissions produïdes per els motors de combustió. Per tractar de pal·liar el dany que produeix sobre els organismes vius la OMS estableix uns llindars o límits de nivell de contaminació de seguretat. EcoMobility [3] pretén aportar solucions per a controlar les emissions produïdes en temps real i adaptar el tràfic en funció dels nivells de pol·lució.

L'exemple més clar es produeix a Madrid [4, 5 i 6], on els dies en el que el nivells de pol·lució son alts es restringeix el tràfic a diferents zones de la ciutat. Amb mesures com la gestió del tràfic controlant els semàfors, reduint la velocitat dels carrers o trams de carretera, restringint el tràfic a diferents carrers, controlar l'enllumenat, ... es pot conseguir reduir els nivells de pol·lució.

EcoMobility està desenvolupat com a una gran aplicació monolítica, que vol dir açò? Que totes les funcionalitats que implementa estan albergades a la mateixa aplicació. Açò implica una serie de desavantatges que descriurem durant tota la memòria i posarem de manifest les millores que aporta la migració d'aquest monòlit a una solució orientada a microserveis.

Exposarem les bases de l'arquitectura i els seus diferents elements (API Gateways, enregistrament de serveis, descobriment de serveis, balancejadors,...).

1.1. Motivació

El nou patró de disseny arquitectònic orientat a microserveis és el moviment lògic a seguir cap als nous desenvolupaments d'aplicacions. L'èxit de grans empreses com Amazon, Netflix [10], Uber [11] i moltes altres, està ajudant a que cada volta es parle més i s'aposte més per aquest model.

Subdividir una aplicació monolítica o crear-ne una nova de tal forma que els serveis que oferisca es puguen desenvolupar per separat, de forma desacoblada, centrant-se solament cadascun en oferir una unitat de negoci.

Desenvolupar codi i funcionalitats de la forma més senzilla possible, utilitzar llibreries o mòduls compartits entre els microserveis, utilitzar diferents llenguatges de programació i tecnologies. Aquests son alguns del exemples que manté la filosofia orientada als microserveis.

Desengranarem una part de les funcionalitats que ofereix per a separar-ho en ens independents, convertint així aquesta aplicació en una aplicació distribuïda on cada microservei serà mantenible, escalable i fàcil d'implementar noves funcionalitats o futurs nous serveis que s'afegiran a l'arquitectura d'ecoMobility.

1.2. Objectius

Pretenem dissenyar la base d'una arquitectura orientada a microserveis aplicable al projecte de *smart citi*es ecoMobility. Seleccionarem una part de les funcionalitats que ofereix i les subdividirem en microserveis.

Seguint la filosofia dels microserveis, d'entre unes quantes tecnologies elegirem quines són les que més s'adeqüen a la nostra arquitectura e intentarem afegir totes les funcionalitats que es puguen desacoblar en una llibreria compartida encarregada de les comunicacions, definició de constants, models i auditoria.

A més implementarem un cas nou en el que en funció de la pol·lució d'una zona en concret es modifiquen els límits de velocitat de la mateixa.

En resum, els objectius principals per al desenvolupament d'aquest treball són:

- Definir l'arquitectura base del nou ecoMobility amb microserveis desengranant les funcionalitats de l'aplicació monlítica.
- Decidir que tecnologies anem a emprar i perquè.
- Familiaritzar-se, aprendre i desenvolupar l'arquitectura plantejada amb aquestes tecnologies.
- Instal·lar i configurar el broker de missatgeria RabbitMQ afegint, el plugin web per a la visualització de les cues i el plugin MQTT que funcionarà una capa per damunt del mateix broker.
- Definir una llibreria compartida que contindrà funcionalitats per a la comunicació entre microserveis, models comuns i auditoria.
- Implementar una nova funcionalitat que modifique els límits de velocitat de les vies urbanes.
- Desenvolupar 4 microserveis que implementaran les funcionalitats de:
    - Dispatcher: Rebrà els missatges dels dispositius IoT.
    - Gateway: Redirigirà el missatge rebut cap al microservei corresponent.
    - Descobridor de serveis: Amb Eureka coneixerem l'estat del nostre entorn, els microserveis que estiguen disponibles.
    - Processament del missatge: processarà el missatge i afegirà informació.
    - Gestor de velocitat de trams i carrers: En funció del nivell de pol·lució rebut modificarà la velocitat d'un carrer o un tram de carretera.

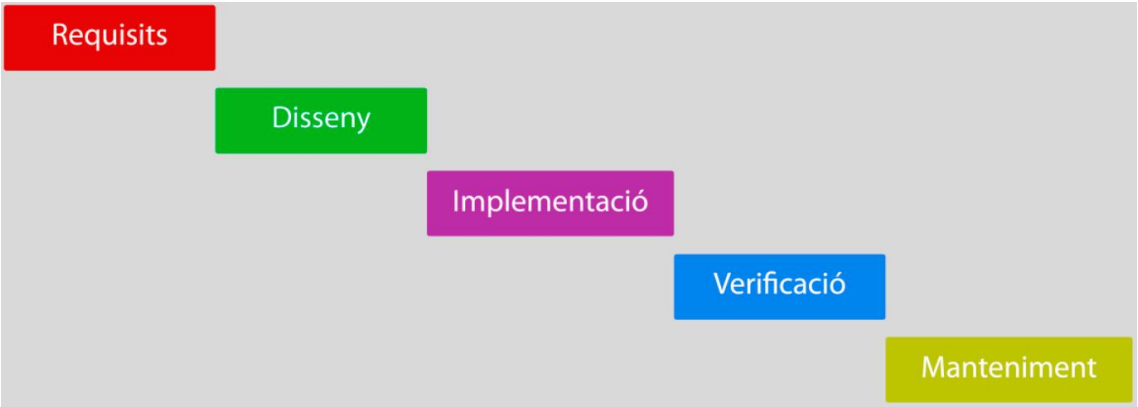
1.3. Impacte esperat

La finalitat del projecte d'smartCity ecoMobility és tindre una xarxa de sensors de diferents tipus desplegats per tota la ciutat. Alguns d'ells aniran focalitzats cap a la detecció de gasos de pol·lució, altres cap a l'eficiència energètica mitjançant sensors de llum que modificaran l'enllumenament públic, altres detectaran els nivells de contaminació acústica, etc... Quin impacte tindrà aquesta nova arquitectura?

EcoMobility és una aplicació monolítica que rep missatges de tots aquestos sensors, els processa i pren decisions per a millorar les condicions de l'aire i l'eficiència energètica de la ciutat. Migrant aquest monòlit dividint els serveis que té en petites unitats de negoci (microservei) on cadascun s'encarregue d'executar petites i senzilles tasques, abastirem els objectius que té aquest nou model d'arquitectura. Amb aquest treball es busca definir les bases de l'arquitectura de microserveis a partir de la que ecoMobility puga anar extraient poc a poc els seus serveis, així en el futur el desenvolupament de nous serveis siga senzill, acoblable, escalable i robust.

1.4. Planificació

Per a realitzar aquest projecte es va a utilitzar la metodologia en cascada [7], es tracta d'un model lineal per al disseny software i que es basa en un procés seqüencial.



Il·lustració 1: Metodologia en cascada

El començament no serà fàcil ja que una arquitectura de microserveis canvia totalment el plantejament del disseny d'aplicacions i les primeres setmanes van a ser dedicades directament a l'estudi i aprenentatge. Així doncs caldrà assabentar-se bé sobre les tecnologies que poden utilitzar-se i com es poden encaixar dintre de la nova arquitectura.

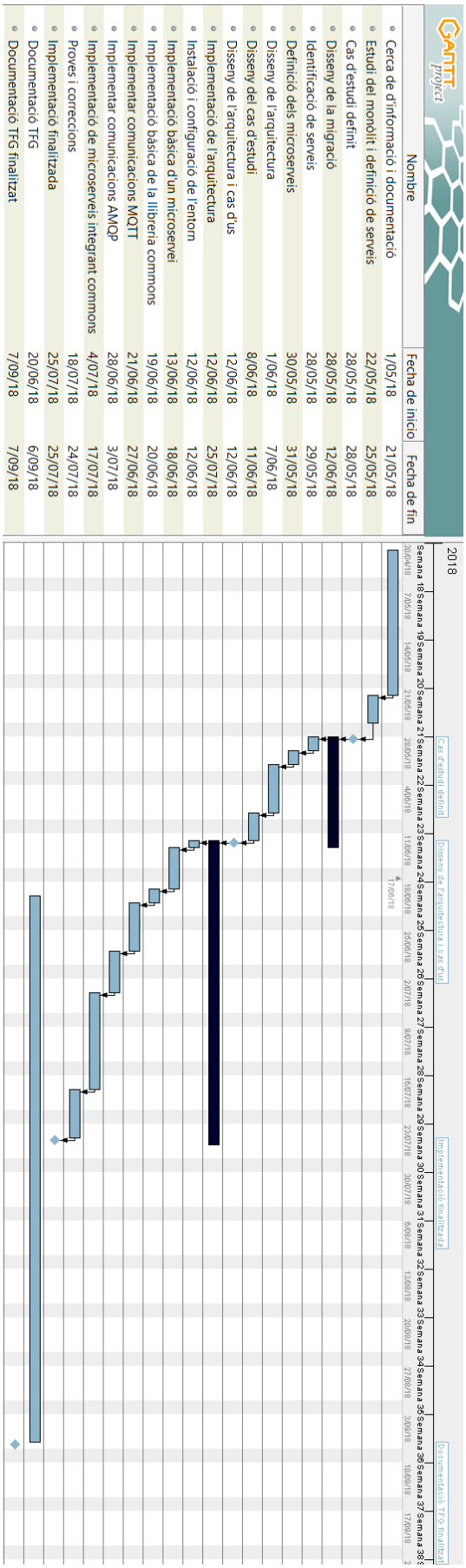
Continuant amb la metodologia en cascada, es va a dedicar un gran esforç a la identificació, extracció, definició de requisits i disseny de funcionalitats ja implementades dintre del monòlit.

Posteriorment amb les funcionalitats extretes i entenent com funciona internament el monòlit es plantejarà i definirà el cas d'us implementat, aquest també tindrà una important part de recerca ja que no estic gens familiaritzat amb les *Smart Cities* i menys amb els protocols de contaminació de la OMS.

Seguint el patró d'aquesta metodologia s'arribarà a la implementació, verificació i manteniment. Segurament durant les fases de disseny e implementació el disseny vaja

evolucionant ja que a mesura que investigue i que avance me n'adornaré que pot ser no tot siga com primerament ho haja pensat. Quelcom que es farà de segur serà implementar una llibreria compartida amb funcionalitats el més genèriques possible per a facilitar la lectura i mantenibilitat del codi.

La planificació de l'escriptura i documentació del projecte es durà a terme durant tota la durada del mateix però es començarà a escriure pràcticament a l'inici de la fase d'implementació.



Il·lustració 2: Diagrama de Gantt[8]

Cal dir que en el cas que aquest projecte s'haguera desenvolupat per una empresa amb recursos l'ideal hauria sigut utilitzar la metodologia *DevOps* [9] (Development and Operations) que es basa en un conjunt de pràctiques que s'utilitzen per a automatitzar processos entre els equips de desenvolupament i sistemes. La finalitat d'aquestes pràctiques és establir una cultura de col·laboració entre equips per a poder compilar, provar i publicar el software amb major rapidesa i eficàcia.

1.5. Estructura del treball

A les següents fulles d'aquest treball us trobareu diferents punts on en cadascú es definiran les parts més importants de la implementació d'aquesta nova arquitectura de microserveis.

Comencem doncs per el capítol dos on definirem el context tecnològic dels microserveis, exposant les bases, definicions de conceptes i casos d'èxit de diferents empreses que han optat per aquesta migració de les seues plataformes. S'exemplificarà una migració d'una arquitectura monolítica a una arquitectura de microserveis. Per últim, es mostraran les tecnologies que poden ajudar el desenvolupament del projecte i d'entre totes aquestes s'aniran definint punt per punt quina serà l'elecció final per a utilitzar-la en la implementació de l'arquitectura.

A continuació s'explicarà àmpliament que és l'arquitectura de microserveis confrontant-la amb l'arquitectura monolítica que és una arquitectura totalment oposada. Aquest capítol es conclourà exposant els avantatges i desavantatges de totes dos arquitectures.

El quart capítol descriurà el context d'aquest treball, es definirà el projecte ecoMobility per tal d'entendre el repte que tenim que afrontar. S'exposaran les parts de l'aplicació monolítica que anem a migrar i les ubicarem cadascuna a un microservei, descrivint breument quina part del monòlit faran. Es definiran els requisits per al desenvolupament de la nova arquitectura i es conclourà amb la definició de la solució proposada.

Al capítol següent es descriuen el disseny de l'arquitectura proposada i també es definirà més detalladament les unitats de negoci que implementen cada microservei dins de l'arquitectura, el descobridor de serveis i el broker de missatgeria.

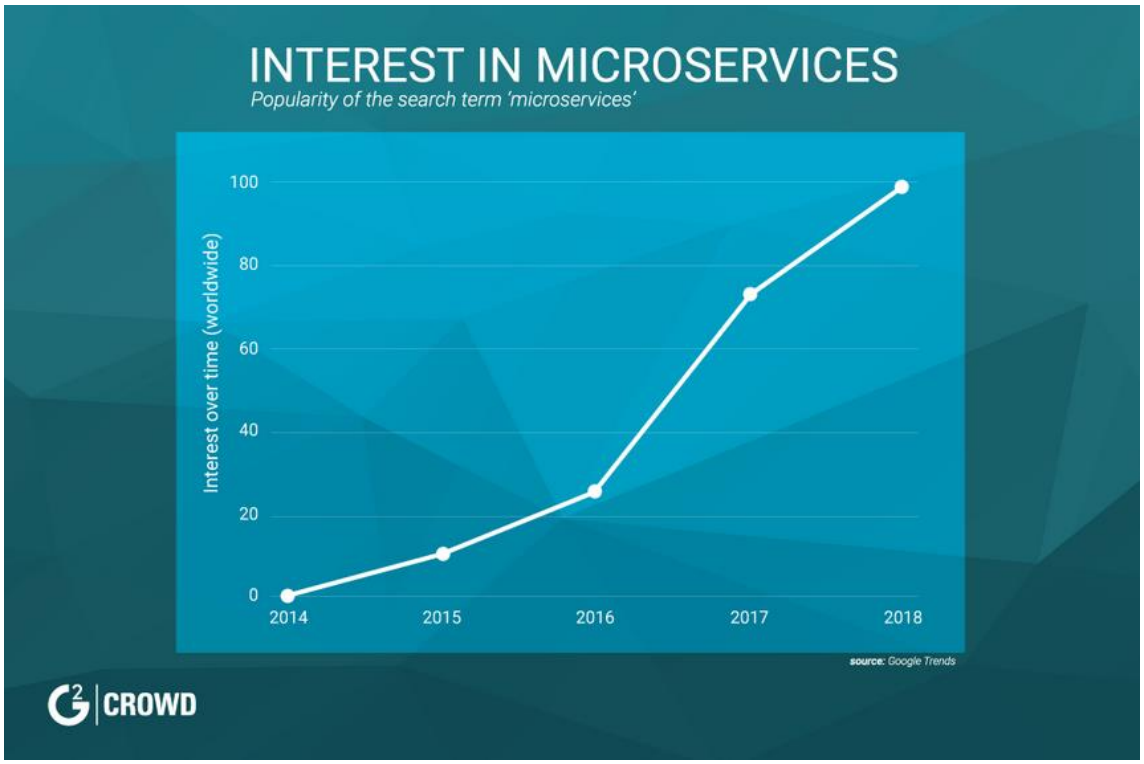
La implementació d'aquesta arquitectura la tindrem al capítol 6, on es defineix com s'implementa microservei a microservei, com es configura la missatgeria i com s'adherix el descobridor de serveis Eureka.

Per últim s'exposaran les conclusions i pensaments que s'han tret fruit del desenvolupament d'aquest projecte i possibles treballs futurs que poden desenvolupar-se.

2. Context tecnològic

2.1. L'arquitectura de microserveis

L'arquitectura de microserveis es un tipus d'arquitectura que gràcies a l'èxit d'empreses com Amazon, Netflix, Apple [10] i Uber [11] entre altres ha fet que l'interés per aquest nou patró de disseny estiga al alça els darrers anys.



Il·lustració 3: Interés a través del temps sobre els microserveis

L'arquitectura de microserveis ofereix la possibilitat a les empreses de aïllar aplicacions per a que siguen desenvolupades per equips petits multidisciplinaris, és a dir, equips conformats per tècnics que dominen les diferents capes que conformen les aplicacions. Aquest patró augmenta la velocitat de desenvolupament, la confiabilitat del serveis i la escalabilitat d'aplicacions a través de la implementació i entrega continua [12].

Segons la companyia *open source* NGNIX [13] prop del 70% de les empreses estan utilitzant o investigant sobre els microserveis i 1 de cada 3 té ja desplegada una arquitectura de microserveis.

Una aplicació monolítica es compon d'una multitud de serveis tremendament acoblats executant-se com a un únic servei [12]. A continuació es va a definir un exemple del que podria ser una aplicació monolítica i dels serveis que pot contindre.

El subjecte de l'exemple serà Netflix [14] que ha sigut tota una pionera i ha migrat tots els seus serveis continguts de una aplicació monolítica cap als microserveis. Amb el

temps ha creat una *suite* de productes *open source* [15] que faciliten el desenvolupament d'aquestos nous patrons d'arquitectura.

Comencem. Imaginem que tota l'aplicació web de vídeo per demanda Netflix estiguera continguda al mateix servidor. Gran part dels serveis que ofereix estan acoblats i són fortament dependents entre ells.

Es pot donar la situació en que momentàniament un gran nombre d'usuaris vulguen connectar-se per a vore l'última serie de moda que Netflix acaba de publicar. Aquest fet pot fer que es sobrecarregue el servidor de peticions d'accés i per tant perdre la disponibilitat de l'aplicació sencera durant un temps.

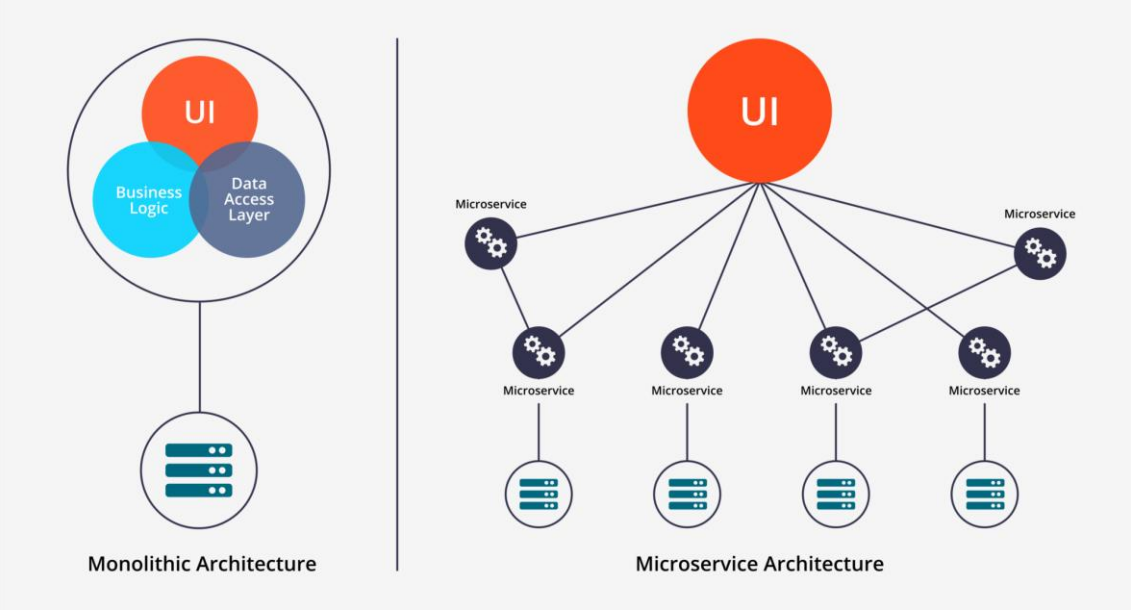
El monòlit pot ser estiga en un entorn on hi hagen moltes instancies, és a dir, que l'aplicació escale en funció de la demanda. Açò podria solucionar els pics de peticions temporals però cada volta que es vulga desplegar una nova versió haurem de replicar aquesta a tots els servidors, el qual deriva en una tasca pesada i complexa si cada nova versió té un gran impacte respecte l'anterior.

A mesura que el monòlit creix i es desenvolupen nous serveis aquestos acaben sent més difícils d'implementar i acoblar als serveis ja existents. Suposen un risc per a la disponibilitat de l'aplicació ja que si no es controlen les errades provocades per el fort acoblament que té pot fer que l'aplicació sencera falle originant una total pèrdua dels serveis. Per exemple, vull accedir als episodis d'una serie, si aquesta petició no obté resposta d'altre servei dependent o es perd la connexió a la base de dades que ens respon a eixa petició i no es controla l'errada l'aplicació pot deixar de funcionar.

Abans d'identificar alguns exemples de serveis que Netflix pot contindre, vaig a definir que son els microserveis.

Un microrservei es una aplicació que ofereix únicament un servei, és a dir, es una unitat petita que està enfocada en fer allò que serveix de la millor forma possible. Així doncs, una arquitectura de microserveis es una serie de petits i autònoms serveis desacoblats fortament cohesionats que es comuniquen i treballen entre sí [12].

El vicepresident d'investigació de la consultoria internacional *Gartner* Gary Olliffe [16], defineix els microserveis com un component de l'aplicació estrictament delimitat, fortament encapsulat, dèbilment acoblat, d'implementació independent e independentment escalable.



Il·lustració 4: Arquitectura monolítica vs arquitectura de microserveis

Tornant al nostre fictici monòlit, podem identificar alguns exemples de serveis: l'autenticació a la plataforma, fer una cerca de series o pel·lícules per temàtica, afegir una serie a la nostra llista de series preferides... Tots continguts dintre de la mateixa aplicació...

Una forma de migrar els monòlits es anar extraient serveis. Per exemple, la nostra llista de preferits. Cada volta que des de el monòlit faja una petició per a vore la meua llista, aquesta sol·licitarà al microservei que li oferisca el servei que estic demanant.

Abans s'han definit un parell de situacions que poden originar problemes greus als monòlits, bé, ara vaig a resoldre aquestes dos situacions però pensant en que la nostra aplicació ja ha sigut migrada.

Pensem que s'ha modificat el codi el servei del que parlàvem abans, la sol·licitud de mostrar la nostra llista de series preferides, i li havem millorat el temps de resposta del microservei i a més s'han afegit que es puga vore una serie d'imatges prèvies de la serie. El canvi suposarà menys temps ja que s'haurà localitzat abans on es podia modificar el codi i a més ha sigut fàcil d'implementar donat la senzillesa del canvi.

Desplegar una nova versió d'aquest servei no implica parar l'aplicació sencera ni deurem replicar el mateix a tots els servidor, a més si el microservei desplegat conté alguna errada en el codi que fa que no estiga disponible, podem continuar utilitzant altres serveis de la plataforma com per exemple vore la primera serie que ens aparega a les novetats.

Aquest ha sigut un exemple per a exposar de forma senzilla la diferència entre els dos conceptes arquitectònics, més endavant entrarem en profunditat definint les característiques bàsiques de totes dos i també exposarem els inconvenients de cadascuna.

2.2. Tecnologies

Després de definir per damunt què és una arquitectura de microserveis, és el moment de descriure quines tecnologies estan al nostre abast i decidir les que es van a utilitzar per a desenvolupar aquesta arquitectura.

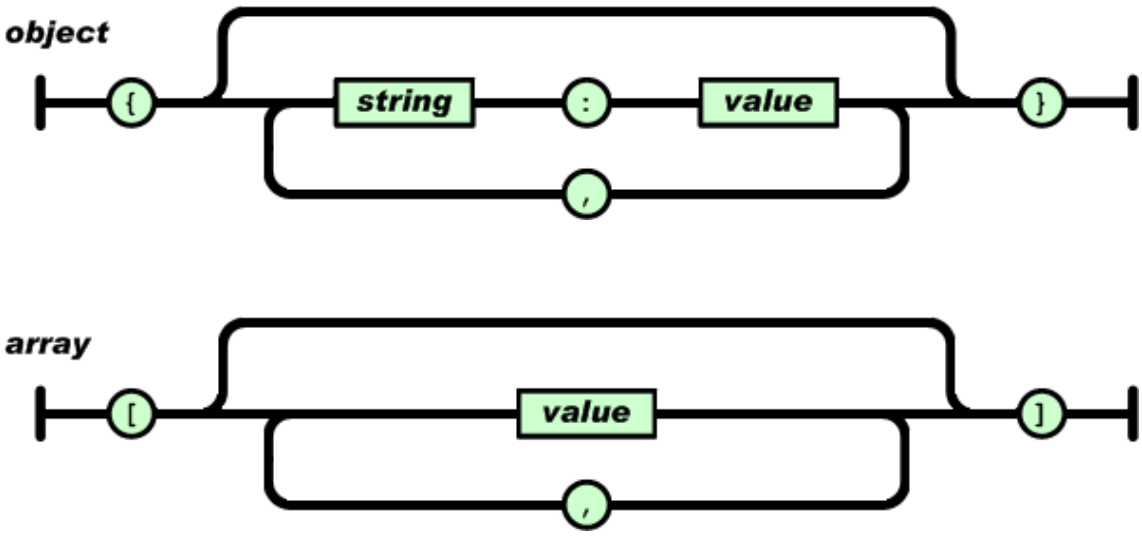
Abans de començar cal dir que com a base aquest projecte s'ha desenvolupat utilitzant el sistema operatiu Linux Mint 18.3 amb nom en codi Sylvia i que està basat en Ubuntu Xenial. Com a IDE (Integrated Development Environment) s'ha emprat Eclipse que és una plataforma software composta per un conjunt de ferramentes de codi lliure i per últim, el llenguatge de programació i un subconjunt de tecnologies utilitzades per a construir els microserveis s'ha fet amb Java 1.8.

2.2.1. Tipus de dades dels missatges entre microserveis.

JSON [17] (JavaScript Object Notation)

És un format de text lleuger que permet l'intercanvi i emmagatzematge d'informació. És un llenguatge totalment independent de la resta però utilitza convencions que són similars a altres programes, com per exemple, C, C++, Java, Perl, Python, JavaScript... Per a les màquines és un llenguatge fàcil d'analitzar gramaticalment i fàcil de generar. Com a avantatge principal, el format JSON és fàcilment llegible i comprensible per els humans.

JSON està construït a partir de dos tipus d'estructures. Per una banda està format per una col·lecció d'elements clau-valor i per altra banda per una llista de valors que en altres llenguatges serien un vector, una llista, un array o una seqüència.



Il·lustració 5: Objecte clau valor i llista de valors en format JSON

XML [18] (eXtensible Markup Language)

El llenguatge de marcat extensible (XML) és un format de text simple i molt flexible derivat de SGML (ISO 8879). Algunes característiques d'aquest llenguatge són: fàcil de llegir i escriure, compatible amb SGML, altres llenguatges de programació poden processar-lo, disseny formal i concís, ... però la més important es la possibilitat de crear documents XML [19], aquestos segueixen una definició de tipus de document (DTD) amb la qual cosa es pot validar la construcció del missatge abans de ser enviat o manipulat.

```
<?xml version="1.0" standalone="yes"?>
<BankAccount>
      <Number>1234</Number>
      <Type>Checking</Type>
      <OpenDate>11/04/1974</OpenDate>
      <Balance>25382.20</Balance>
      <AccountHolder>
            <LastName>Singh</LastName>
            <FirstName>Darshan</FirstName>
      </AccountHolder>
</BankAccount>
```

Il·lustració 6: Exemple d'estructura de document en format xml

Elecció de tipus de dades

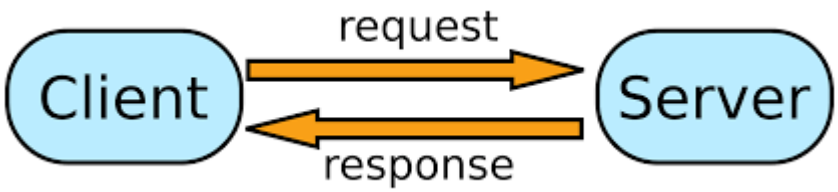
D'entre els dos tipus de dades es va a utilitzar el proporcionat per JSON. Aquesta decisió ve donada per la lleugeresa del missatge i el context en que l'anem a utilitzar ja que xml és més utilitzat per a volums de dades majors que un missatge amb pocs camps, i per últim el seu format també és molt més comprensible per a l'usuari.

2.2.2. Protocol de missatges

HTTP [20] (Hypertext Transfer Protocol)

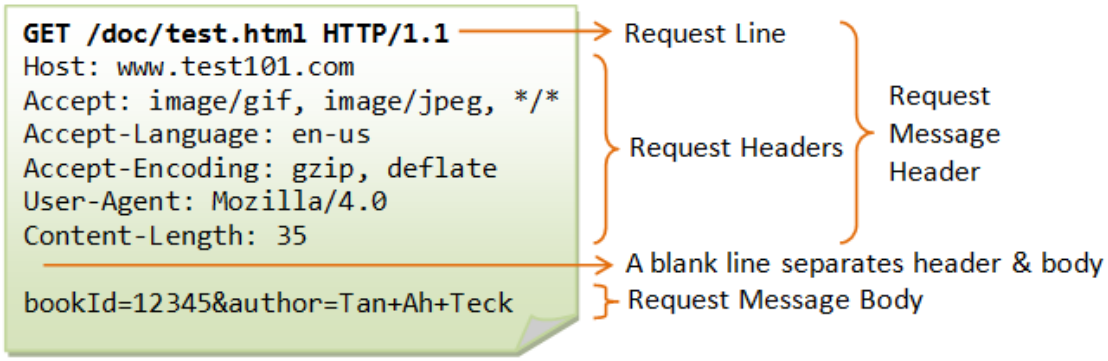
El protocol de transferència d'hipertext és un protocol a nivell d'aplicació que s'utilitza en sistemes d'informació distribuïts. Es defineix com a un protocol genèric, sense estat i que pot ser utilitzat per a moltes tasques més enllà del seu ús per a hipertext, com per exemple: servidors de noms, sistemes d'administració i d'objectes distribuïts, codis d'error i encapçalaments, etc... Una característica d'HTTP és el tipat i la negociació de la representació de dades, permetent que els sistemes es construesquen independentment de les dades que es transfereixen. Aquest protocol s'està utilitzant des de el 1990.

És un protocol orientat a transaccions i segueix l'esquema petició-resposta entre un client i un servidor.



Il·lustració 7: Definició de comportament del protocol HTTP

Alguns dels mètodes que implementa aquest protocol són: GET, HEAD, POST, PUT, DELETE, TRACE i CONNECT.



Il·lustració 8: Exemple de petició GET HTTP

STOMP [21]

Stomp és un protocol interoperable simple dissenyat per a passar missatges asíncrons entre clients a través de servidors de mediació. Defineix un format cablejat de text per a missatges transmesos entre clients i servidors. Aquest és un protocol basat en marcs HTTP. Un marc consisteix en un comandament, un conjunt de encapçalaments opcionals i un cos opcional.

Un servidor STOMP es modela com a un set de destinacions als quals els missatges es poden enviar, aquest protocol tracta les destinacions com a una cadena opaca i la seua sintaxis és específica del a implementació del servidor. A més STOMP no defineix la forma semàntica d'entrega de les destinacions o intercanvi de missatge i poden variar d'un servidor a altre e inclòs d'una destinació a altra.

Un client STOMP pot actuar de dos modes diferents. Per una banda pot actuar com a productor, enviant missatges a un destí al servidor amb un marc SEND i per altra banda també pot actuar com a consumidor enviant un marc SUBSCRIBE per a un destí donat i rebre el missatge del servidor amb marcs MESSAGE [22].

```
SEND
destination:/queue/a
content-type:text/plain

hello queue a
^@
```

```
SUBSCRIBE
id:0
destination:/queue/foo
ack:client

^@
```

```
MESSAGE
subscription:0
message-id:007
destination:/queue/a
content-type:text/plain

hello queue a^@
```

Il·lustració 9: Exemple de protocol STOMP

AMQP [23] (Advanced Message Queuing Protocol)

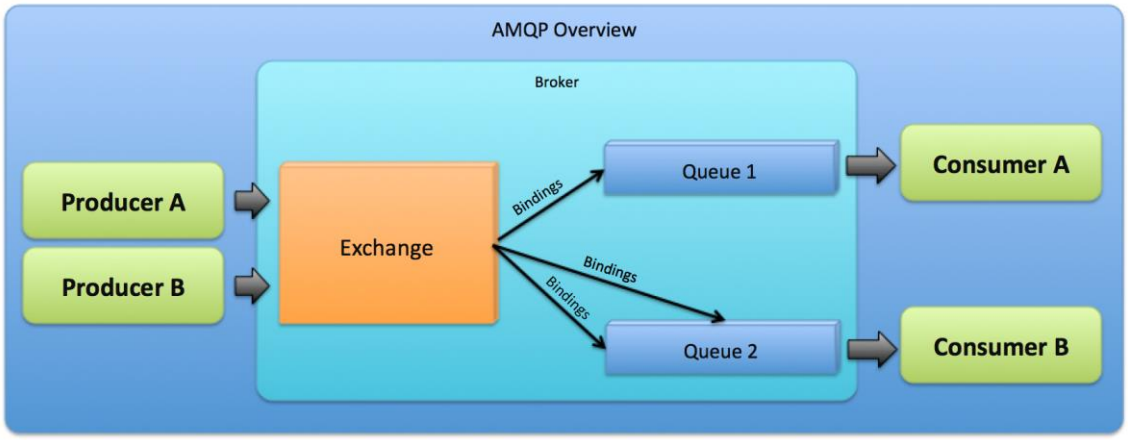
AMQP és un protocol binari de codi obert que permet el pas de missatges asíncrons entre clients i servidors de missatgeria middleware (servidors). Aquest protocol està dividit en dues capes:

Per una banda tenim la capa funcional que defineix un conjunt de comandaments agrupats en classes lògiques de funcionalitats i que treballen a favor de l'aplicació.

Per altra banda tenim a la capa de transport i que porta aquestos mètodes des de l'aplicació fins al servidor i cap enrere. Aquesta capa gestiona la multiplexació de canals, la codificació del context, la representació de les dades i el control d'errors.

Aquest protocol es forma amb els següents components bàsics [24]:

- Client: És qui inicia la connexió o el canal. Els clients poden tant produir com consumir missatges.
- Servidor: És el procés que accepta connexions dels clients e implementa les funcions d'encolat i enrutament de missatges. També anomenat "broker".
- Exchange: L'entitat dins del servidor que rep els missatges del productor i els enruta a les cues corresponents.
- Bindings: És una entitat que crea una relació entre una exchange i una cua.
- Message queue: És una entitat que conté els missatges i els envia cap als consumidors.



Il·lustració 10: Publicació/Extracció de missatges amb el protocol AMQP

MQTT [25, 26] (Message Queuing Telemetry Transport)

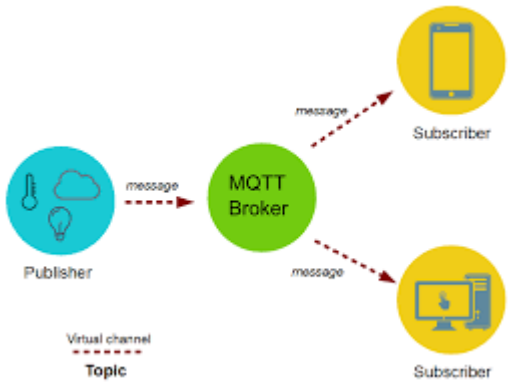
MQTT és un protocol de missatgeria "machine-2-machine" (M2M) lleuger, obert, simple i dissenyat per a ser fàcil d'implementar. És un protocol molt adient als entorns de comunicació d'*Internet of Things* (IoT) ja que es requereixen pocs recursos i ample de banda per al seu funcionament.

Utilitza el patró de missatgeria publicació/subscripció que proporciona distribució de missatges i desacoblament d'aplicacions. L'element que contendrà els missatges s'anomena tòpic i s'utilitza per a filtrar els missatges de cada client. Una característica curiosa és que dintre d'aquest tòpic les comunicacions poden ser bi-direccionals

La forma comuna de comunicar-se és la següent:

- Un client subscriptor es connecta al servidor per a subscriure's a un "tòpic".
- Un client productor es connecta al servidor per a publicar un missatge a un tòpic.

En aquest cas si el tòpic al que tots dos clients s'han connectat és el mateix, el missatge viatjarà d'un a l'altre.



Il·lustració 11: Publicació/Subscripció del protocol de missatges MQTT

Elecció del protocol de missatges

Per a l'elecció del protocol de missatges s'ha decidit emprar 3 dels 4 per a les següents situacions.

HTTP:

- Comunicació dels sensors amb el *Dispatcher*: Els sensors ja venien implementats amb aquest protocol, per tant, l'hem adaptat per a que puga rebre els missatges que envien.
- Comunicació del *Gateway* amb el *Service Discover*: Aquesta es tracta d'una petició molt lleugera que simplement retorna els serveis enregistrats al descobridor de serveis.
- Senzillesa davant STOMP on la configuració d'un servidor pot ser molt enrevessat.

MQTT:

- És un protocol molt lleuger i senzill i que amb molt pocs recursos fa molt bon paper, per tant permet que puguem tindre moltes connexions utilitzant una petita part de la màquina. Ens servirà si fem aquesta arquitectura escalable.

AMQP:

- Utilitzarem aquest protocol per a la gestió del missatges provinents dels sensors i també dels missatges processats, encolant-los en cues de treball. És un protocol que si es declara la *Exchange* com a "directa" permet el balanceig automàtic entre els microserveis subscrits a les cues.

2.2.3. Servidor de missatgeria (Broker)

Eclipse MOSQUITTO [27]

Eclipse Mosquitto es un broker de missatgeria de codi obert molt lleuger que implementa el protocol MQTT. Està dissenyat per a poder utilitzar-se inclús en dispositius que tinguen poca potencia fins a servidors d'altes prestacions. El projecte Mosquitto també proporciona una biblioteca C per a la implementació de clients.

El fitxer executable de Mosquitto sol tindre un tamany aproximat de 120 Kilobytes i que consumeix al voltant de 3 Megabytes per cada 1.000 clients connectats [28, 29]. És un servidor fàcil de configurar e d'instal·lar amb bons tems de resposta.

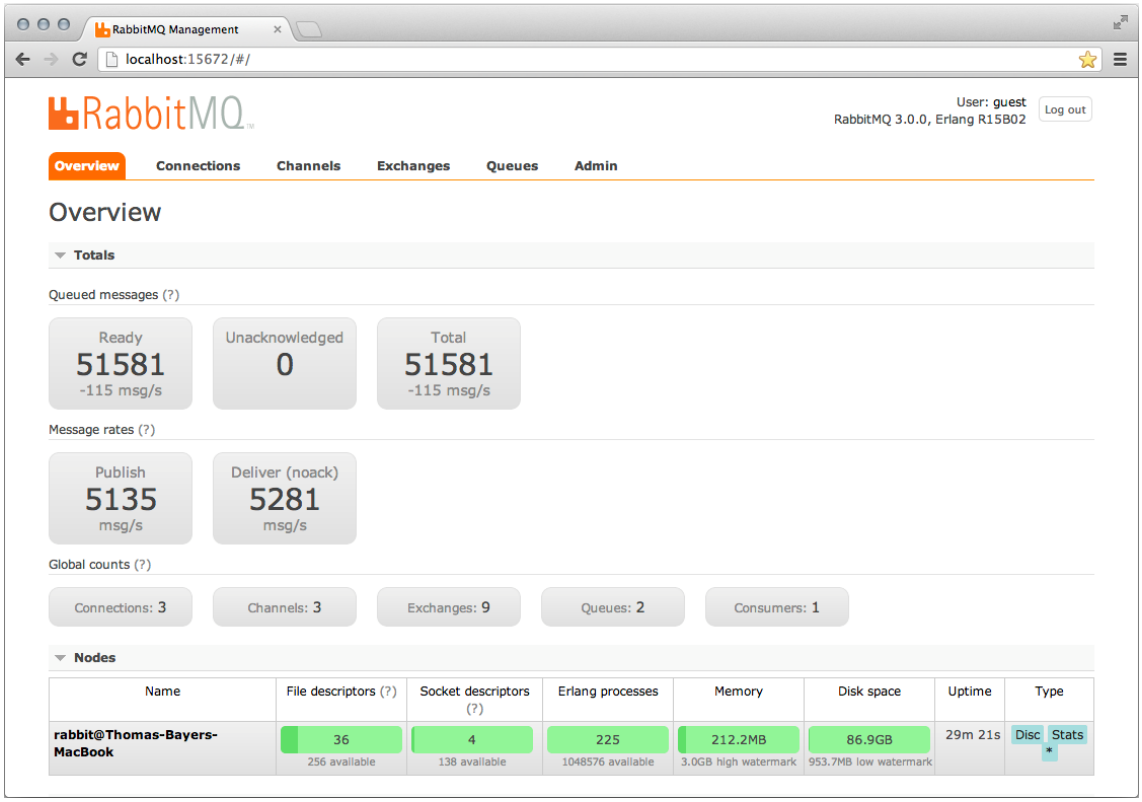
RabbitMQ [30]

RabbitMQ és un broker de missatgeria lleuger i configurable. Soporta multitud de protocols de missatgeria com per exemple, AMQP, MQTT, STOMP i HTTP. Està escrit amb el llenguatge de programació Erlang que és un llenguatge dissenyat per a implementar sistemes en temps real escalables i amb requisits d'alta disponibilitat.

Disposa d'una interfície gràfica amigable per a la monitorització i manipulació del elements de les diferents tecnologies que suporta. A més ofereix flexibilitat d'enrutament de missatges i traçabilitat, és a dir, pots dirigir els teus missatges a l'*exchange* que vulgues o crear-ne una nova i si qualsevol quelcom li succeeix pots traçar el missatge per a vore que ha li succeït.

Quelcom també molt interessant d'aquest broker es que té una serie de *plugins* que faciliten la interoperabilitat dels protocols de missatgeria i el broker [31], i també entre els diferents protocols com poden ser mecanismes d'autenticació (SSL, HTTP externs o servidors LDAP) o implementacions STOMP i MQTT a través de *WebSockets*. Per últim comentar que inclús pots escriure el teu propi *plugin*.

Per últim cal dir que aquest broker és suportat per diferents sistemes operatius com Windows, CentOS, Fedora, Ubuntu... i també per multitud de llenguatges de programació com Python, Java, Ruby, PHP, C#, JavaScript, Go, Haskell Unity 3D, Erlang, Perl, C i C++, Scala, ... [32]



Il·lustració 12: Interfície web de RabbitMQ

Elecció del broker de missatgeria

RabbitMQ és el broker d'enrutament de missatges elegit. La decisió ve presa per la interoperabilitat amb els diferents protocols que anem a utilitzar, la senzillesa de configuració amb el fitxer *rabbitmq.conf* i la àmplia documentació que disposa per a diferents llenguatges de programació. Amb RabbitMQ configurarem les cues de treball i li afegirem el *plugin* MQTT que implementa els patrons de disseny d'AMQP però adaptats a la sintaxis MQTT.

Aquest *plugin* ens ofereix inclús interoperabilitat amb altres implementacions de servidors MQTT com Mosquitto, Paho i WebSphereMQ.

2.2.4. Backends

El disseny de la nostra arquitectura base de microserveis no disposa de *frontends* per tant solament exposarem *frameworks* per a implementar funcionalitats als *backends* de cada microservei. El llenguatge de programació que utilitzarem serà Java ja que és el que llenguatge que he aprés durant la titulació i el que utilitze diàriament al meu treball.

STRUTS2 [33]

Struts és un *framework* web de codi obert per a la creació de aplicacions web amb *Java Enterprise Edition (JavaEE)*. Basat en el model vista controlador (MVC). Aquest *framework* és l'evolució del ja existent Struts1 i la unió d'aquest amb el *framework WebWork*. Les seues tres característiques clau són un controlador de peticions proporcionat per el desenvolupador de l'aplicació i que està assignat a una *URI* (*uniform resource identifier),* un controlador de respostes que transfereix control a altre recurs que completen la resposta i per últim una llibreria d'etiquetes que ajuda als desenvolupadors a escriure menys codi. Algunes característiques d'aquest *framework* són:

- Accions basades en POJOS.
- Fàcil integració amb altres *frameworks* com *Spring* o *Tiles*.
- Arquitectura modular basada en *plugins*, afegeixen funcionalitat al *framework*.
- Suport per a generar vistes utilitzant plantilles.
- Suport per a *frameworks* de generació de vistes com JSP, JSF, Velocity...

JAX-WS [34, 35]

JAX-WS (Java API for XML-Based Web Services) és un framework que forma part de JavaEE i que simplifica el desenvolupament d'aplicacions web i clients mitjançant la utilització d'anotacions Java (especificació JAX-WS 2.2) i proxies dinàmics. Es basa en un model de missatgeria de documents (XML) i està molt orientat al desenvolupament de serveis web, com per exemple, *Simple Object Acces Protocol* (SOAP) que es basa en l'intercanvi de dades amb xml.

Altre exemple on també utilitza documents xml és per a descriure els serveis que oferix utilitzant el format *Web Services Description Language* (WSDL).

Spring [36, 37]

Spring és un *framework* molt potent i madur que complementa a la perfecció la plataforma JavaEE. És un *framework* que té una arquitectura modular, i per tant disposa d'una àmplia oferta que s'ajusta a les necessitats de les aplicacions. Alguns d'aquestos mòduls s'utilitzen per al desplegament d'aplicacions web al núvol, integració amb diferents tecnologies de bases de dades, desenvolupament àgil de microserveis, integració amb servidors de missatgeria, seguretat, ...

Suporta la injecció de dependències i l'ús d'anotacions, és a dir, suporta altres tecnologies que implementen aquestes dos característiques i que es poden utilitzar en volta de les natives d'Spring.

Elecció del framework per a implementar el backend

S'ha elegit *Spring* com a *framework* per a implementar el *backend* dels microserveis per que la seua modularitat e integritat amb moltes tecnologies facilita el desenvolupament d'aplicacions. Dintre d'aquest *framework* utilitzarem tot el que inclou el projecte *Spring-Boot*, pensat pe al desenvolupament ràpid i senzill d'aplicacions i molt enfocat en concret al desenvolupament de microserveis.

2.2.5. Peristència

Definirem la capa de persistència dins de cada microservei per a implementar l'auditoria de les comunicacions, enregistrant tant els missatges que s'utilitzen per a la comunicació entre microserveis com els missatges que arriben dels sensors.

Hibernate [38]

Hibernate és un *framework* desenvolupat per RedHat de codi obert i de tipus *Object Relational Mapping* (ORM). Aquest framework s'utilitza per a connectar les aplicacions amb les dades que existeixen fora d'aquesta, facilitant el mapeig de la representació del model d'objectes cap a la representació del model relacional de dades. Permet desenvolupar classes persistents seguint expressions idiomàtiques orientades als objectes que inclouen herència, polimorfisme, associació, composició i les col·leccions Java. Hibernate no requereix interfícies o classes base per a persistir qualsevol classe o estructura de dades persistent.

Spring Data JDBC [39]

Ja que hem elegit utilitzar el framework Spring per al desenvolupament dels backends dels nostres microserveis, no es pot deixar de banda el mòdul Spring Data JDB que és defineix com a un kit de construcció del teu propi ORM i que pots definir de la forma que vulgues. Proporciona accés i gestió de les dades de la base de dades oferint diferents llocs dintre de l'arquitectura on implementar aquesta lògica, integrar-la amb altres tecnologies o elegint com crear les teues pròpies sentències SQL. Al no ser un ORM en sí no és tan orientat a objectes.

ActiveJDBC [40]

ActiveJDBC és un framework ORM lleuger que es centra en simplificar les interaccions amb les bases de dades llevant la capa extra típica dels managers de persistència i es focalitza amb la usabilitat de sentencies SQL. Infereix directament sobre els paràmetres de l'esquema de base de dades, llevant la necessitat de mapeig d'entitats o taules subjacents. No existeixen sessions, ni managers, no hi ha necessitat d'aprendre

altre llenguatge de consulta de dades, tot ve aportat des de la pròpia llibreria. Per a utilitzar ActiveJDBC primer s'haurà de crear la base de dades i després amb l'ajuda de la llibreria podrem fer tot tipus d'operacions sobre ella.

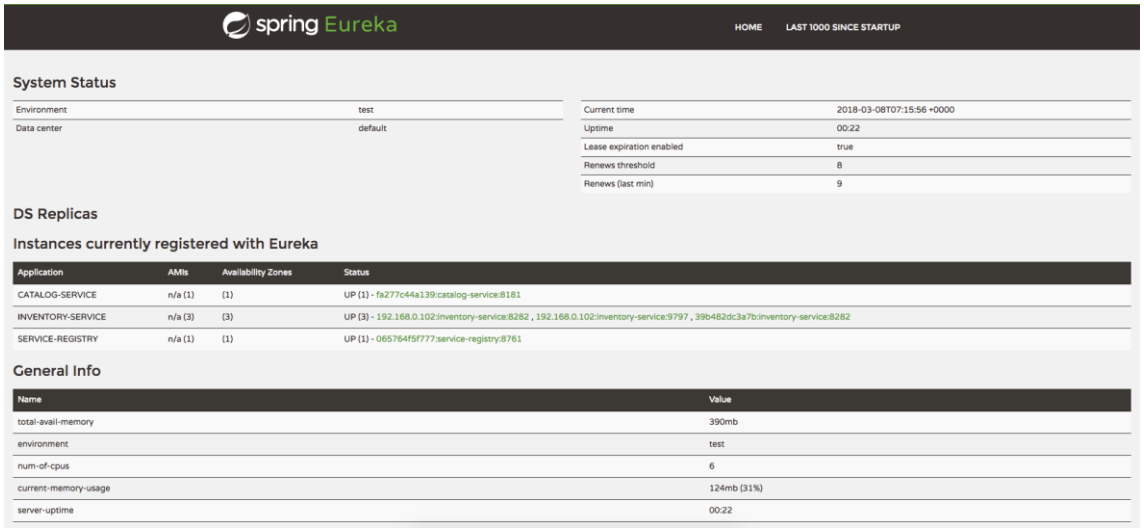
Elecció de framework per a la capa de persistència.

Per a la capa de persistència s'ha decidit utiltizar Hibernate ja que és totalment integrable en Spring gràcies al mòdul *spring-boot-starter-data-jpa* i ens servirà per a crear dinàmicament les taules amb facilitat utilitzant les anotacions natives de JPA.

2.2.6. Descobridor de serveis

Eureka [41, 42]

Eureka és un servei REST que s'utilitza principalment en el núvol de AWS per a ubicar serveis amb l'objectiu d'equilibrar la càrrega i gestionar errors però Eureka també és un component client basat en Java, Eureka Client. Per al desenvolupament d'aplicacions Java existeix un client embegut que permet l'auto registre i la gestió dels batecs. A més té una interfície web molt senzilla que serveix per a vore l'estat actual del servidor Eureka, del host on s'hosteja i de les aplicacions registrades.

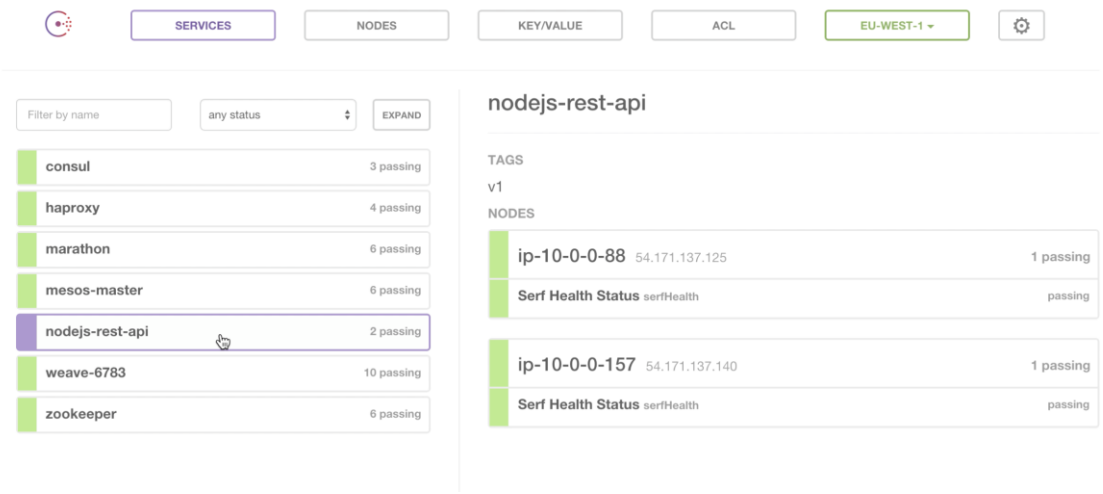


Il·lustració 13: Interfície web del descobridor de serveis Eureka

Consul [43]

Consul és una solució per al registre, localització i salut dels microserveis. Ofereix una API HTTP per a consultar els microserveis registrats saber-ne les seues característiques i estat. A més, ofereix un servici implementat com un servidor DNS que

permet a les aplicacions integrar-se amb els microserveis via nom en volta de per IP. Està escrit amb GO que és un llenguatge desenvolupat per Google. Al igual que Eureka, Consul també té una interfície web amigable per a vore l'estat dels serveis, dels nodes i altres paràmetres.



Il·lustració 14: Interfície web del descobridor de serveis Consul

Elecció del Service Discover

Per a la implementació del Service Discover s'ha elegit Eureka ja que el seu client és molt senzill i molt configurable via propietats d'aplicació d'Spring en volta de Consul que només ens oferiria una API REST.

3. Anàlisi del problema

3.1. L'Arquitectura monolítica

Com ja s'ha definit anteriorment, una aplicació monolítica es compon d'una multitud de serveis tremendament acoblats executant-se com a un únic servei [44], en altres paraules, està construïda com a una sola unitat. Normalment les aplicacions estan definides en tres capes:

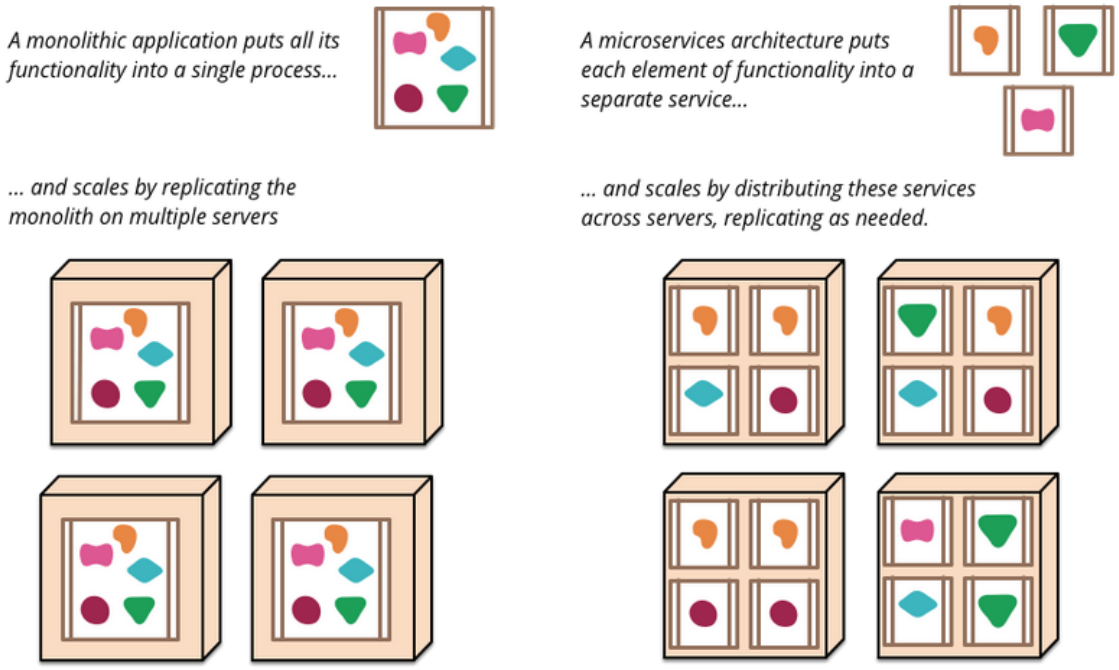
- La capa de presentació es la capa que normalment està executant-se sobre algun explorador web i que s'encarrega comunicar i capturar informació del client amb la mínima càrrega de processament. Aquesta capa està directament connectada amb la capa de negoci.

- La capa de negoci és on s'executen les peticions rebudes per l'usuari i s'envien les respostes després del processament. Conté tota la llògica de l'aplicació. És la capa que es connecta directament amb la capa de persistència per als casos en que es necessite informació de la base de dades.

- La capa de persistència és on resideixen les dades i és l'encarregada d'accedir a eixa informació. Està formada per un o més gestors de bases de dades que reben sol·licituds d'emmagatzemament o peticions d'informació de la capa de negoci.

Totes aquestes tres capes estan en execució davall el mateix procés desplegat a un servidor d'aplicacions. Les aplicacions monolítiques es poden escalar executant moltes instancies de la mateixa aplicació amb un balancejador al capdavant de totes que repartisca la càrrega entre les diferents instàncies. És una pràctica comú tindre les aplicacions desplegades d'aquesta manera, així si algun servidor es cau degut a alguna fallada de l'aplicació, totes les peticions es derivarien cap als altres servidors.

Així doncs diguem que qualsevol cas d'ús en que s'empren les tres capes podria ser una petició d'usuari que necessita accedir a una informació però aquesta s'ha de sol·licitar a la base de dades i transformar-la per representar-la de la manera que espera l'usuari.



Il·lustració 15: Funcionalitats a les arquitectures monolítiques i microserveis

3.2. L'arquitectura de microserveis.

Martin Fowler i James Lewis descriuen l'arquitectura de microserveis com la manera d'abordar una aplicació com una *suite* de serveis, cadascun executant-se al seu propi procés i comunicant-se per protocols de missatgeria o mitjançant cridades HTTP. Aquestos microserveis estan desenvolupats al voltant de les unitats de negoci i són

independentment desplegables per mecanismes automatitzats de desplegament continu. Cada microservei pot estar implementat amb llenguatges de programació (Java, PHP, C,...) i diferents tecnologies (Spring, Hibernate, JPA,...), tanmateix cadascun pot utilitzar diferents bases de dades (Oracle, MySQL, PostgreSQL,...) [45].

Quines son les bases d'un microservei? Un microservei ha de ser una petita unitat de negoci que desenvolupe una tasca de la millor forma possible. Les limitacions del seu desenvolupament es deuen d'establir per les limitacions del context, és a dir, que no ens posem a dividir en moltíssimes parts que facen que l'arquitectura de l'aplicació es torne inmantenible. La finalitat sempre es simplificar, si una funcionalitat no es pot dividir més no cal enrevessar-ho més [12].

Abans de continuar amb les característiques dels microserveis, m'agradaría llançar la pregunta que li pot vindre a qualsevol al cap.

Com de gran ha de ser un microservei?

El tamany d'un microservei no es pot interpretar per el nombre de línies perquè depenent de la tecnologia i del que vulgues implementar pot variar. A sovint està fortament relacionat amb el nombre de persones i les capacitats de cadascuna que conformen el equip de desenvolupament, un microservei molt gran per a un petit equip pot aconseguir que en alguns aspectes tinguem un "mini monòlit" [12].

Aleshores el tamany del microservei el definirà les limitacions del context a implementar, la complexitat de l'arquitectura i les habilitats de l'equip de desenvolupament. En quant al microservei en sí, quan més menut siga un servei més es maximitzen els beneficis i els desavantatges de l'arquitectura de microserveis. Per exemple, quan més menut siga un microserve la interdependència i la complexitat de les comunicacions es vorà incrementada quasi per parts iguals ja que, tindrem moltes parts mòbils dintre de l'arquitectura i el puzle que vulguem definir pot convertir-se en un de ben gran i difícil de resoldre. Es a dir, aquestes 3 característiques han d'existir als nostres microserveis: simples, desacoblats i fortament cohesionats [12, 45].

Els microserveis han de ser autònoms, són entitats separades, serveis aïllats que han de ser desplegats com a tals en servidors independents o en sistemes operatius independents, tractant de no empaquetar múltiples serveis en la mateixa màquina [12]. Les comunicacions entre ells es deuen fer mitjançant cridades a través de la xarxa com poden ser peticions HTTP o protocols de missatgeria asíncrona com pot ser MQTT.

No va a ser el nostre cas d'estudi però una de les característiques que fa que es diferencie molt d'una aplicació monolítica és la multitud de llenguatges i tecnologies que es poden utilitzar a cada microservei. Per exemple, podem implementar un microservei amb llenguatge PHP i una base de dades PostgreSQL e implementar altre amb el que es comunicarà amb Java i una base de dades Oracle [45].

L'elasticitat als microserveis és un altre món, amb els microserveis podem fer que cadascun s'escale per separat en funció de les necessitats puntuals que tinga l'aplicació. Podem implementar arquitectures on les aplicacions puguen controlar la total fallada del servei i degradar la funcionalitat acorde amb la limitació que suposa aquest error. Tot açò fa que els microserveis aprofiten millor els recursos hardware ja que podrem (en

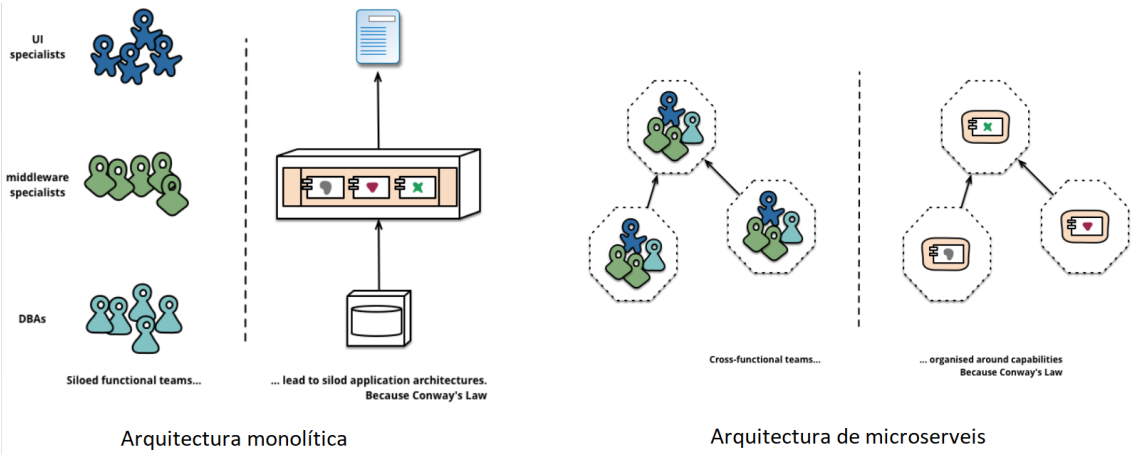
funció de la ferramenta que s'utilitze) inclús assignar la quantitat de CPU o RAM que utilitze [12].

Altre benefici clau que aporten els microserveis és la facilitat en el desplegament de l'aplicació. Canviar una sola línia de l'aplicació monolítica requereix re-desplegar l'aplicació sencera per a alliberar el canvi. Per a evitar estar fent continus desplegaments de canvis menuts, normalment abans d'alliberar una nova versió aquesta sol vindre carregada d'un nombre important de canvis. Si les diferencies entre les dos versions són grans pot augmentar el risc de que quelcom vaja mal. Amb els microserveis podem fer un canvi senzill i re-desplegar-lo amb independència de la resta del sistema, així podem tindre re-desplegat el codi més ràpidament. [12]

Il·lustració 16: Exemple de desenvolupament e integració continua

Per tant si ocorre qualsevol problema durant el desplegament o posteriorment, aquest problema es pot aïllar en un servei individual i fer una tornada enrere cap a l'anterior versió mentre aquesta es corregeix. Per contra, si tot funciona correctament haurem aconseguit desplegar la nova funcionalitat ràpidament. Aquesta es una de les principals raons per les que Amazon o Netflix utilitzen aquest tipus d'arquitectura, així s'asseguren de llevar d'enmig el màxim nombre d'impediments per a tindre el seu software alliberat [12].

Altra característica interessant es la idea de formar petits grups de tècnics multidisciplinaris fent que els equips estiguen més cohesionats i per tant siguen més productius. Aquesta afirmació es regeix per la llei de Conway [45] que enuncia que les organitzacions dedicades al disseny de sistemes estan abocades a produir dissenys que son copies de les estructures de comunicació de dites organitzacions. Resumint, si els equips que desenvolupen els microserveis estan distribuïts al llarg del planeta i no tenen una forta comunicació entre ells, el desenvolupament del microservei està avocat al fracàs.



Il·lustració 17: Filosofia de grups de treball en una arquitectura monolítica vs una arquitectura de microserveis

Els microserveis són més fàcils de monitoritzar que les aplicacions monolítiques, al tindre un microservei desplegat en un host podem saber el percentatge de cpu que utilitza, la ram que necessita, l'espai de disc que va ocupant... Si gestionem be els logs de la nostra aplicació ens serà més senzill de trobar l'errada que té el microservei. A més existeixen ferramentes que ens ajuden a mostrar tota aquesta informació en quasi temps real, una d'elles seria ElasticStack [46]. Amb aquest software *open source* podem monitoritzar el microservei, capturar logs mitjançant patrons i representar-ho tot amb una ferramenta d'explotació de dades inclosa a la suite i que s'anomena Kibana. També existeixen altres ferramentes com Kong Mashape [47], que es un Proxy molt lleuger i escalable, que ens permet posar inclús més capes al damunt, incloent serveis com autenticació, auditoria, disminució de velocitat, monetització d'APIs... A aquest treball no ens anem a parar a gestionar la monitorització dels microserveis però deixaré els enllaços a les corresponents webs per si fora del vostre interés.

Algunes característiques més dels microserveis són [12]:

- Reutilització: amb els microserveis, si aquestos exposen APIs (Application Programming Interface) es poden reutilitzar de manera que siga més fàcil implementar noves funcionalitats.
- Optimitzant el reemplaçament: al tindre l'arquitectura dividida podem rebutjar qualsevol microservei antic per altre nou sencer que implemente diverses i noves funcionalitats respecte l'anterior.
- Utilització de llibreries pròpies o de tercers: A mesura que es desenvolupen els microserveis podem aïllar funcionalitats en diferents llibreries. Ens dona llibertat per a poder compartir funcionalitats entre equips i serveis. Per exemple, una llibreria podria contindre mètodes que ens permeten publicar missatges a una cua de treball AMQP.
- Mòduls: Alguns llenguatges proveeixen les seues tècniques de descomposició de mòduls i que van més enllà de simples llibreries. Un mòdul podria ser per exemple, un mòdul d'autenticació.

Però els microserveis no són la panacea i moltes voltes no poden encaixar en tots els contextos de les aplicacions. Poden existir elements que ens restringesquen el

desacoblament de funcionalitats dintre de l'arquitectura i per altra banda també poden dependre de les infraestructures de les que dispose l'empresa.

A més, no sempre s'ha de recorre a un monòlit o als microserveis, també poden existir solucions hibrides i fins i tot en alguns casos en els que no siga possible migrar l'aplicació.

3.3. Els microserveis són el futur de les arquitectures software?

Una volta definits els dos tipus d'arquitectures anem a resumir els seus avantatges i desavantatges per a resoldre aquesta pregunta que ens formulem.

El primer en que ens anem a fixar es l'estructura interna. Les aplicacions monolítiques estan compostes per 3 capes, en canvi els microserveis podem tindre cada capa en un microservei i és més si existeixen funcionalitats que tinguen els límits del context ben definides, aquestos microserveis encara es poden dividir més.

Encara que l'anterior paràgraf us puga haver pogut fer pujar les ànims, no tot és tan bonic... Comencem per el monòlit, les 3 capes poden estar molt ben definides, tota la llògica estarà en la capa de negoci i la persistència funcionarà de meravella però clar, si comencem a desenvolupar molts serveis al llarg del temps i que damunt de tot, comencen a tindre necessitat d'altres per a completar funcionalitats... Aquesta aplicació que ens havia quedat ben bonica es transforma en un monstre de laberints i mal de caps entre serveis que segurament farà que la nostra aplicació es vaja degradant lentament. Els microserveis no s'escapen d'aquestos problemes. Si no definim bé els límits de context de les funcionalitats a implementar, podem caure en l'error de dividir massa l'aplicació i complicar en excés les comunicacions. Recordem, simplicitat, desacoblament i forta cohesió. No és una tasca senzilla...

Els microserveis son un tipus d'arquitectura que encara està en evolució i per tant no és perfecta però això no vol dir que si es defineixen bé les coses i som conscients del que volem que poc a poc faça la nostra arquitectura, no arribem a bon port.

Com que aquesta tecnologia és encara jove, no totes les empreses es poden plantejar desplegar una arquitectura de microserveis, es tracta de enfocar l'arquitectura a un altre punt de vista totalment diferent al actual de les aplicacions monolítiques. A més, els equips de desenvolupadors deuen d'adaptar-se a aquesta nova filosofia, tenint que aprendre tot el que aquesta porta [12].

Altre punt son les limitacions dels projectes, els microserveis requereixen d'una infraestructura hardware diferent. Fins ara podíem escalar l'aplicació horitzontalment i verticalment amb repetides instàncies de l'aplicació sancera i *n* servidors físics. Als microserveis no ens podem plantejar aquesta infraestructura, no podem tindre un servidor amb multitud de microserveis, això va en contra de la nova filosofia. Es deuran de buscar ferramentes alternatives de desplegament de contenidors i açò una volta més implica un esforç d'aprenentatge e investigació [12].

Recordem el que dèiem fa un moment sobre dividir massa els límits del context de cada funcionalitat. Definir molts mircroserveis que implementet petites funcionalitats implica un major risc d'inconsistència de les dades, per exemple, no és el mateix guardar tota la informació d'un client a un microservei qué la necessitat d'un segon microservei per a completar informació rellevant del mateix. Aquestes inconsistències es poden vore incrementades quan es redefineixen microserveis, si aquestos canvien sencerament la seua funcionalitat podrà afectar a la resta de l'aplicació, devent fer un gran nombre de canvis [12].

Les inconsistències en les dades dels microserveis o les errades humanes que puga tindre un desenvolupador en cada cas poden provocar catàstrofes que en funció de l'arquitectura es poden resoldre a diferent velocitat [12]. A les aplicacions monolítiques encara es pot fer més complicat, trobar l'errada entre la tela d'aranya de serveis dependents pot provocar que llancem a perdre hores i hores buscant a Wally. Per contra, els microserveis són serveis aïllats per tant es pot trobar i controlar l'errada més fàcilment.

Aquestes errades poden provocar la pèrdua total del funcionament en el cas dels monòlits ja que escalar l'errada entre mil dependències de serveis és molt complicat. Els microserveis compleixen la funció de simplicitat ja que si un servei no està disponible, l'aplicació pot limitar la funcionalitat oferida sense deixar de funcionar [12, 45]. Recordem l'exemple de Netflix, pot ser no puga vore la meua llista de series preferides però puc seguir veient qualsevol altra cosa que m'oferisca.

No ens podem oblidar de les llibreries i els mòduls, be implementats per l'empresa o be de tercers. Internament poden canviar i açò suposa altre canvi del monòlit o dels microserveis, pot afectar a un o a molts [12].

I quan trobem l'errada com tornem a tindre l'aplicació en producció? Quan de temps necessitem per a abastir aquest objectiu? Una volta més el monòlit té l'obstacle de ser un "tot en ú" i necessita que l'aplicació sencera s'empaquete, es compile i es desplegue. Aquesta àrdua tasca no ocorre als microserveis, amb els tècniques de desplegament continu, empaquetar, compilar i desplegar un microservei que estiga allotjat per exemple a un contenidor Docker pot ser qüestió d'uns pocs minuts.

Aleshores, m'estudie tot el que puga sobre els microserveis i comence "a les braves" a desenvolupar una arquitectura sense tindre cap experiència?

Per a anar incorporant-se a aquesta nova filosofia, Martin Fowler, James Lewis i Zhamak Dehghani recomanen començar per migrar una aplicació monolítica a una arquitectura de microserveis o desenvolupar primer l'aplicació monolítica i després desmuntar-la. Anar aprenent a aïllar funcionalitats limitant el context i familiaritzar-se amb les tecnologies de comunicació entre ells, així doncs anar iterant repetidament en la descomposició del monòlit i una volta definida ja una arquitectura base de microserveis vore si encara es podria subdividir més les funcionalitats extretes. [48, 49]

Sembla doncs que hem agafat el camí correcte. Més endavant extraurem la conclusió de si ha valgut la pena o no.

En resum, segurament els microserveis siguen l'arquitectura del futur. La tecnologia avança a passos de gegant, les empreses inverteixen molts diners en

desenvolupar ferramentes que suporten aquesta nova filosofia i també van eixint molts altres softwares que ajuden al seu creixement. Spring-boot, Docker, Swarm, Kubernetes, Prometheus, RabbitMQ, MQTT, Netflix, ElasticSearch, Kibana, ...

4. Cas d'estudi: ecoMobility

4.1. Què és ecoMobility?

EcoMobility [3] pretén aportar una solució d'Intel·ligència Artificial en l'àmbit de les ciutats intel·ligents que permeta desplegar una infraestructura IoT per a aplicar polítiques orientades a la reducció de la contaminació ambiental utilitzant tècniques de computació autònoma. Els seus objectius són:

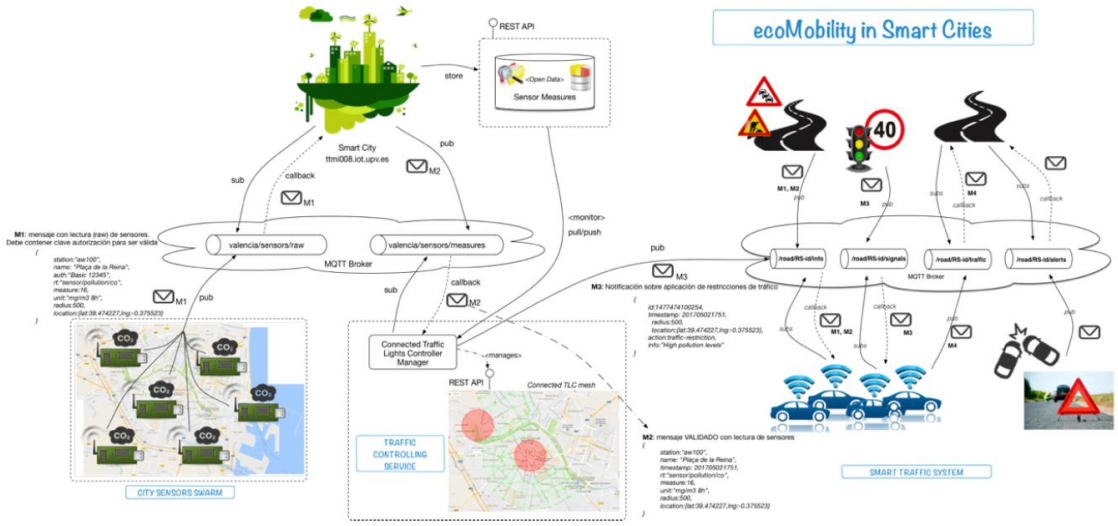
- Reduir el temps de resposta entre la detecció dels nivells contaminants i la repercussió sobre les restriccions de circulació definint plans de mobilitat urbana sostenibles.
- Gestionar el tràfic rodat per a restringir el tràfic rodat i que aquestes restriccions apareguen i desapareguen dinàmicament i en temps real segons la contaminació ambiental de la pròpia ciutat.
- Mantindre informat als ciutadans (vianants, conductors, reguladors de tràfic, etc...) sobre les accions i canvis rellevants en les configuracions de mobilitat.

Aquest últim punt pren una rellevància important ja que pretén establir uns nivells de confiança amb els usuaris (ciutadans) que originen un canvi de mentalitat dintre la població. Per això totes les decisions i accions que fa la infraestructura són directament notificades a tots els usuaris. Per exemple, quan s'identifique que hi ha que tancar o obrir al tràfic una zona en concret per tindre uns nivells alts de pol·lució, es notifica a tots els usuaris interessats per a que siguen conscients de la decisió presa i el motiu que per la qual s'ha originat.

EcoMobility permet ajudar a la gestió social i administrativa de la ciutat, ja que permet delegar certes decisions i responsabilitats a l'arquitectura i que actualment recauen sobre les Administracions Públiques (AAPP) o diferents actors socials. Així doncs, es permeten establir relacions de col·laboració entre ciutadans/infraestructura i aquestes Administracions Públiques per tal d'accelerar i enriquir els processos de presa de decisions atenent a unes polítiques parametritzables.

4.2. Descripció de l'escenari

Actualment, ecoMobility ja té un prototip funcional que implementa parcialment l'arquitectura final del projecte i que està desplegat a la ciutat de València [3].



Il·lustració 18: L'arquitectura monolítica d'ecoMobility

Està compost per una xarxa d'estacions de mesurament Arduino UNO WiFi que notifica les lectures contaminants a la plataforma a través del protocol de missatgeria MQTT. Un procés allotjat a un servidor rep les lectures, les filtra i les retransmet a altre canal de comunicació MQTT amb el servei de gestió dels controladors de semàfors. Aquest s'encarrega de canviar tant la configuració en les restriccions de circulació com de notificar als sistemes de gestió de tràfic i de senyalització els canvis sobre la infraestructura i que son debuts als alts nivells de contaminació.

El servei de gestió de controladors funciona seguint un enfoc de microserveis on cada node es un microservei de la xarxa i està connectat als seus adjacents però no estan desplegats de manera distribuïda ni estan connectats amb la infraestructura física semafòrica.

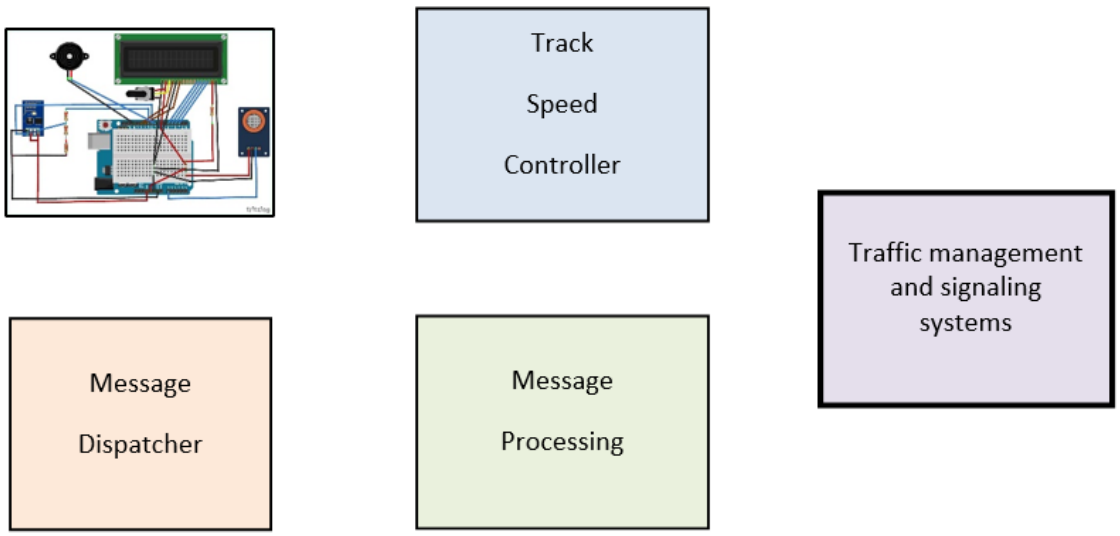
4.3. Serveis extrapolables

Després de definir l'arquitectura actual del projecte ecoMobility, anem a identificar quins son els serveis que anem a extraure del monòlit per a la nova definició de l'arquitectura de microserveis.

Ens fixarem a la part descrita en l'anterior apartat que es refereix des de l'arribada del missatge que genera el sensor, seguit del processament d'aquest missatge i per últim la gestió del nou cas d'us que canviarà la velocitat màxima fixada a la via.

El primer que identifiquem a l'arquitectura és la gestió del missatge. Al monòlit tots aquestos apleguen a un servidor que els filtra i els processa, podem diferenciar el processat de cada missatge en diferents serveis, per exemple, no serà igual el missatge que puga enviar un sensor de contaminació de l'aire que un d'eficiència energètica que mesure el nivell de il·luminació ambiental. Així doncs, per una banda tindrem el microservei que gestione la recepció del missatge (*Message Dispatcher*) i per altra el microservei que filtre i processe la informació (*Message Processing*).

Tal i com es defineix al resum i als objectius, aquest treball consisteix en la migració de part de l'arquitectura i la implementació d'un cas d'us nou, així doncs aquest últim microservei s'encarregarà de la decisió del canvi de velocitat a la via. *Track speed controller* necessitarà el missatge que ha generat el microservei anterior que filtra i processa la informació. Una volta estiga presa la decisió, deurà d'enviar la nova velocitat definida als sistemes de gestió del tràfic i senyalització els canvis sobre la infraestructura.



Il·lustració 19: Identificació dels serveis extrapolables

4.4. Serveis bàsics de l'arquitectura de microserveis.

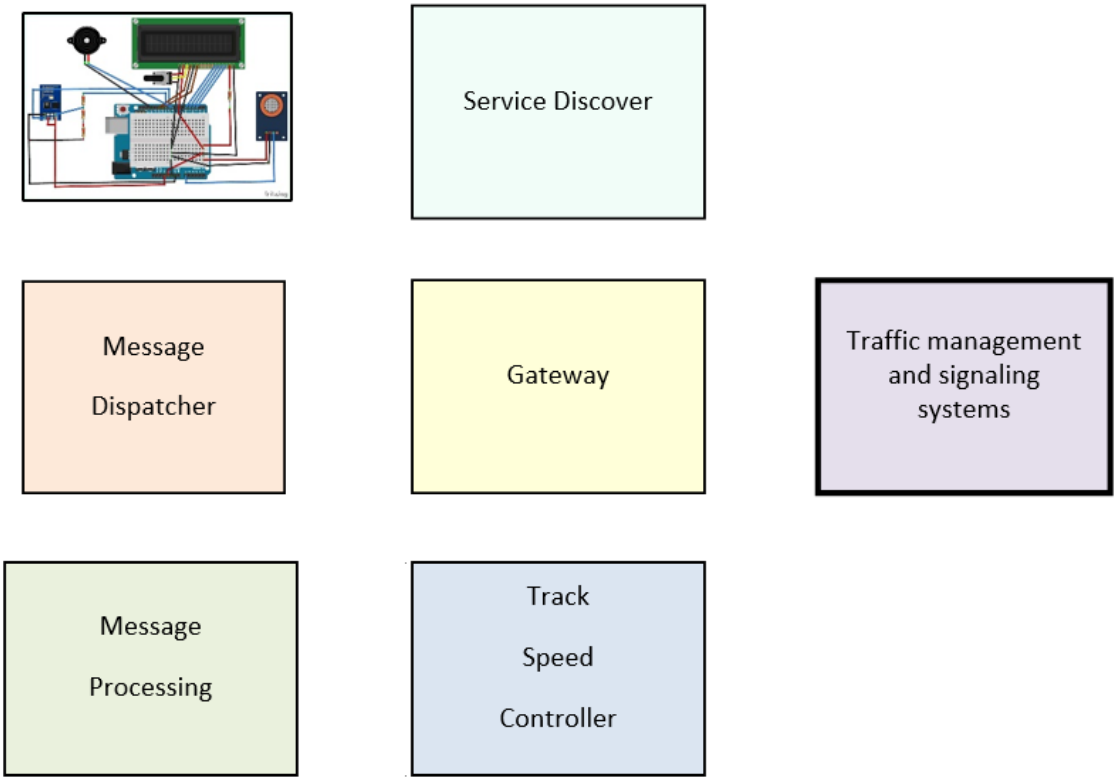
A més dels serveis descrits e identificats en l'anterior punt, es necessària la implementació de dos microserveis nous per a l'orquestració de la nostra arquitectura.

El primer és el *Gateway*, aquest microservei serà l'encarregat de rebre informació dels microserveis per a contestar-li cap a quin altre tenen que atacar. Per exemple, quan el *Gateway* reba un missatge del *Message Dispatcher* el que farà serà redirigir aquesta petició cap al microservei destí encarregat de processar el missatge original. El contingut del mateix el definirem més endavant però bàsicament la informació que contindrà serà la cua de treball a la que el microservei de processament s'ha subscriure.

Per a tindre consciencia de l'estat dels microserveis desplegats serà necessària la implementació d'un microservei que ens faja de descobridor de serveis. Amb l'ajuda d'aquest, el *Gateway* serà conscient dels microserveis disponibles i seguint l'exemple

descrit a l'anterior paràgraf, podrà fer que el microservei *Message Processing* destí s'assabente a quina cua de treball s'ha de subscriure.

La nova distribució de serveis queda per tant així:



Il·lustració 20: Identificació dels serveis necessaris dintre de l'arquitectura

Una part molt important i que es descriurà més endavant són les comunicacions entre els microserveis. De moment s'han definit els microserveis que implementaran les funcionalitats desacoblades del monòlit i els microserveis necessaris per a l'orquestració de la nova arquitectura.

4.5. Descripció de les unitats de negoci de cada microservei.

Una volta definits tant els serveis que podem extraure del monòlit com dels microserveis bàsics necessaris per a la implementació d'una arquitectura de microserveis, es el moment de descriure les funcionalitats de cadascun d'ells. Donat que coneixem el servei que desenvolupen dintre del monòlit, es tracta de re-enunciar-les però encaixant-les amb la nova arquitectura.

4.5.1. Requisits funcionals.

Message Dispatcher: Serà el microservei encarregat d'implementar la part de recepció i encuament de missatges, ací identificarem el tipus de sensor que ens està enviant la informació i l'encuarem a la cua de treball corresponent.

- Definició de l'objecte que faja mapping amb el missatge del sensor.
- Interfície REST per a la recepció del missatge.
- Filtrat del missatge i encolat en la cua corresponent en funció del tipus de sensor.
- Connexió amb missatgeria asíncrona per a l'enviament d'un missatge al *Gateway* amb informació necessària per a que orquestre.
- Persistir els missatges de comunicació per a auditar l'arquitectura.

Gateway: Microservei encarregat d'implementar la part d'orquestració de l'arquitectura. Rebrà missatges dels diferents microserveis i decidirà amb qui comunicar-se per a resoldre el servei que es demana en funció del tipus de sensor.

- Connexió amb Eureka per a rebre tots els microserveis registrats.
- Rebre missatges d'altres microserveis.
- En funció del tipus de microservei, enviar informació al següent per a que puga oferir el seu servei.
- Persistir els missatges de comunicació per a auditar l'arquitectura.

Message Processing: Aquest microservei serà el que processe el missatge, concretament arrodonirà valors rebuts del missatge del sensor.

- Rebre missatges d'altres microserveis.
- Enviar missatges a altres microserveis.
- Processar la informació i encolar el missatge processat a altra cua de treball per a que el microservei de tràfic obtinga el missatge.
- Persistir els missatges de comunicació per a auditar l'arquitectura.

Track Speed Controller: Per últim, aquest microservei s'encarregarà de decidir en funció de les mesures ja processades que haja capturat el sensor, de decidir si es manté, s'augmenta o es disminueix la velocitat de les vies.

- Rebre missatges d'altres microserveis.
- Decidir si canviar la velocitat de les vies de la zona.
- Fer una crida HTTP al sistema que canvia la velocitat de les vies.
- Persistir els missatges de comunicació per a auditar l'arquitectura.

4.5.2. Requisits no funcionals

Com a requisits no funcionals tindríem els que ja de per sí portaria ecoMobility i que serien millorar la qualitat de l'aire i millorar l'eficiència energètica de les ciutats, ajudant tots dos a reduir les emissions contaminants cap a l'atmosfera. Aquestos dos

requisits tenen un impacte directe sobre la ciutadania ja que l'emissió de gasos contaminants destrueix la capa d'ozó i provoca malalties.

L'OMS estima que la contaminació ambiental de l'aire, tan en ciutats com en zones rurals fou la causa de 4,2 milions de morts prematures l'any 2016. Millorant la qualitat de l'aire millorarem la salut cardiovascular i respiratòria de les persones [2].

Els requisits no funcionals respecte a la migració de l'aplicació serien la millora de resposta en cas de sobrecàrrega de missatges de sensors, també la millora davant possibles errades que provoquen la indisponiblitat del sistema ja que podrem reduir la unitat a corregir a un microservei. A més, facilitem la integració i el desplegament continu ja que no es tracta desplegar l'aplicació sencera sinó que podem desplegar noves versions dels microserveis amb les seues noves funcionalitats o refactoritzacions. Millorant l'escalabilitat de la plataforma fent créixer (en un futur) aquells microserveis que més demanda tinguen.

4.6. Solució proposada

Una volta descrites els dos tipus d'arquitectures que en aquest cas, al ser una totalment contraria a l'altra ens serveix per a exemplificar les seues bondats i els seus defectes, es planteja la següent solució.

Amb els serveis identificats es poden ja anar començant a muntar les peces del puzle. Tenim per una banda una serie de "microserveis físics" (sensors) encarregats d'abastir-nos amb missatges que representen la realitat de la contaminació del aire en temps real. Aquestos missatges s'han de processar per a després prendre decisions i aquesta decisió s'ha de comunicar al servei que gestione les accions i senyalitzacions sobre la infraestructura vial.

Així doncs el primer serà fer que l'arquitectura siga conscient del seu entorn, sabent quins microserveis estan disponibles i quina és la informació que aporten per a prendre decisions en el futur. Una volta aconseguit aquest punt serà necessari veure què tenim disponible al missatge i com podem aprofitar-ho per a fer l'arquitectura el més genèrica possible.

Després tindrem que controlar tot el munt de missatges que ens poden aplegar, com fem que no es perda cap missatge? Els microservei *Message Dispatcher* serà l'encarregat de organitzar en cues de treball (encolen els missatges 1 a 1 i s'extrauen 1 a 1) els missatges que apleguen dels sensors.

Ara és el moment on les comunicacions entre microserveis apareixen per primera volta. No sabem quins microserveis estan disponibles dintre de l'arquitectura, d'això n'és conscient solament el microservei *Gateway*. Aleshores és necessari que implementem un canal de comunicació entre aquestos per a poder fer arribar fins al microservei *Message Processing* el missatge a processar.

L'arquitectura ja és conscient d'en quin punt es troba el missatge i sap que necessita saber si el microservei *Message Processing* està disponible. En cas afirmatiu el

Gateway enviarà un missatge per altre canal de comunicació definit fins al microservei destí, el qual extraurà de la cua de treball els missatges dels sensors.

Es processarà el missatge reduint el nombre de decimals que el sensor ens haja pogut enviar en excés. Acabada la feina, el següent serà fer arribar el missatge processat fins al microservei *Track Speed Controller*. Per tant, el que farà el microservei *Message Processing* serà enviar el missatge processat a altra cua de treball diferent a l'anterior per a que gràcies a la ajuda del *Gateway* el missatge arribe fins a *Track Speed Controller*.

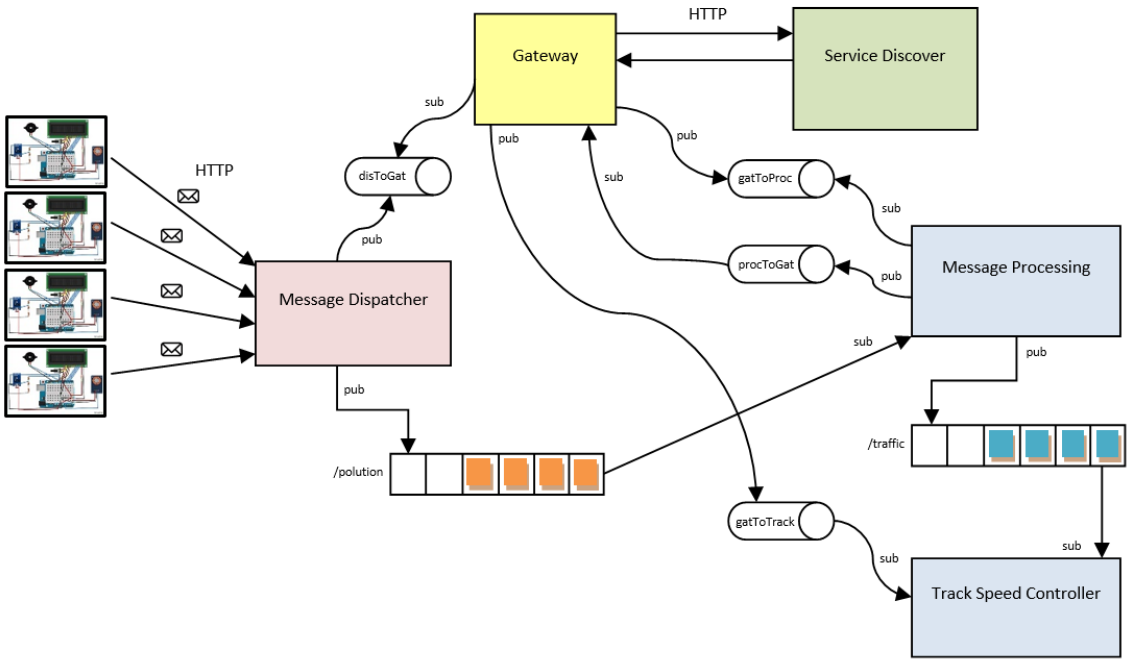
Gràcies a la comunicació amb el *Gateway*, el microservei *Track Speed Controller* es subscriurà a la cua de treball on els missatges processats estaran esperant e informarà del canvi de la velocitat de la via en funció de la qualitat de l'aire. Com no és conscient de la velocitat actual de la via, delega aquesta decisió als sistemes de gestió de tràfic i de senyalització.

5. Disseny de la solució

Una volta definits els límits de context que representa la funcionalitat que implementarà cadascun dels microserveis juntament amb els requisits, és el moment de definir el disseny de la nova arquitectura de microserveis.

5.1. Disseny de l'arquitectura

A continuació podem observar gràficament com és l'arquitectura de microserveis plantejada. Ací és on es mostra per primera volta com entra en acció la missatgeria asíncrona.



Il·lustració 21: Disseny de l'arquitectura de microserveis

Com s'ha comentat més enrere, els tres protocols de comunicació entre microserveis que es troben a l'arquitectura són:

- El protocol HTTP: Utilitzat per a enviar/rebre missatges dels sensors i també per a comunicar-se amb el *Service Discover*.
- Cues de treball: Aquestes van definides per el caràcter "/" i tenen forma de caselles. Seran les encarregades d'anar encolant i enviant els missatges dels sensors i també els missatges ja processats.
- Missatgeria asíncrona: S'utilitzarà per a la comunicació entre microserveis d'informació rellevant sobre les cues de treball. Ens ajudaran a que els microserveis siguen conscients dels altres microseveis i orquestrar el processament de missatges.

Una decisió molt important ha sigut la de encasellar els diferents tipus de missatges que puguen arribar. Aquestos vindran definits per el camp **rt** del missatge rebut des de el sensor, amb aquest camp podem obrir la possibilitat a diferents lectors/processadors de missatges de diferents tipus, assignant cues de treball per **rt** i modificant el seu valor una volta processat el missatge per a orquestrar el següent pas.

Per a aquest cas per exemple, hem definit que quan arribe del sensor s'anomene */polution* i en el moment es processe el seu valor canvie a */trafficSpeed*.

5.2. Cas d'us a implementar

El prototip d'ecoMobility que actualment està desplegat a la ciutat de València, gestiona el tràfic llegint les lectures dels sensors i canviant la circulació dels vehicles a motor modificant els semàfors i així restringir les zones més afectades. També comunica

als sistemes de gestió i senyalització del tràfic els canvis fets sobre l'arquitectura debut a la detecció dels nivells de contaminació.

Com es pot apreciar, anem a aprofitar part de l'arquitectura ja migrada per a implementar aquest nou cas d'ús. Durant el treball s'ha mencionat ja que no s'ha migrat el 100% del monòlit, s'ha migrat les parts que ens interessaven per a poder implementar aquesta nova funcionalitat i s'ha deixat la migració de la resta del monòlit per al futur.

Per a conèixer els llindars dels nivells de pol·lució mínims i màxims s'ha utilitzat la guia de la OMS [50] que descriu quins nivells són molt perjudicials per als humans i quins menys. Aquesta guia defineix els nivells màxims i mínims per a mesures cada 24 hores i també anuals als que ens veiem sotmesos i que es detallen per a els següents elements:

- Material particulat.
- Ozó.
- Diòxid de nitrogen.
- Diòxid de sofre.

Per a simplificar el disseny inicial de l'arquitectura, es va a emprar solament els nivells màxims i mínims del material particulat comprés entre tamanys de 2,5µ i 10µ (MP_{10}).

MP_{10}:	**20 $\mu g/m^3$, media anual** **50 $\mu g/m^3$, media de 24 horas**

Il·lustració 22: Mesures mínimes de material particulat

Com es pot vore al document emés per la OMS es recomana que els nivell d'aquestos elements siga menor de 50µ cada 24 hores. Com a mesura inicial s'ha proposat que quan es supere el nivell de contaminació en 70µ reduirem el tràfic de la zona afectada fins que les mesures baixen a 30µ. Evidentment aquestos números no són fixes, pot ser siguen molt restrictives i es tinguen que modificar, és quelcom que es veurà en un futur quan estiga desplegada aquesta nova funcionalitat inclosa a la nova arquitectura.

Per tant, deurem de processar el missatge, enviar-lo a la cua de treball /polution per a després comunicar-se amb el *Gateway*, aquest li comunicarà al servei encarregat de processar eixe missatge de quina cua ha d'extraure'l. Processarà el missatge, l'encolarà altra cua de treball i tornarem a comunicar amb el *Gateway* per a poder fer saber a *Traffic Speed Controller* sobre quina cua ha d'extraure el missatge processat per a calcular la nova velocitat, per últim, una volta ja presa la decisió enviarà una petició HTTP informant de la necessitat del canvi de la velocitat de via.

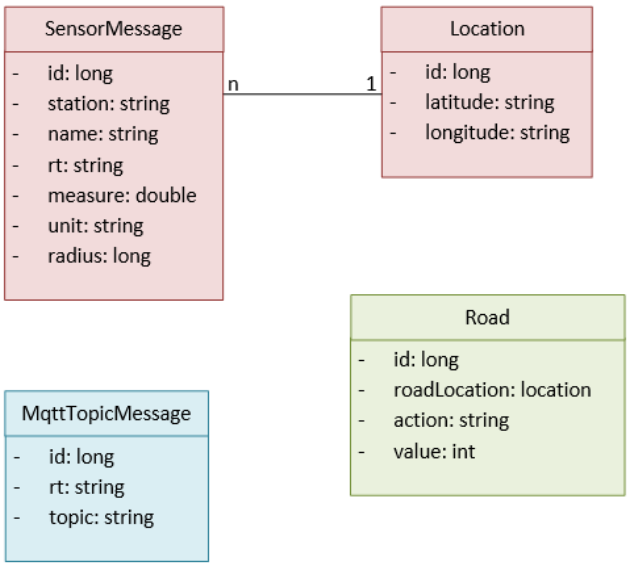
5.3. Disseny detallat dels components

Una volta definida l'arquitectura, el cas d'ús a implementar i les tecnologies que anem a utilitzar, avancem doncs cap als components que la composen descrivint el

disseny estàtic, l'estructura bàsica de cada microservei i el comportament dinàmic de l'aplicació.

5.3.1. Diagrama de classes.

L'arquitectura base que proposem està composta per 4 classes que serviran per a rebre missatges dels sensors, establir la informació que comparteixen per a les comunicacions entre microserveis i per últim per a enviar informació final al sistema de gestió i senyalització del tràfic.



Il·lustració 23: Diagrama de classes

La classe SensorMessage conté els següents camps:

- Id: Serveix com a identificador de l'objecte per a la persistència.
- Station: Nom de l'estació (sensor).
- Name: Nom de l'ubicació.
- Rt: Camp identificador del tipus de missatge.
- Measure: Camp numèric amb la mesura presa per el sensor.
- Unit: Unitat en la que s'ha mesurat.
- Radius: Radi d'alcanç del sensor.

La classe Location conté els següents camps:

- Id: Serveix com a identificador de l'objecte per a la persistència.
- Latitude: Defineix la latitud d'un punt geogràfic.
- Longitude: Defineix la longitud d'un punt geogràfic.

La classe MqttTopcMessage:

- Id: Serveix com a identificador de l'objecte per a la persistència.
- Rt: Camp identificador del tipus de missatge
- Topic: Tòpic sobre el qual ha sigut publicat aquest missatge.

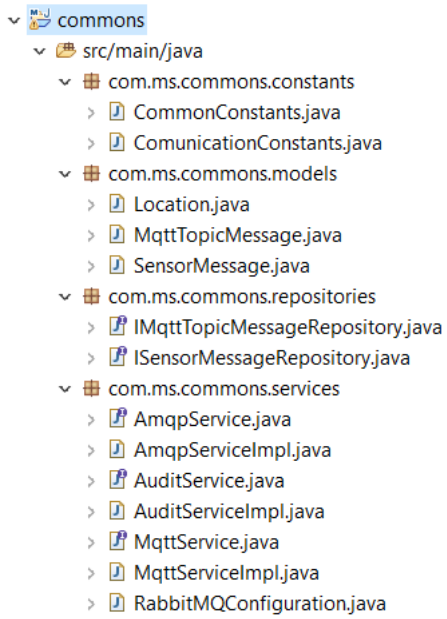
La classe Road:

- Id: Serveix com a identificador de l'objecte per a la persistència.
- RoadLocation: Localització del sensor.
- Action: Acció a implementar pot ser augmentar o disminuir la velocitat.
- Value: Nou valor que ha de tindre el tram de carretera/via.

5.3.2. La llibreria compartida *commons*

Seguint la filosofia dels microserveis, s'han agrupat funcionalitats comuns a una llibreria anomenada *commons*. Aquesta contindrà valors constants que contenen informació important sobre les comunicacions, serveis que implementaran funcionalitats que facilitaran les comunicacions amb MQTT i AMQP i també serveis d'auditoria de missatges.

L'estructura d'aquesta llibreria és la següent:



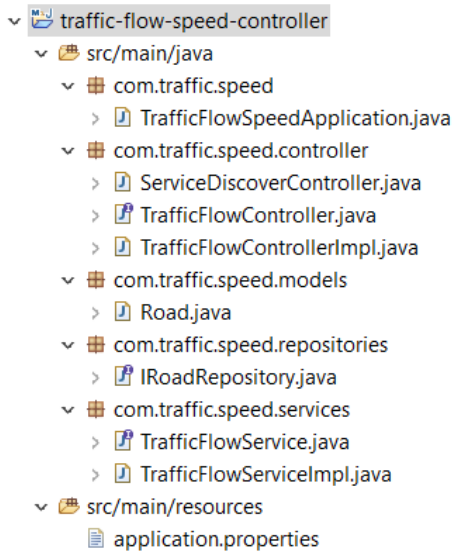
Il·lustració 24: Estructura de la llibreria compartida commons

El procés que s'ha dut per a la implementació d'aquesta llibreria ha vingut donat per l'observació mentre construïa els microserveis de funcionalitats que es podien fer genèriques i es podien desacoblar dels microserveis.

5.3.3. Definició de l'estructura de cada microservei

Anem a definir solament el microservei sobre el que implementarem el nou cas d'ús. En essència tots els microserveis excepte el que implementa el servidor Eureka tenen la mateixa estructura, la diferència ve en que a Eureka solament li cal la classe principal de l'aplicació, una anotació i la configuració mínima del servidor. El que es tradueix en el fitxer *aplication.properties* útil per a la configuració d'*Spring-boot* i aquestes dues anotacions:

- @EnableEurekaServer
- @SpringBootApplication



Il·lustració 25: Estructura exemple d'un microservei de l'arquitectura

Així doncs, com es pot apreciar a la imatge anterior, l'estructura de paquets dels nostres microserveis estarà composta per:

- La classe principal que tindrà el format {ApplicationName}Application.java
- Un paquet que contindrà el controlador REST API del microservei. Com que no anem a utilitzar quasi comunicacions HTTP només contindran un "hola món!" i es deixara per a millores futures. Un controlador que tindran tots serà *ServiceDiscoverController.java* que juntament amb l'anotació @EnableDiscoveryClient continguda a la classe principal, seran les encarregades del registre automàtic del microservei a Eureka.

- En cas que el microservei tinga models propis del mateix, tindrem un paquet *models* on es definiran aquells objectes que utilitzarà i altre paquet *repositories* per a persistir aquestos models.
- Altre paquet *services* on s'inclouran les classes que implementen la llògica del microservei.
- I per últim, un fitxer de configuració d'*Spring-boot* que s'anomena *application.properties* i que contindrà propietats de l'aplicació i propietats per a la configuració amb la connexió de la base de dades i amb el servidor Eureka.

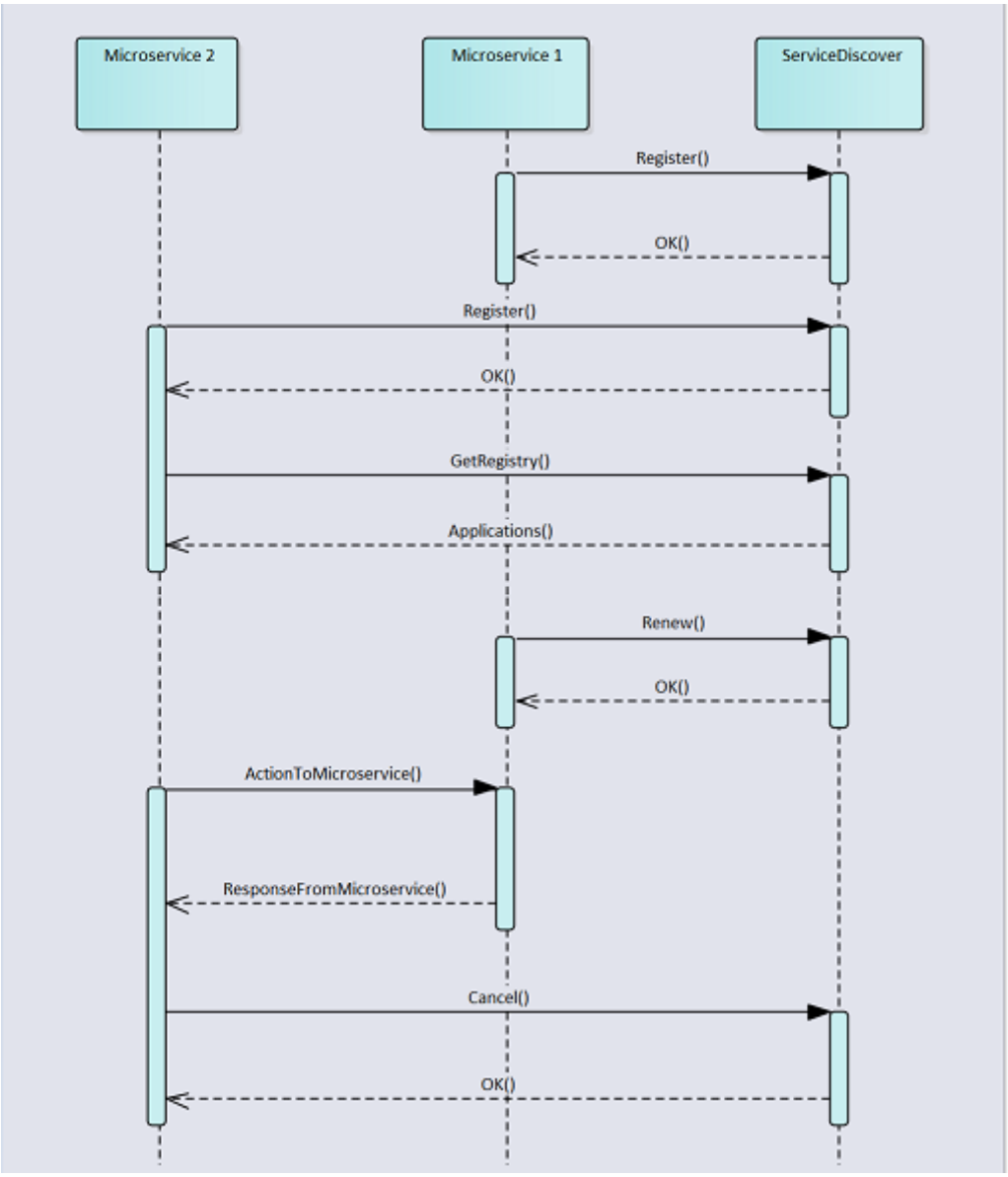
5.3.4. Comportament dinàmic de l'aplicació.

El comportament dinàmic de l'aplicació està definit a dos punts clarament diferenciats: el registre del microservei en el *Service Discover Eureka* i la implementació de la part final del cas d'us del canvi de la velocitat de la via. Tots dos engloben les accions i les comunicacions provocades per cada acció amb la seua consegüent reacció.

Registre de client Eureka a Eureka Server [56]

Comencem doncs per el registre del client d'Eureka al servidor. Aquest es fa de forma automàtica gràcies a la implementació del client Java que ofereix i que es divideix en 4 fases diferenciades: registre, registre de cerca, renovació, i cancel·lació.

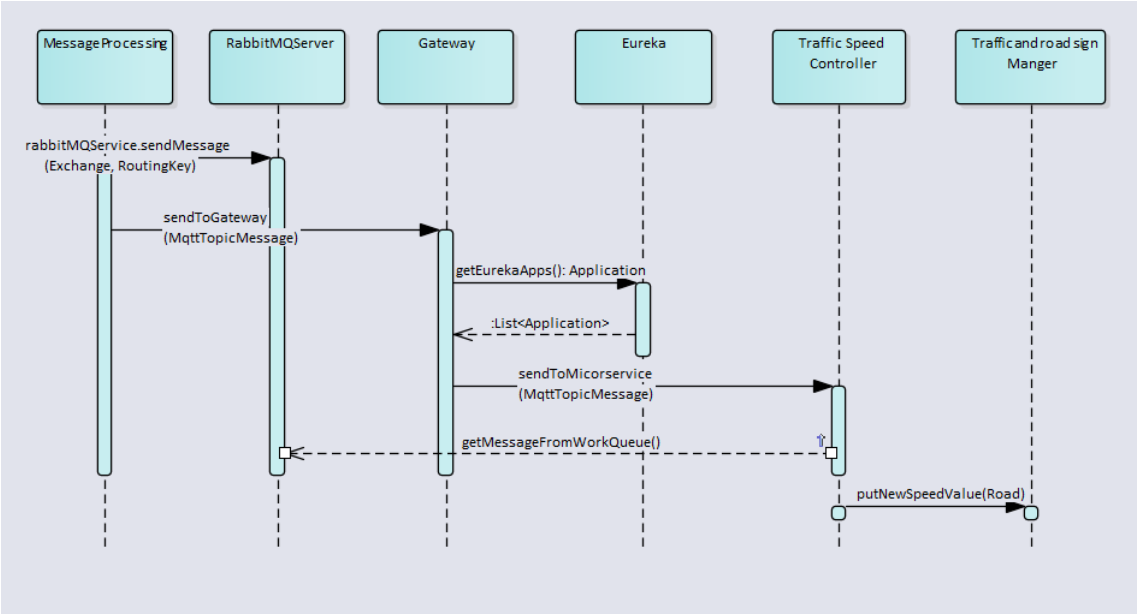
- Registre: el client d'Eureka registra tota la informació de la instancia en execució al servidor Eureka. Aquesta acció ocorre al primer batec (després de 30 segons).
- Registre de cerca: el servidor d'Eureka cachea el registre d'informació localment, aquesta informació és actualitzada cada 30 segons actualitzant els camps modificats dintre d'aquest interval. Serà el registre que utilitzen els clients que vulguen conèixer els microserveis registrats.
- Renovació: el client d'Eureka necessita renovar-se amb batecs cada 30 segons. La renovació informa al servidor que la instancia encara està en execució. Si el servidor no obté cap renovació als 90 segons, lleva la instancia del registre de cerca de microserveis.
- Cancel·lació: el client d'Eureka envia una petició de cancel·lació quan la instancia del microservei s'atura. Açò fa que la instancia mantinguda dintre del registre del servidor d'Eureka s'elimine.



Il·lustració 26: Comportament dinàmic enregistrament d'aplicació a Eureka

Processament de missatge i gestió de la velocitat de la via.

Tot seguit anem a descriure el comportament de l'aplicació que comprén la lectura del missatge fins a l'enviament HTTP amb l'ordre del canvi de velocitat de la via. Per a arribar fins ací el missatge ja ha arribat des de un sensor fins al *Message Dispatcher*, encolant el missatge sense processar a la cua de treball corresponent i comunicant al *Gateway* el tipus de missatge que ha arribat i que vindrà marcat per el camp **rt**. Així doncs una volta rebuda la petició que haurà efectuat contra Eureka, es comunicarà amb el microservei *Message Processing* per a fer-li saber a quina cua de treball s'ha de subscriure.



Il·lustració 27: Comportament dinàmic processament de missatge i gestió de la velocitat

6. Implementació

Durant aquest capítol es van a detallar els passos seguits per a la implementació de l'arquitectura. Aquestos passos es produeixen gràcies a la planificació exposada al començament del treball.

Per a posar en context la totalitat d'aquest apartat, la implementació de l'arquitectura al començament ha tingut una gran corba d'aprenentatge durant els dos primers microserveis ja que aquestos engloben la base de la resta de l'arquitectura. És a dir, hem implementat funcionalitats com l'enregistrament amb Eureka, la connexió i enviament de missatges tant a AMQP com MQTT i l'auditoria de les accions realitzades.

El procés d'implementació de la llibreria s'ha dut durant tot el treball fins a la seua finalització, ja que com s'ha comentat anteriorment, ha sigut fruit de l'extracció de serveis genèrics utilitzats a tots els microserveis i la declaració de constants compartides. Per a facilitar la comprensió de l'arquitectura, serà el primer que es descriurà només acabar la instal·lació i configuració del broker RabbitMQ.

Així doncs, es comença per explicar com instal·lar i configurar el broker RabbitMQ, definint els requisits d'instal·lació per a continuar amb la instal·lació i configuració del *plugin* MQTT.

A continuació s'extrauran i es descriuran les tres funcionalitats internes als microserveis que implementa la llibreria pròpia *commons* mostrant el codi més important d'aquesta.

Seguidament es detallaran cronològicament la implementació de cadascun dels microserveis i el codi més rellevant. Aquestos son molt senzills ja que la càrrega forta l'hem extreta a la llibreria i els microserveis implementen funcionalitats senzilles molt desacoblades.

6.1. Instal·lació i configuració del broker RabbitMQ

Comencem per una peça clau dintre de l'arquitectura. Primer és important conèixer quins són els requisits mínims software i hardware per a la instal·lació de RabbitMQ.

El sistema operatiu mínim que recomanen és un CentOS 5 però també son compatibles la majoria de distribucions Linux. [51, 52]

- Sistemes operatius Linux:
  - Ubuntu 14.04, Debian Jessie, Suse.
- Sistemes operatius Windows:
  - NT (fins a Windows 10) o Server 2003 (fins a 2016).
- Mac
  - MacOS X 10.6

A més serà necessari que tinguem instal·lat l'entorn de desenvolupament Erlang/OTP. Per a la versió 3.7.4 de RabbitMQ que utilitza l'arquitectura es requereix les versions que van compreses entre la versió 19.3.6.4 i la versió 20.3 [53]. Versions més anteriors o més recents no són compatibles i per tant no s'assegura el correcte funcionament del broker.

Una volta vistos els requisits, és moment d'instal·lar el servidor RabbitMQ [54]. Mitjançant la consola del sistema operatiu, s'executen els comandaments necessaris per a instal·lar les dependències:

```
#Instal·lació de dependències:

apt-get install erlang-diameter

apt-get install erlang-nox


#Instal·lació de la suite:

apt-get install esl-erlang
```

Script 1: Comandaments d'instal·lació de dependències de RabbitMQ

Una volta instal·lades les dependències directes necessàries per a l'execució del servidor, és moment d'instal·lar i configurar RabbitMQ.

```
#Afegim el repositori al sistema operatiu:

echo "deb https://dl.bintray.com/rabbitmq/debian xenial main erlang" | sudo tee
/etc/apt/sources.list.d/bintray.rabbitmq.list

#Executem el comandament d'instal·lació:

sudo apt-get install rabbitmq-server
```

Script 2: Comandaments d'instal·lació de RabbitMQ

Executant l'anterior comandament ja ens instal·la el servidor com a servei del sistema, per la qual cosa, no ens tindrem que preocupar d'executar-lo cada volta que iniciem l'ordinador ni tancar-lo cada volta que l'apaguem. Continuem la instal·lació afegint el *plugin* MQTT al broker i que funcionarà una capa per damunt del protocol AMQP. [55]

```
#Instal·lació del plugin MQTT:

rabbitmq-plugins enable rabbitmq_mqtt
```

Script 3: Comandament d'instal·lació del plugin MQTT a RabbitMQ

6.2. Activació i configuració del *plugin* MQTT

Després de la instal·lació del *plugin*, es defineix un usuari que només tindrà permisos de configuració, lectura i escriptura sobre l'*exchange amq.topic*, així s'està limitant l'accés del protocol per a que no puga arribar fins a les demés *exchanges* destinades a altres protocols.

```
#Creació d'un usuari MQTT:

rabbitmqctl add_user mqtt mqtt

rabbitmqctl set_permissions -p / mqtt "amq.topic" " amq.topic " " amq.topic "

rabbitmqctl set_user_tags mqtt management
```

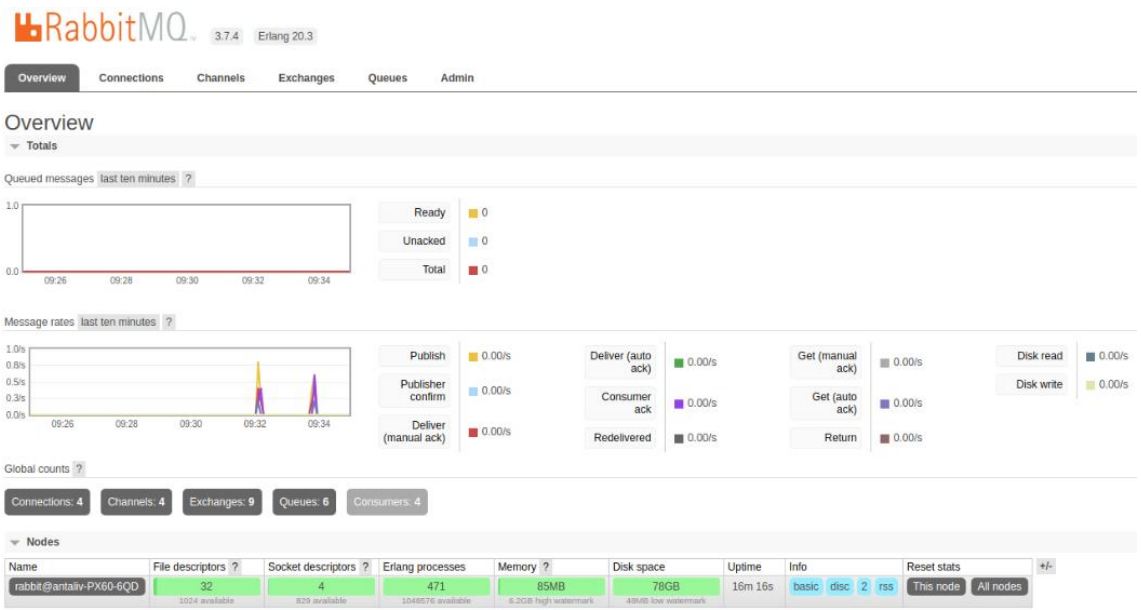
Script 4: Creació d'un usuari per a les comunicacions MQTT

Per últim, s'afegixen les configuracions bàsiques al fitxer que llegeix el servidor quan s'arranca i que configura algunes propietats bàsiques del mateix. Amb el següent fragment es defineix la configuració bàsica per al *plugin* MQTT.

```
listeners.tcp.default = 5672

mqtt.default_user     = guest

mqtt.default_pass     = guest

mqtt.allow_anonymous  = true

mqtt.vhost            = /

mqtt.exchange         = amq.topic

mqtt.subscription_ttl = 86400000

mqtt.prefetch         = 10

mqtt.listeners.ssl    = none

mqtt.listeners.tcp.default = 1883

mqtt.tcp_listen_options.backlog = 128

mqtt.tcp_listen_options.nodelay = true
```

Script 5: Configuració de rabbitmq.conf

A partir d'ací ja està al nostre abast el servidor configurat i per tant es disposa de la interfície web amb la que es poden gestionar les *exchanges*, les *queues* fent operacions de creació, esborrar missatges, esborrar *exchanges*, esborrar *queues*, unir *queues* amb *exchanges* i més... A l'arquitectura totes aquestes es crearan mitjançant la llibreria.



Il·lustració 28: Interfície web del broker RabbitMQ del treball

6.3. Maven

Maven és una ferramenta de gestió i comprensió de projectes software. Està basat en el concepte de model d'objecte de projecte (POM) i pot gestionar la creació, la documentació i la construcció d'un projecte a partir d'una peça central d'informació [58].

Aquesta ferramenta es pot afegir com a *plugin* al IDE que s'utilitza per a desenvolupar l'arquitectura, Eclipse. Disposa de ferramentes que ajuden a la gestió de dependències amb llibreries internes del projecte.

Així doncs tindrem un *pom.xml* dintre de cada projecte que definirà les dependències amb les llibreries que ens ajudaran a crear els microserveis. Cal destacar que quasi totes les dependències han sigut mogudes a la llibreria compartida *commons* ja que dintre d'aquest projecte es defineixen serveis comuns a tots els microserveis i per tant facilita la implementació de la resta de projectes.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.test</groupId>
    <artifactId>microservice</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>microservice</name>
    <url>http://maven.apache.org</url>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.RELEASE</version>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Script 6: Exemple de pom.xml d'un microservei

6.4. Implementació de la llibreria *commons*

Aquesta llibreria és clau al projecte, conté la base de les comunicacions de tots els microserveis i a més algunes llibreries bàsiques comuns dintre del seu *pom.xml*. Com aquesta llibreria s'importa als demés projectes, aquestos tindran totes les llibreries heretades que continga la llibreria *commons* dintre del seu *pom.xml*.

Commons ens ofereix la possibilitat d'agrupar les funcionalitats que hem identificat durant la implementació dels microserveis i que es repeteixen dintre de l'arquitectura. Aquestos serveis han sigut modificats per a fer-los el més genèrics possible facilitant la seua integració amb cada microservei. Si en el futur aquesta llibreria evoluciona, l'impacte seria menor que si no haguérem extret cap servei, doncs en volta

de canviar el mateix tros de codi replicat que ofereix el servei sancer soles tenim que canviar la crida al mètode de la llibreria, el que fa que siga més mantenible.

A continuació descriurem les parts de les que es composa *commons*: constants, models, repositoris i serveis.

Constants

Com a bona pràctica de programació en Java, definirem totes les constants en dues classes una dedicada a les comunicacions i altra a les variables comuns dels microserveis. Utilitzem aquesta bona pràctica perquè és molt més mantenible canviar una constant que no la mateixa *String* allà on s'utilitze.

```
package com.ms.commons.constants;

public class ComunicationConstants {

    //RabbitMQ Constants
    public static final String RABBITMQ_BROKER_USER = "guest";
    public static final String RABBITMQ_BROKER_PASSWORD = "guest";
    public static final String RABBITMQ_EXCHANGE =
"microservice.exchange";
    public static final String RABBITMQ_ROUTING_KEY = "messageQueued";
    public static final String RABBITMQ_HOST = "localhost";
    public static final String RABBITMQ_DIRECT_EXCHANGE = "direct";
    public static final String RABBITMQ_SLASH = "/";

    //Polution Constants
    public static final String POLUTION_ROUTING_KEY = "polutionMessage";

    //MQTT Constants
    public static final String MQTT_BROKER_URL = "tcp://localhost:1883";

    public static final String MQTT_MANAGER_TOPIC = "dispToGat";
    public static final String MQTT_MANAGER_TO_POLUTION_TOPIC =
"gatToProc";
    public static final String MQTT_MANAGER_TO_TRAFFIC_TOPIC =
"procToGat";
    public static final String MQTT_POLUTION_TO_MANAGER_TOPIC =
```

Script 7: Constants de comunicació definides a commons

Models

Per altra banda també conté els objectes bàsics definits a l'apartat del disseny de classes: SensorMessage, MqttTopicMessage i Location. A continuació definirem algunes anotacions utilitzades dintre d'aquestos objectes i que Hibernate interpretarà per a crear la base de dades. Per simplicitat s'han eliminat els paquets que importa, la majoria són de la llibreria javax.

```java
@Entity
public class SensorMessage implements Serializable{

    private static final long serialVersionUID = 3716844901778602215L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotBlank
    private String station;

    @NotBlank
    private String name;

    @NotBlank
    private String rt;

    @NotNull
    private Double measure;

    @NotBlank
    private String unit;

    @NotNull
    private Long radius;

    @NotNull
    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private Location location;

    @Column(nullable = false, updatable = false)
    @CreationTimestamp
    private Date creationDate;
```

Script 8: Exemple del model SensorMessage

Dintre de l'anterior tros de codi podem veure diferents anotacions, la més important és @Entity que defineix que aquest objecte és una entitat JPA i que per defecte la taula serà el nom de la classe. Podem personalitzar aquest últim amb l'anotació @Table però no és el cas.

L'anotació @Id fa que JPA reconega aquest camp com a l'id de l'objecte. Just avall d'aquesta anotació tenim @GeneratedValue que indica que el id s'ha de crear automàticament.

A continuació tenim dos anotacions que serveixen per a limitar que els objectes no s'escriguen amb camps buits. @NotBlank serveix per a comprovar que una String no és nul·la i que la seua longitud és major que zero, i @NotNull serveix per a comprovar

que una col·lecció, un Map, un Array o qualsevol Objecte no és nul i per tant no estiga buit.

Per últim descrivim les anotacions que conté el camp *creationDate*. La anotació @Column serveix per a definir una columna, en aquest cas la columna no pot ser nul·la i no serà un camp actualitzable. Justament davall tenim l'anotació @CreationTimestamp que persistirà automàticament el timestamp (data i hora) en que s'ha persistit l'objecte.

Repository

Aquestos objectes definits dintre del *package models* necessiten ajuda per a poder persistir-se, per tant s'han definit una serie de classes per a persistir cada objecte amb anotacions bàsiques que complesquen aquesta funció.

```
@Repository
@Transactional
public interface IMqttTopicMessageRepository extends
JpaRepository<MqttTopicMessage, Long>{

}
```

Script 9: Configuració d'un repository

L'anotació @Repository i la interfície JpaRepository ens ajudaran a complir aquest objectiu. L'anotació ens defineix que aquesta classe va a actuar com a *Data Access Object* (DAO) i que amb l'ajuda de la interfície JpaRepository implementa operacions bàsiques (guardat, búsqueda, i esborrat) per als objectes MqttTopicMessage.

Per a poder emmagatzemar qualsevol model, primer s'han de crear les bases de dades de cada microservei. Durem aquesta tasca a terme creant primer la base de dades des de una consola del sistema contra un servidor mysql instal·lat en local, després afegim al pom.xml de *commons* el connector java i per últim al fitxer de propietats d'Spring *application.properties* definirem la url i algunes propietats necessàries per a connectar amb la base de dades.

```
create database polutionMS default character set utf8 default collate
utf8_general_ci;

use polutionMS;

create user 'polution'@'localhost' identified by 'polutionMS2018';

grant all on polutionMS.* to 'polution'@'localhost';
```

Script 10: Exemple d'script de creació de base de dades d'un microservei

```
#Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url = jdbc:mysql://localhost:3306/polutionMS?useSSL=false
spring.datasource.username = polution
spring.datasource.password = polutionMS2018

#Hibernate properties
#The SQL dialect makes Hibernate generate better SQL for the chosen
database
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5Dialect

#Hibernate ddl auto (create, create-drop, validate, uptdate)
spring.jpa.hibernate.ddl-auto = update
```

Script 11: Configuració de jpa e hibernate a application.properties

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.6</version>
</dependency>
```

Script 12: Definició de llibreria mysql-connector-java al pom.xml

Services: Auditoria

Per últim, es defineixen els tres serveis comuns i que implementen l'auditoria i les comunicacions. Per a poder oferir aquestos serveis a la resta de microserveis ho hem aconseguit amb les anotacions @Component i @Services.

- @Component és una anotació que li indica a Spring durant l'inici de l'execució què les classes anotades amb aquesta anotació són components de l'aplicació i per tant deuen injectar-se per a pertànyer al context de l'aplicació.
- @Service, al igual que totes les altres anotacions de classes, són meta-anotacions de @Component i en aquest cas serveix per a exposar les interfícies com un model que implementa la lògica de l'aplicació.

```
@Service
public interface AuditService {
    public void saveMqttMessage(MqttTopicMessage mqttMessage);
    public void saveSensorMessage (SensorMessage sensorMessage);

}
```

Script 13: Exemple de definició d'un servei

```
@Component
public class AuditServiceImpl implements AuditService{

    @Autowired
    ISensorMessageRepository sensorMessageRepository;

    @Autowired
    IMqttTopicMessageRepository mqttTopicMessageRepository;

    @Override
    public void saveSensorMessage(SensorMessage sensorMessage) {
        sensorMessageRepository.save(sensorMessage);
    }

    @Override
    public void saveMqttMessage(MqttTopicMessage mqttMessage) {
        mqttTopicMessageRepository.save(mqttMessage);
    }

}
```

Script 14: Exemple de definició d'un component

Com comentàvem abans, a la interfície exposem els dos serveis (@Service) que implementarem (@Component). La classe *AuditServiceImpl* pretén persistir els objectes que s'utilitzen per a la comunicació entre microserveis, per a poder arribar fins a la capa de persistència, haurem d'injectar les classes anotades amb @Repository. Aquesta tasca l'abastim amb l'ajuda de l'anotació @Autowired que injecta els serveis d'auditoria afegits al context de l'aplicació durant l'inici de la mateixa, per tant una volta injectats els repositoris podem utilitzar les funcions que implementen d'emmagatzemament dels objectes.

```
mysql> select * from mqtt_topic_message;
+----+---------------------+-----------+-----------+
| id | creation_date       | rt        | topic     |
+----+---------------------+-----------+-----------+
|  1 | 2018-09-07 10:27:55 | /polution | dispToGat |
|  4 | 2018-09-07 10:28:10 | /polution | dispToGat |
+----+---------------------+-----------+-----------+
2 rows in set (0,00 sec)

mysql> select * from sensor_message;
+----+---------------------+---------+--------------------+--------+-----------+-----------------+------+-------------+
| id | creation_date       | measure | name               | radius | rt        | station         | unit | location_id |
+----+---------------------+---------+--------------------+--------+-----------+-----------------+------+-------------+
|  2 | 2018-09-07 10:27:55 |      54 | Plaça de la Reina  |     20 | /polution | StationMolonaTu | cm   |           3 |
|  5 | 2018-09-07 10:28:10 |      30 | Plaça de la Reina  |     20 | /polution | StationMolonaTu | cm   |           6 |
+----+---------------------+---------+--------------------+--------+-----------+-----------------+------+-------------+
2 rows in set (0,00 sec)

mysql> select * from location;
+----+-----------+-----------+
| id | latitude  | longitude |
+----+-----------+-----------+
|  3 | 39.474647 | -0.375488 |
|  6 | 39.474647 | -0.375488 |
+----+-----------+-----------+
2 rows in set (0,00 sec)
```

Il·lustració 29: Exemple de dades emmagatzemades sobre l'auditoria del microservei Message Dispatcher

Services: Missatgeria MQTT

Continuem la implementació de *commons* amb la missatgeria asíncrona que ens ofereix MQTT. Aquest servei consta solament d'un mètode públic dintre dels serveis exposats i que serveix per a publicar un missatge a un tòpic en concret.

```
@Service
public interface MqttService {

    public void publish(String clientId, String payload, String topicName,
int qos);

}
```

Script 15: Implementació del servei MQTT

Com es pot apreciar, és una interfície molt senzilla. A quasi tots els serveis, les seues interfícies no exposen més de 2-3 mètodes, així es busca la senzillesa del codi que els microserveis volen abastir. Per a la implementació d'aquest mètode hem dividit la funcionalitat que ofereix en dos parts: la connexió amb MQTT i la publicació del missatge.

```
@Component
public class MqttServiceImpl implements MqttService {

    private MemoryPersistence persistence = new MemoryPersistence();

    private MqttClient connectMQTTtoRabbitMQ(String clientId) {

        //Configurem la connexió amb el plugin MQTT
        try {
            MqttClient client = new
MqttClient(ComunicationConstants.MQTT_BROKER_URL, clientId, persistence);
            MqttConnectOptions connOpts = new MqttConnectOptions();
            connOpts.setCleanSession(true);
            connOpts.setUserName("/:"+ComunicationConstants.MQTT_USER);
            connOpts.setPassword(ComunicationConstants.MQTT_PASSWORD.toChar
Array());
            client.connectWithResult(connOpts);
            return client;
        } catch (MqttException e) {
            e.printStackTrace();
        }

        return null;
    }
```

Script 16: Implementació del component MQTT

Aquesta primera part amb l'ajuda de les constants definides dintre de *commons* adquirim la *url* del broker MQTT, l'usuari i la contrasenya per a poder autenticar-se contra el servidor. A la definició del client *MqttClient* que connectarà amb el broker, a banda de l'*url,* aquest espera també un *id* amb el que identificarà a cada subscriptor i un objecte MemoryPersistence. Aquest objecte serà l'encarregat de persistir els missatges en memòria mentre es van enviant al tòpic. Una volta ja hagem configurat les propietats de connexió, establim la connexió amb el broker MQTT i tornem el client per a poder publicar el missatge.

```
@Override
public void publish(String clientId, String payload, String topicName, int
qos) {

    try {
       MqttClient client = this.connectMQTTtoRabbitMQ(clientId);

        if(client != null) {
           MqttTopic topic = client.getTopic(topicName);
           MqttMessage message = new MqttMessage(payload.getBytes());
           message.setQos(qos);
           MqttDeliveryToken token = topic.publish(message);
           token.waitForCompletion();

           //Tanca la connexió
           client.disconnect();
       }else {throw new Exception ("El client es null");}
   } catch (Exception mqe) {
       mqe.printStackTrace();
   }
}
```

Script 17: Implementació de publicació d'un missatge MQTT

Acabem de definir al paràgraf anterior com es configura un client per a poder connectar-se al broker MQTT, dit mètode s'executa perquè es crida dintre del mètode *publish* i retorna la connexió a la línia on es declara: *MqttClient client* = ... Ara per tant el que ens falta serà el contingut del missatge *payload*, el *topic*, i el *QoS* (Quality of service).

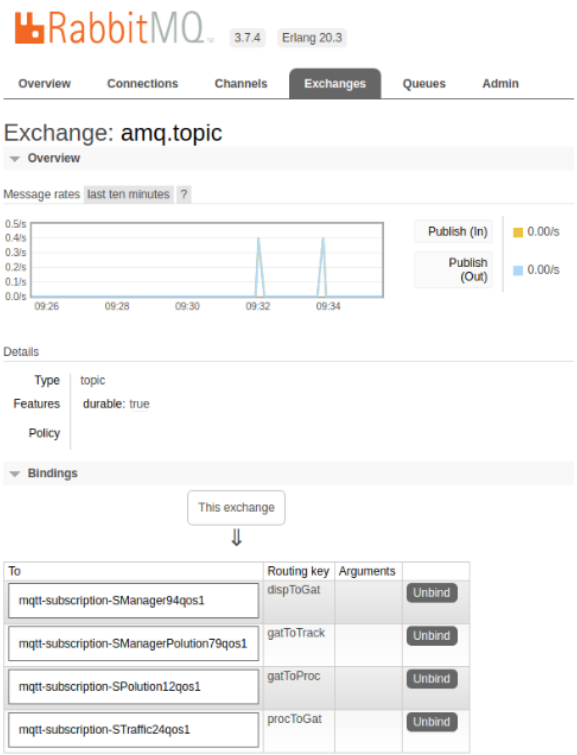
Existeixen 3 nivells de QoS, de menor a major garantissen la confiabilitat en la recepció del missatge entre client/broker.

- 0: El broker/client enviarà el missatge una volta, sense confirmació.
- 1: El broker/client enviarà el missatge almenys una volta, amb confirmació.
- 2: El broker/client enviarà el missatge exactament una volta utilitzant el un *handshake* de 4 pasos.

El plugin MQTT de RabbitMQ no accepta QoS de nivell 2 però per al nostre cas tampoc és necessari tindre un nivell tan alt.

Per últim *MqttDeliveryToken* crearà un *token* que publicarà el missatge amb format *json* dintre del tòpic i que bloquejarà el fil d'execució fins que no finalitze l'acció i tancarem la connexió amb el broker.

La recepció del missatge per part del microservei subscriptor es fa implementant la interfície *MqttCallback* i per tant implica que tots els microserveis tindran que implementar els mètodes definits a la interfície dintre de la capa lògica de cadascun. La definició d'aquesta part serà el primer punt dintre de la implementació del microservei *Gateway*.



Il·lustració 30: Subscripció de cues MQTT enllaçades a l'exchange amq.topic

Services: Cues de treball AMQP

L'Última funcionalitat que implementa aquesta llibreria és la publicació/consum de cues de treball. Com s'ha definit anteriorment, aquestes s'encarreguen d'encolar els missatges que apleguen dels sensors i també els missatges que es processen.

Com es pot apreciar a la següent imatge, la definició de la interfície que exposa els serveis que implementa, és molt senzilla, solament disposem d'un mètode que publica i d'altre que consumeix missatges de la cua. A continuació definirem els mètodes que configuren una connexió i també la publicació/subscripció sobre aquestes cues.

```
@Service

public interface AmqpService {

    public void sendMessage (SensorMessage message, String exchange, String
routingKey) throws IOException, TimeoutException;
    public SensorMessage getFromRabbitMQQueue(String queue);

}
```

Script 18: Implementació del servei AMQP

Ja coneguts els serveis exposats, es defineix a continuació la connexió del client i l'extracció del missatge. Cal dir que així com per al *plugin* d'MQTT sí que s'ha configurat un usuari per a publicador/consumidor per a AMQP s'ha deixat la configuració per defecte.

També s'ha dividit la connexió amb el broker AMQP per a facilitar la llegibilitat del codi, separant la creació del exchange i la creació de la cua de treball. Aquestos dos, els definim a continuació.

```
private void createQueue(String rt, String exchange, String routingKey)
throws IOException, TimeoutException {

    //Generem una nova connexió amb l'exchange que definim per a poder crear
la cua i posteriorment enviar el missatge
    this.createExchange(exchange);
    try {
       channel.queueDeclare(rt, true, false, false, null);
       channel.queueBind(rt, exchange, routingKey);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private void createExchange(String exchange) throws IOException,
TimeoutException {
    ConnectionFactory factory = new ConnectionFactory();
    // "guest"/"guest" by default, limited to localhost connections
    factory.setUsername(ComunicationConstants.RABBITMQ_BROKER_USER);
    factory.setPassword(ComunicationConstants.RABBITMQ_BROKER_PASSWORD);
    factory.setVirtualHost(ComunicationConstants.RABBITMQ_SLASH);
    factory.setHost(ComunicationConstants.RABBITMQ_HOST);
    factory.setPort(5672);

    conn = factory.newConnection();
    channel = conn.createChannel();
    channel.exchangeDeclare(exchange,
ComunicationConstants.RABBITMQ_DIRECT_EXCHANGE, true);

}
```

Script 19: Implementació de connexió amb AMQP

Els objectes que ens ajuden a crear la connexió *Connection* i *Channel* del paquet *com.rabbitmq.client* seran globals de la classe *AmqpServiceImpl*. Channel establirà el canal de connexió amb *l'exchange* i ens permetrà enrutar el missatges cap a la cua de treball corresponent. Tant a AMQP com a MQTT quan s'intenta crear una connexió ja existent no crea connexions noves si no que ho identifica i manté les que estan actives.

Definint un poc els dos mètodes tenim per una banda *createExchange* i per altra *createQueue*. El primer configurarà unes propietats definides a la classe *ComunicationConstants* que ens serviran per al igual que en MQTT, definir les propietats de connexió i en aquest cas definir una nova *exchange* "directa" amb la que enllaçarem les cues de treball mitjançant la *routingKey* que definim.

Definir l'*exchange* de forma "directa" significa que distribuirà la càrrega d'extracció de missatges entre tots els consumidors aplicant una política Round-Robin. Per tant si en el futur fem d'aquestos microserveis una arquitectura escalable, balancejarem la càrrega dels consumidors de missatges.

Cal recordar que les cues de treball es defineixen per el camp **rt**.

Ja disposem de la connexió amb el broker AMQP i per tant ja es pot publicar o consumir missatges de les cues de treball. A continuació es defineixen els dos mètodes encarregats de fer dites tasques.

```
@Override
public void sendMessage(SensorMessage message, String exchange, String
routingKey) throws IOException, TimeoutException {

   //Si existeix la cua la no la crearà de nou
   this.createQueue(message.getRt(), exchange, routingKey);
   byte[] messageBodyBytes = this.mapToString(message).getBytes();
   channel.basicPublish(exchange, routingKey,
           new AMQP.BasicProperties.Builder()
           .contentType("application/json")
           .build(),
           messageBodyBytes);

   //Tanquem la connexió després d'enviar el missatge
   conn.close();
}
@Override
public SensorMessage getFromRabbitMQQueue(String queue) {
   SensorMessage message = null;
   message = amqpTemplate.receiveAndConvert(queue,
             new ParameterizedTypeReference<SensorMessage>() {
             });
   return message;
}
```

Script 20: Publicació i extracció d'un missatge AMQP

Dintre del mètode *sendMessage* una volta ja tenim la connexió definida es continua amb la publicació del missatge, aquest el convertirem a bytes per a de seguida enviar-lo a la cua de treball amb el mètode *channel.basicPublish* on el definirem com a un missatge de tipus *json* i tancarem la connexió.

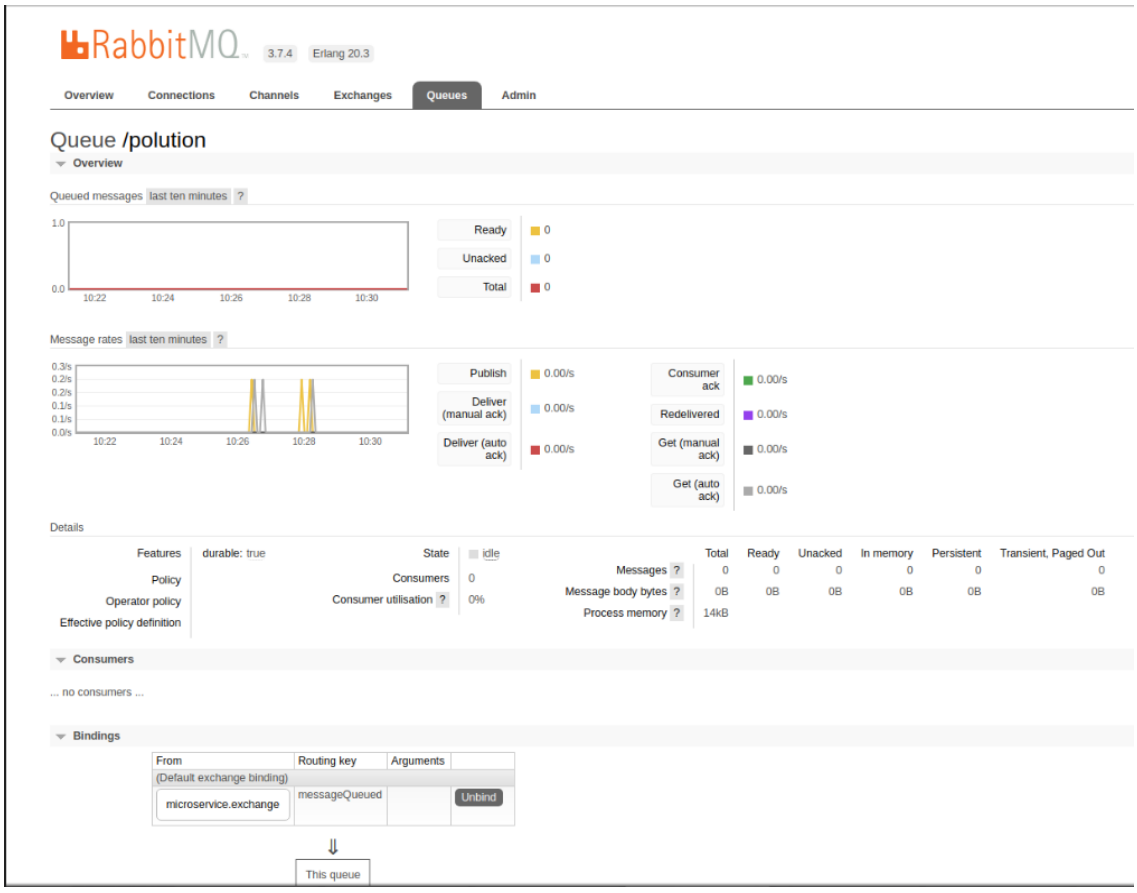
Per altra banda el microservei encarregat de consumir el missatge enviat anteriorment l'extraurà amb l'ajuda de la interfície *amqpTemplate*. Aquesta amb simplement la cua a la que s'ha enviat el missatge, extraurà i convertirà el missatge de format *json* a un objecte SensorMessaje.

Per a la transformació del missatge consumit és necessari declarar una classe *RabbitMQConfiguration* a la qual hem definit un *MessageConverter* que juntament amb la llibreria *jackson-core* convertiran l'objecte de *json* a SensorMessage.

```
@Configuration
public class AmqpConfiguration {

    @Bean
    public MessageConverter messageConverter() {
        Jackson2JsonMessageConverter converter = new
Jackson2JsonMessageConverter();
        converter.setCreateMessageIds(true);
        return converter;
    }

}
```

Script 21: @Configuration de AMQP

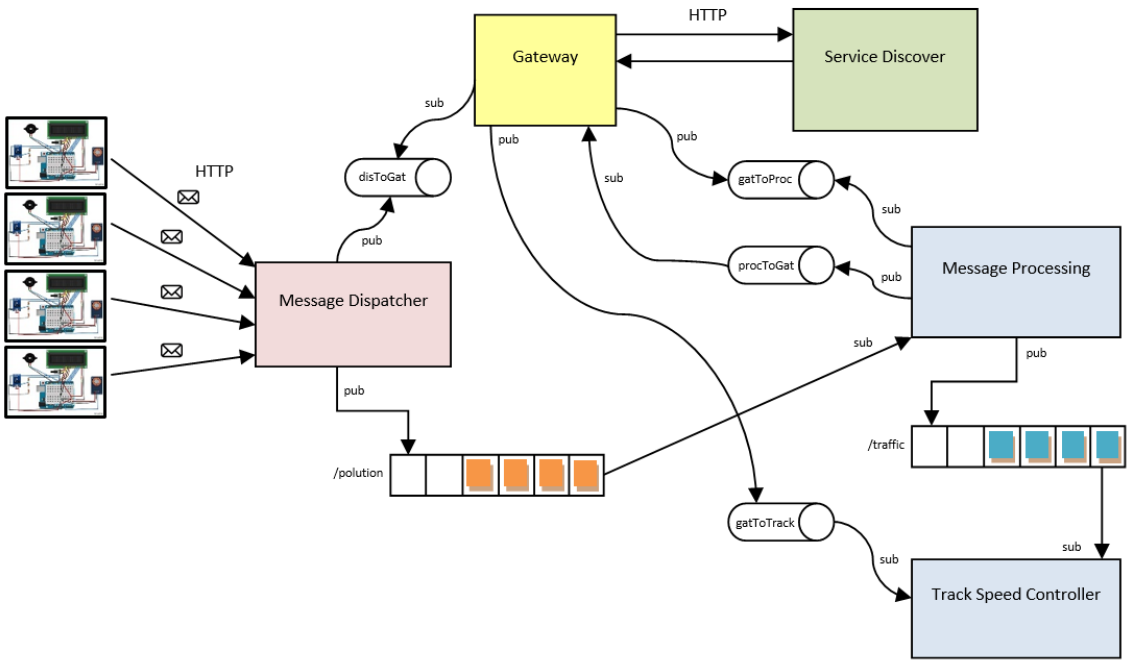
La implementació de publicació també s'hauria pogut fer amb *amqpTemplate* però açò ens limitaria molt la personalització de les cues, ja que implica definir l'exchange i la cua dintre del @Configuration que acabem de mostrar just a l'inici. De la forma que l'hem implementat el que aconseguim és fer-ho el més genèric possible podent crear múltiples *exchanges* vinculades a moltes *queues* amb *routingKeys* diferents.



Il·lustració 31: Exemple de la cua de treball de missatges rebuts de sensors /polution

Ja tenim la base, què és el següent?

Fins ací ja s'ha descrit com instal·lar i configurar el servidor RabbitMQ que servirà de broker de missatgeria tant MQTT com AMQP, també s'ha definit una peça molt important de l'arquitectura, la llibreria *commons*. Aquesta contindrà les constants comunes, els mètodes comuns de publicació/subscripció d'MQTT i publicació/extracció d'AMQP i per últim els mètodes bàsics d'auditoria per a persistir les comunicacions entre microserveis.



Il·lustració 32: Recordem el disseny de l'arquitectura

A continuació es defineixen els microserveis que composen aquesta arquitectura i on tots portaran inclosa la llibreria anteriorment descrita.

Per tant començarem amb el *Message Dispatcher* que s'encarregarà d'enviar el missatge "en brut" a la cua de treball corresponent al seu **rt** i quan acabe es comunicarà amb el *Gateway* que identificarà amb l'ajuda d'Eureka els microserveis disponibles.

Acte seguit el *Gateway* es comunica amb el microservei *Message Processing* per a extraure el missatge de la cua de treball, processar-lo, canviar-li l'**rt** i enviar-ho a la nova cua de treball. Com que *Message Processing* no coneix l'estat del seu entorn necessita de nou el *Gateway*.

Així doncs amb l'ajuda d'Eureka el *Gateway* es comunica amb l'últim microservei *Track Speed Controller* que canviarà la velocitat de la via en funció dels nivells de pol·lució i es comunicarà amb el sistemes de gestió de tràfic i de senyalització per a que coneixent l'estat actual decidesquen si canviar la velocitat o no.

6.5. Message Dispatcher

Passem doncs al primer microservei que va ser implementat, aquest com hem mencionat anteriorment s'encarrega d'encolar el missatge, comunicar-ho al *Gateway* i auditar les accions fetes.

Com que el missatge aplega del sensor, necessitem implementar un controlador REST que siga capaç de rebre una petició POST i transformar el missatge rebut en un *SensorMessage*.

```
@RestController
@RequestMapping("/api/queue")
public class QueueControllerImpl implements QueueController {

    @Autowired
    private RabbitMQManagerService rabbitMQManagerService;

    @Override
    @RequestMapping(value = "/sendQueue", method = RequestMethod.POST,
consumes = MediaType.APPLICATION_JSON_VALUE)
    public void sendQueue(@RequestBody SensorMessage message) {

        if (message != null) {
            try {
                rabbitMQManagerService.incomingMessage(message);

            } catch (IOException | TimeoutException e) {
                e.printStackTrace();
            }
        }
    }
```

Script 22: Controlador REST que rep els missatges dels sensors

L'anotació @RestController ens indicarà que aquesta classe és un controlador REST i juntament amb @RequestMapping definirà la ruta per a accedir al servei. També podem observar que s'ha injectat la interfície que exposa el @Service que implementa el tractament del missatge rebut, aquesta s'utilitza dintre del mètode *messageReceived,* el qual com podem observar també te l'anotació @RequestMapping per a completar la ruta del servei junt al mètode REST que espera i el tipus de missatge que consumeix, en aquest cas un *json*.

```
@Component
public class AmqpServiceManagerImpl implements RabbitMQManagerService{

    @Autowired
    private AuditService auditService;
    @Autowired
    private AmqpService amqpService;
    @Autowired
    private MqttService mqttService;

    @Override
    public void incomingMessage (SensorMessage message) throws IOException,
TimeoutException {

        // Objecte rebut desde microservei físic s'envía a la cua de treball
        amqpService.sendMessage(message,
ComunicationConstants.RABBITMQ_EXCHANGE,
ComunicationConstants.RABBITMQ_ROUTING_KEY);

        // Preparem el missatge a enviar per MQTT
        MqttTopicMessage mqttTopicMessage = new MqttTopicMessage();
        mqttTopicMessage.setRt(message.getRt());
        mqttTopicMessage.setTopic(ComunicationConstants.MQTT_MANAGER_TOPIC);

        mqttService.publish("RabbitMQ" + (int)(Math.random()*100),
mqttService.ConvertToString(mqttTopicMessage),
ComunicationConstants.MQTT_MANAGER_TOPIC, 1);

        // Auditoria: persistim la comunicació.
        auditService.saveMqttMessage(mqttTopicMessage);
        auditService.saveSensorMessage(message);
    }

}
```

Script 23: Lògica de reenviament de missatges Message Dispatcher

A l'script anterior podem observar el @Component encarregat de implementar la funcionalitat del @Service cridat al @RestController, aquest necessita que s'injecten els tres serveis continguts dintre de la llibreria *commons*. Amb l'ajuda d'aquestos tres serveis *amqpService.sendMessage()* enviarà el missatge del sensor a la cua de treball, *mqttService.publish()* publicarà un missatge al tòpic corresponent per a comunicar al *Gateway* que ha rebut un missatge d'un sensor i per últim persistirà tots dos missatges amb l'hora en que s'han creat els registres a la BBDD durant la seua inserció amb els serveis *auditService.saveMqttMessage()* i *auditService.saveSensorMessage()*.

6.6. Gateway

Continuem amb el microservei *Gateway*, encarregat de conèixer els microserveis registrats i en funció del **rt** detectar el tipus de microservei necessita per a després comunicar-se amb ell i dir-li a quina cua de treball s'ha de connectar.

Com comentava al punt en que descrivíem la implementació de les comunicacions MQTT, començarem doncs definint la interfície MqttCallback de la llibreria *Paho* [65], la qual és l'encarregada de subscriure's i escoltar el tòpic fins que arriba un missatge.

```
@PostConstruct
private void subscribe() {

   this.subscribeToTopic("SGateway"+random.nextInt(100-
0+1),ComunicationConstants.MQTT_MANAGER_TOPIC);
   this.subscribeToTopic("SGatewayProcessing"+random.nextInt(100-
0+1),ComunicationConstants.MQTT_POLUTION_TO_MANAGER_TOPIC);
}
```

Script 24: Mètode @PostConstruct per a la subscripció de topics MQTT

El mètode *subscribe()* està anotat per @PostConstruct la qual està proporcionada per la llibreria *javax* i s'encarrega d'ordenar la execució d'aquest mètode després del desplegament. Aquest mètode es subscriurà a dos *topics* un per a rebre el missatge MQTT del MessageDispatcher i altre per a rebre'l del microservei que haurà processat el missatge.

```
public void subscribeToTopic(String clientId, String topic) {
   MqttClient client;
    try {
       client = new MqttClient(ComunicationConstants.MQTT_BROKER_URL,
clientId);
       MqttConnectOptions connOpts = new MqttConnectOptions();
       connOpts.setCleanSession(true);
       connOpts.setUserName("/:"+ComunicationConstants.MQTT_USER);
       connOpts.setPassword(ComunicationConstants.MQTT_PASSWORD.toCharArray
());
       client.connect(connOpts);

       client.setCallback(this);
       client.subscribe(topic);
   } catch (MqttException e) {
       e.printStackTrace();
   }
}
@Override
public void messageArrived(String topic, MqttMessage message){
   try{
       Gson gson = new Gson();
       MqttTopicMessage mqttMessage = gson.fromJson(message.toString(),
               MqttTopicMessage.class);
       this.sendToMicroservice(mqttMessage);
   } catch(Exception e){
       e.printStackTrace();
   }
}
```

Script 25: Subscripció i tractament dels missatge MQTT

El mètode encarregat de la subscripció als *topics* és *subscribeToTopic()* que juntament amb les constants de comunicacions definides crea un client que escoltarà al topic que li diguem. En el moment aplegue algun missatge és el mètode *messageArrived()* l'encarregat de transformar-lo de *json* a objecte i després de cridar al mètode encarregat d'implementar la lògica.

Per a conèixer els microserveis registrats s'ha implementat el següent mètode que junt amb la injecció de la interfície *EurekaClient* crea una petició HTTP que ens torna un llistat de les aplicacions registrades.

```
@Override

public List<Application> getEurekaApps() {

    Applications apps = eurekaClient.getApplications();
    List<Application> appList = apps.getRegisteredApplications();
    return appList;
}
```

Script 26: Métode que torna els microserveis enregistrats a Eureka

Tenint ja l'últim element que ens faltava ja podem decidir cap a quin servei li anem a enviar un missatge MQTT amb la informació necessària per a desenvolupar la seua funció.

```
private void sendToMicroservice (MqttTopicMessage mqttTopicMessage) {
   List<Application> listApps = getEurekaApps();
   for (Application app : listApps) {
      InstanceInfo instanceInfo = app.getInstances().get(0);
      Map<String, String> metaData = instanceInfo.getMetadata();
      // Comprovar si la cua conté la substring del tipus del MS.
      // Comprovar el tipus del microservei rebut
      if
(mqttTopicMessage.getRt().toLowerCase().contains(CommonConstants.POLUTION)
&&
CommonConstants.MESSAGE_PROCESSING_TYPE.equals(metaData.get(CommonConstants
.TYPE))) {
            mqttTopicMessage.setTopic(ComunicationConstants.MQTT_MANAGER_TO_
POLUTION_TOPIC);

            mqttService.publish("ManagerPolution"+random.nextInt(100-0+1),
mqttService.ConvertToString(mqttTopicMessage),
ComunicationConstants.MQTT_MANAGER_TO_POLUTION_TOPIC, 1);

            auditService.saveMqttMessage(mqttTopicMessage);
            return;
      }else if
(mqttTopicMessage.getRt().toLowerCase().contains(CommonConstants.TRAFFIC_SP
EED) &&
CommonConstants.TRAFFIC_SPEED_MANAGER.equals(metaData.get(CommonConstants.T
YPE))) {
            mqttTopicMessage.setTopic(ComunicationConstants.MQTT_MANAGER_TO_
TRAFFIC_TOPIC);
            mqttService.publish("ManagerTraffic"+random.nextInt(100-0+1),
mqttService.ConvertToString(mqttTopicMessage),
ComunicationConstants.MQTT_MANAGER_TO_TRAFFIC_TOPIC, 1);

            auditService.saveMqttMessage(mqttTopicMessage);
            return;
      }
   }
}
```

Script 27: Lògica per al tractament de missatges rebuts al Gateway

Per a decidir cap a quin microservei es va a comunicar analitzarem el camp **rt** que ens dirà si és un missatge d'un sensor (*polution*) o si és un missatge ja processat (*traffic*), recordem que aquest valor es canvia una volta es processa. L'altra part de la comprovació dels *if* és amb una metadada que definim al fitxer *application.properties* de cada projecte i que es defineixen en la descripció del microservei *Eureka*. Una volta tots els missatges han sigut enviats, es persisteixen les comunicacions.

6.7. Eureka Server

El descobridor de serveis Eureka pren consciencia de l'estat dels microserveis de l'arquitectura per aquestos es comuniquen amb ell en el moment en que s'executen. El primer que es va a definir són les peces per a que els clients puguen contactar i després el necessari per a configurar el servidor [57].

Client

Per a poder registrar-se serà necessari que dintre de la classe principal de cada microservei estiga l'anotació @EnableDiscoveryClient i el controlador *ServiceDiscoverController*.

```
@RestController
public class ServiceDiscoverController {

    @Autowired
    private DiscoveryClient discoveryClient;

    @RequestMapping("/service-instances/{applicationName}")
    public List<ServiceInstance>
serviceInstancesByApplicationName(@PathVariable String applicationName){
        return discoveryClient.getInstances(applicationName);
    }
}
```

Script 28: Controlador REST dels clients Eureka

Per a poder configurar un client deurem tindre la llibreria del client Eureka de Netflix anomenada *spring-cloud-starter-netflix-eureka-client*.

Per últim, cada microservei necessita que definim unes propietats bàsiques per a la configuració del client i a més per necessitats de l'arquitectura es definiran tres propietats que seran metadades amb informació extra de l'aplicació.

```
#Eureka client config
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

eureka.instance.statusPageUrlPath = /actuator/info
eureka.instance.healthCheckUrlPath = /actuator/health

eureka.instance.metadataMap.appOwner = ecoMobility
eureka.instance.metadataMap.description = Traffic Lights Controller Manager
eureka.instance.metadataMap.type = TrafficLightManager
```

Script 29: Propietats bàsiques dels clients Eureka a application.properties

La primera propietat defineix la zona en la que treballa el servidor d'Eureka. En el cas que hi hagueren més zones aquestes són totalment transparents entre elles per el si en tinguérem de diferents deuríem d'agrupar be els microserveis. Les dos següents propietats es defineixen perquè és obligatori que cada microservei publique el seu estat, per a dur aquesta tasca s'ha utilitzat la llibreria *spring-boot-starter-actuator* implementant la seua configuració per defecte. Per últim les tres propietats restants serveixen com a metadades per a implementar la lògica que orquestra el *Gateway*.

Una volta definit què necessiten els clients per a contactar amb el servidor, continuem exposant la configuració mínima d'aquest últim.

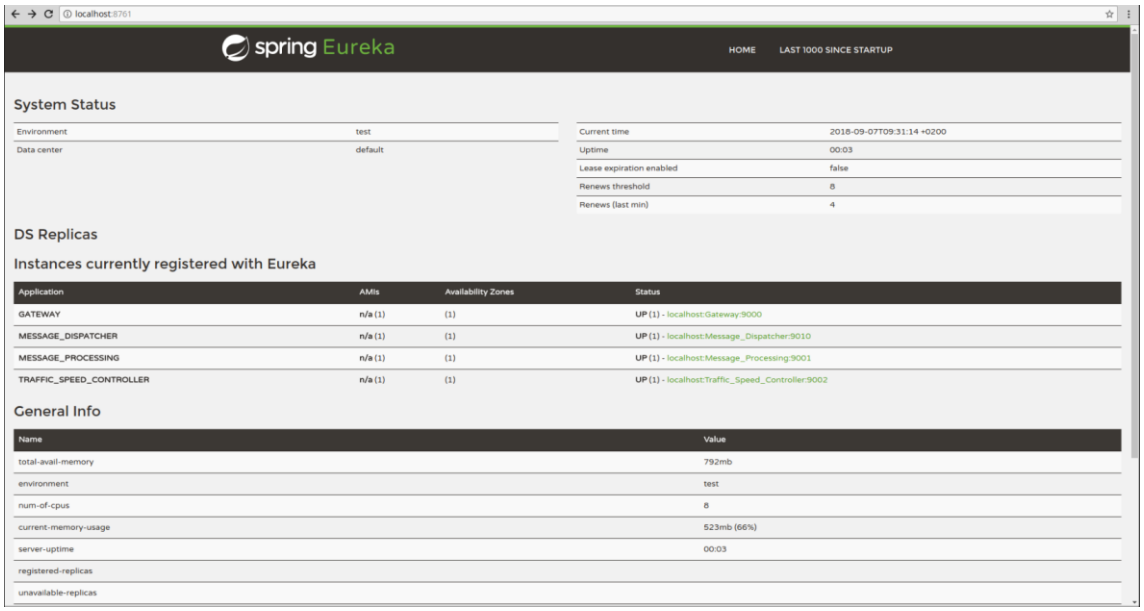
Servidor

Aplicar una configuració mínima és molt senzill, solament tindrem que tindre al nostre *pom.xml* la llibreria *spring-cloud-starter-netflix-eureka*-server, una classe *main* amb les anotacions @EnableEurekaServer i @SpringBootApplication que configuraran el context i definir unes poques propietats bàsiques.

```
server.port=8761

eureka.instance.hostname = localhost
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.serviceUrl.defaultZone =
http://${eureka.instance.hostname}:${server.port}/eureka/
eureka.client.healthcheck.enabled = true
```

Script 30: Propietats bàsiques del servidor Eureka

Dintre d'aquestes propietats tenim el port del servidor on escoltarà les peticions rebudes, dues propietats que es configuren per a que el propi servidor no es registre dintre del descobridor de serveis i que són *register-with-eureka* i *fetch-registry* i per últim la zona per defecte on es registraran els microserveis i un valor booleà que ens permetrà comprovar l'estat de salut de cadascun.



Il·lustració 33: Exemple de la interfície web Eureka a l'arquitectura ecoMobility

6.8. Message Processing

Una volta el *Gateway* ja s'ha comunicat amb el *Message Processing*, aquest s'encarrega de recuperar el missatge de la cua de treball per a processar-lo i comunicar-se amb el *Gateway* de nou per a fer-li saber que necessita a *Traffic Speed Controller* per a poder desenvolupar la seua funció.

Al igual que el microservei anterior, aquest també implementarà la interfície *MqttCallback* per a definir els mètodes de subscripció al *topic* corresponent. Com aquestos son pràcticament iguals al anterior, continuem la implementació de la lògica.

```
@Override

public void askToManager(MqttTopicMessage mqttMessage) {
   SensorMessage sensorMessage =
rabbitMQService.getFromRabbitMQQueue(mqttMessage.getRt());

   if(sensorMessage != null &&
sensorMessage.getRt().toLowerCase().contains(CommonConstants.POLUTION)) {
       try {
           Locale locale  = new Locale("en", "UK");
           Double aux;
           DecimalFormat decimalFormat = (DecimalFormat)
                   NumberFormat.getNumberInstance(locale);
           decimalFormat.setRoundingMode(RoundingMode.UP);
           decimalFormat.applyPattern("###,###.00");
           aux =
Double.parseDouble(decimalFormat.format(sensorMessage.getMeasure()));
           sensorMessage.setMeasure(aux);

           // Missatge processat, canviem el RT per a que la nova cua de
treball siga diferent
           sensorMessage.setRt(CommonConstants.TRAFFIC_SPEED_WORK_QUEUE);
           rabbitMQService.sendMessage(sensorMessage,ComunicationConstants.
RABBITMQ_EXCHANGE , ComunicationConstants.POLUTION_ROUTING_KEY);

           // Preparem el missatge a enviar per MQTT
           MqttTopicMessage mqttTopicMessage = new MqttTopicMessage();
           mqttTopicMessage.setRt(sensorMessage.getRt());
           mqttTopicMessage.setTopic(ComunicationConstants.MQTT_POLUTION_TO
_MANAGER_TOPIC);

           mqttService.publish("Polution+" + random.nextInt(100-0+1),
mqttService.ConvertToString(mqttTopicMessage),
ComunicationConstants.MQTT_POLUTION_TO_MANAGER_TOPIC, 1);

           // Auditoría
           auditService.saveSensorMessage(sensorMessage);
           auditService.saveMqttMessage(mqttTopicMessage);

       } catch (IOException | TimeoutException e) {
           e.printStackTrace();
       }
   }
}
```

Script 31: Lògica per al tractament dels missatges dels sensors

Dintre d'aquest mètode, processarem el missatge i li canviem el **rt** sabem que ve de la cua */polution* i per tant és un missatge d'un sensor de pol·lució. També sabem que el microservei al que ha de redirigir-se i modifiquem el **rt** per a que al tornar al *Gateway* la nova cua de treball on envie el missatge serà */trafficSpeed*.

En el futur es pot jugar amb aquest camp **rt.** Per exemple, si es desenvolupen més serveis, el camp pot passar a dir-se */polution/trafficSpeed* i així saber que primer haurà de processar el missatge i després enviar-lo al microservei *Traffic Speed Controller*.

6.9. Traffic Speed Controller

Últim servei de l'arquitectura i l'encarregat de prendre la decisió de si es canvia la velocitat o no. Aquest microservei al igual que els dos anteriors, implementa la interfície *MqttCallback* per a definir els mètodes de subscripció al *topic* corresponent. Com aquestos son pràcticament iguals al anterior, continuem la implementació de la lògica.

```
private void manageSpeed (MqttTopicMessage mqttTopicMessage) {
   SensorMessage sensorMessage =
rabbitMQService.getFromRabbitMQQueue(mqttTopicMessage.getRt());

   Road road = new Road();

   if(sensorMessage.getMeasure() >= CommonConstants.MINIMUM_POLUTION) {
       road.setVelocitat(CommonConstants.SET_SPEED_30);
       }else {
           road.setVelocitat(CommonConstants.SET_SPEED_50);
       }
   road.setLocation(sensorMessage.getLocation());
   road.setAction(CommonConstants.SET_SPEED_LIMIT);
   // Enviem resultat al monòlit
   this.putToMonolith("", HttpMethod.PUT, road);
   // Auditoría, deuría d'anar dintre de putToMonolith.
   auditService.saveSensorMessage(sensorMessage);
   this.saveRoad(road);
}
```

Script 32: Lògica per a la modificació de la velocitat de les vies

Aquest microservei al no conèixer la velocitat actual del tram de carretera a modificar simplement notifica que els nivells són alts o baixos i demana un canvi en funció a això. Aquesta notificació la farà mitjançant HTTP amb el mètode següent.

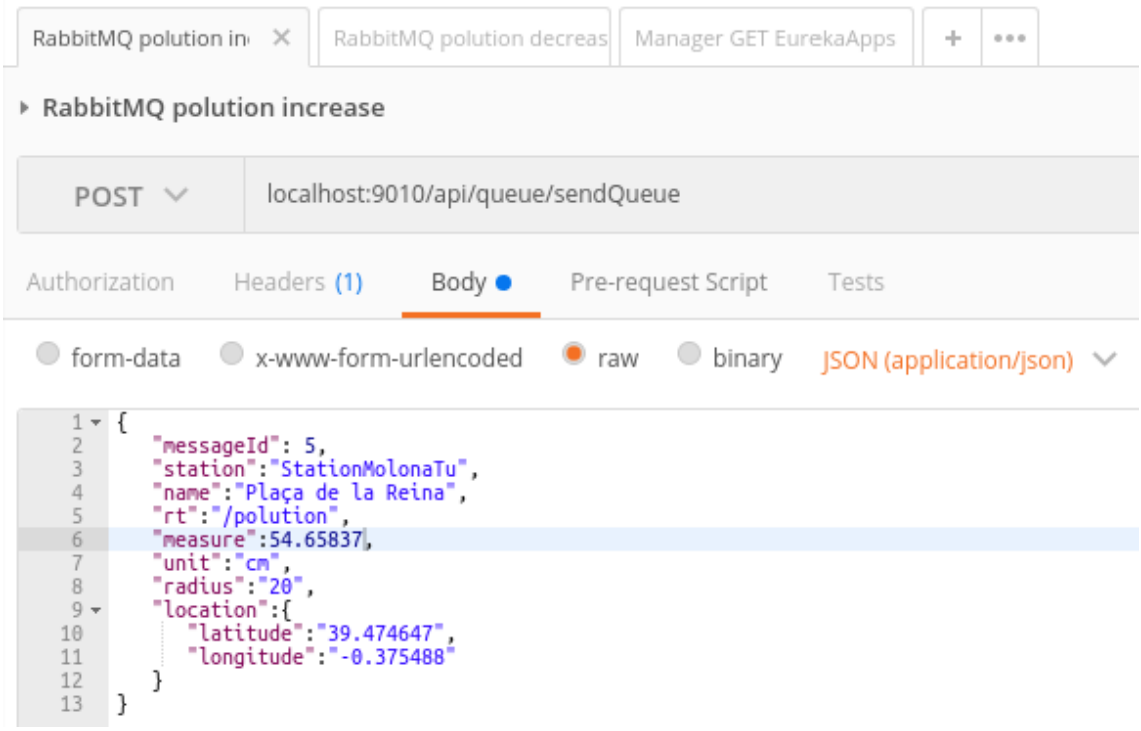
```
private void putToMonolith(String url, HttpMethod httpMethod, Road road) {

   HttpHeaders headers = new HttpHeaders();
   headers.setContentType(MediaType.APPLICATION_JSON);

   HttpEntity entity = new HttpEntity(road, headers);

   RestTemplate restTemplate = new RestTemplate();
   ResponseEntity<Road> sendRequest = restTemplate.exchange(url,
httpMethod, entity, Road.class);
}
```

Script 33: Enviament de la modificació de la velocitat al monòlit

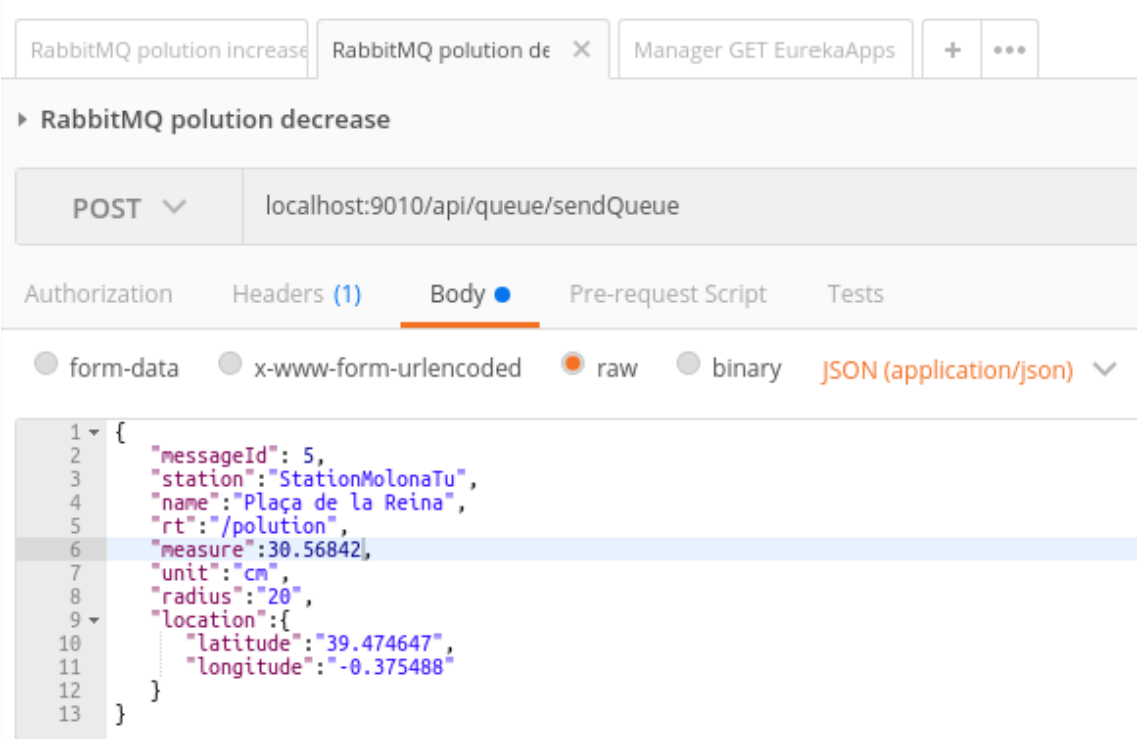
6.10. Mockeig de missatges de sensors.

Per a poder provar tota l'arquitectura s'ha utilitzat el software lliure Postman, aquest software és molt senzill i permet realitzar peticions HTTP de qualsevol tipus.

A continuació mostrarem dos captures en les que els nivells de pol·lució es superen i altre en que els nivells de pol·lució es redueixen:



Il·lustració 34: Mock que detecta nivells alts de pol·lució



Il·lustració 35: Mock que detecta nivells baixos de pol·lució

7. Conclusió i treballs futurs

Aquest treball ha suposat un canvi gran en la manera de pensar que tenia per a desenvolupar aplicacions. Conforme llegia documentació més m'interessava, sé que no és quelcom que qualsevol empresa es puga adaptar i que requereix d'un canvi en la filosofia i manera de desenvolupar aplicacions però pense que el fet que tinga tant part de desenvolupament com part d'infraestructura és el que més m'agrada.

És una arquitectura que encara continua en evolució però que ja ha madurat prou i existeixen empreses que ja desenvolupen productes per a que altres puguen migrar les seues aplicacions o desenvolupar-ne de noves. Netflix fou la pionera, després altres com Amazon, Mulesoft [59], WSO2 [60], NGINX i Kong [61] han continuat aquest camí per a desenvolupar aplicacions escalables, elàstiques i confiables.

A banda de l'èxit de Netflix i Amazon, l'adopció d'aquesta arquitectura per altres empreses com la pròpia Netflix amb la seua plataforma de series i pel·lícules baix demanda, Spotify, IBM, The New York Times, Soundcloud o Groupon [61, 62 i 63], ha fet que poc a poc vaja augmentant el seu interés i és el camí a seguir durant els pròxims anys.

Quasi totes les empreses nombrades anteriorment són empreses privades i que obtenen beneficis venent les seues plataformes, frameworks i proxies per a poder crear una arquitectura de microserveis però és important comentar, com ha quedat demostrat, que avui dia amb les ferramentes de software lliure existents el canvi cap a aquestes noves tecnologies és més fàcil.

També cal dir, encara que no haja sigut el tema cental del treball, el interès en augment sobre les *Smart Cities* i la preocupació creixent des de fa temps sobre el canvi climàtic. En l'actualitat moltes ciutats estan lluitant per aconseguir millorar la vida i la salut de les persones que les habiten, entre elles, podem trobar ciutats com Nova York, Londres, París, Barcelona o Bogotá, tractant de gestionar de millor forma el transport públic i privat, i l'ús eficient de recursos per a fer les ciutats el més eficient i ecològiques possible [64].

En base als objectius definits es pot concloure que:

- S'ha dissenyat una proposta base per a definir una arquitectura de microserveis aplicable al monòlit d'ecoMobility. Deixant així el camí per on continuar el seu desenvolupament e implementar totes les funcionalitats que conté.
- He aprés els coneixements necessaris sobre les arquitectures de microserveis i les seues tecnologies, moltes de elles ni les coneixia. Fruit de l'aprenentatge al llarg del treball s'ha desenvolupat aquesta arquitectura base.
- Els microserveis s'han desenvolupat acorde als requisits de les arquitectures de microserveis, com per exemple, la importància de les comunicacions o la creació pròpia de mòduls o llibreries que continguen funcionalitats compartides per a aconseguir una millor mantenibilitat de l'aplicació.
- S'ha implementat un nou cas d'ús que modifica el límits de velocitat de les vies en funció dels nivells de pol·lució i que formarà part de la resta de funcionalitats d'ecoMobility quan es migre totalment el monòlit.

Durant el desenvolupament d'aquest treball se m'han anat ocorrent possibles millores que es podrien implementar en un futur per a millorar l'arquitectura i que podrien utilitzar-se per a proposar treballs futurs:

- Desenvolupament d'una infraestructura de contenidors escalable aplicable a l'arquitectura de microserveis d'ecoMobility.
- Implementació de funcionalitats existents dintre del monòlit, com per exemple, la gestió del tràfic mitjançant la gestió dels semàfors.
- Afegir un mòdul d'autenticació per a les comunicacions HTTP i xifrar les comunicacions de missatgeria entre microserveis.
- Implementar la gestió del enllumenat públic afegint a la xarxa de sensors un mòdul de sensor de llum de tal forma que l'automatització de l'enllumenament siga més eficient.
- Implementació d'un microservei que actue com a repositori central on emmagatzemar tota la informació que produeixen els microserveis per a poder explotar aquesta informació en un futur i poder realitzar estudis per a millorar la qualitat de vida dels habitants de les ciutats.
- Desenvolupar un microservei que implemente una interfície web que mostre tota la informació rellevant sobre la velocitat de les vies de la ciutat. Aquest podria utilitzar l'auditoria pròpia de cada microservei.

8. Bibliografia

Organització Mundial de la Salut obtingut de:

- [1] www.who.int/phe/health_topics/outdoorair/databases/background_information/es/
- [2] http://www.who.int/es/news-room/fact-sheets/detail/ambient-(outdoor)-air-quality-and-health
- Cas d'us guia de qualitat de l'aire:
    - [50] http://apps.who.int/iris/bitstream/handle/10665/69478/WHO_SDE_PHE_OEH_06.02_spa.pdf?sequence=1

Restricció del tràfic a Madrid obtingut de:

- [4] https://elpais.com/elpais/2018/07/10/opinion/1531234272_494970.html
- [5] http://www.elmundo.es/madrid/2018/05/10/5af44b4b22601d98468b45e3.html

Protocol de contaminació de Madrid obtingut de:

- [6] https://www.madrid.es/portales/munimadrid/es/Inicio/Actualidad/Noticias/Nuevo-protocolo-de-contaminacion-en-funcion-de-los-distintivos-ambientales-de-la-DGT/?vgnextfmt=default&vgnextoid=3f0e72fbcc943610VgnVCM2000001f4a900aRCRD&vgnextchannel=a12149fa40ec9410VgnVCM100000171f5a0aRCRD

Planificació:

- [7] https://www.obs-edu.com/es/blog-project-management/metodologia-agile/pros-y-contras-de-la-metodologia-en-cascada
- [8] https://www.ganttproject.biz/
- [9] https://es.atlassian.com/devops

Context tecnològic obtingut de:

- [10] https://blog.leanix.net/en/why-netflix-amazon-and-apple-care-about-microservices
- [11] https://eng.uber.com/building-tincup/
- [15] https://netflix.github.io/
- [13] https://www.nginx.com/resources/library/app-dev-survey/
- https://martinfowler.com/microservices/
- [14] https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/
- [16] https://www.computerworld.com.au/article/621169/how-determine-when-why-use-microservices/

Tecnologies:

JSON:

- [17] https://www.json.org/

XML:

- [18] https://www.w3.org/XML/
- [19] https://www.w3.org/TR/2008/REC-xml-20081126/#sec-origin-goals

HTTP:

- [20] https://www.w3.org/Protocols/rfc2616/rfc2616.html

STOMP:

- [21] https://stomp.github.io/
- [22] https://stomp.github.io/stomp-specification-1.2.html#Abstract

AMQP:

- [23] https://www.amqp.org/about/what
- [24] http://www.amqp.org/specification/0-9-1/amqp-org-download

MQTT:

- [25] http://mqtt.org/
- [26] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

Mosquitto:

- [27] https://mosquitto.org/
- [28] https://www.eclipse.org/proposals/technology.mosquitto/
- [29] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," ***The Journal of Open Source Software***, vol. 2, no. 13, May 2017

RabbitMQ:

- [30] https://www.rabbitmq.com
- [31] https://www.rabbitmq.com/plugins.html
- [32] https://www.rabbitmq.com/devtools.html

STRUTS:

- [33] https://struts.apache.org

JAX-WS:

- [34] https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jax-ws/index.html
- [35] https://www.ibm.com/support/knowledgecenter/es/SSEQTP_9.0.0/com.ibm.websphere.base.doc/ae/cwbs_jaxws.html

SPRING:

- [36] https://spring.io/
- [37] https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html

Hibernate:

- [38] http://docs.jboss.org/hibernate/orm/5.3/quickstart/html_single/#preface

Spring jdbc:

- [39] https://projects.spring.io/spring-data-jdbc/

ActiveJDBC:

- [40] http://javalite.io/documentation

Eureka:

- [41] https://github.com/Netflix/eureka/wiki
- [42] https://spring.io/guides/gs/service-registration-and-discovery/

Consul:

- [43] https://www.consul.io/intro/index.html

Anàlisi del problema i Cas d'estudi: ecoMobility, obtingut de:

- [12] Sam, Newman 2015: Building microservices
- [44] https://es.wikipedia.org/wiki/Programaci%C3%B3n_por_capas
- [45] https://martinfowler.com/articles/microservices.html
- [46] https://www.elastic.co/blog/scaling-the-elastic-stack-in-a-microservices-architecture-rightmove
- [47] https://konghq.com/plugins/
- [48] https://martinfowler.com/articles/break-monolith-into-microservices.html
- [49] https://martinfowler.com/bliki/MonolithFirst.html
- https://www.mulesoft.com/resources/api/microservices-vs-monolithic
- [3] Projecte ecoMobility: mobilitat sostenible en ciutats intel·ligents, Joan Fons. Càtedra Telefònica UPV, Ajudes per al desenvolupament de demostradors tecnològics 2017. Unviversitat Politècnica de València

Disseny e implementació de l'aplicació:

Instal·lar i configurar RabbitMQ:

- Macero Moises 2017: Learn Microservices with Spring Boot
- Requisits mínims:
    - [51] https://rabbitmq.docs.pivotal.io/37/topics/about-suppconfigsrabbit.html
    - [52] https://www.rabbitmq.com/platforms.html

    - [53] https://www.rabbitmq.com/which-erlang.html
- Instal·lació broker:
    - [54] https://www.rabbitmq.com/install-debian.html
- Plugin:
    - [55] https://www.rabbitmq.com/mqtt.html

Comportament de Eureka Netflix:

- [56] https://github.com/Netflix/eureka/wiki/Understanding-eureka-client-server-communication
- [57] https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka

Maven:

- [58] https://maven.apache.org/

Spring i Spring-boot:

- https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
- https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html
- https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html
- https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html
- https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html

JPA i Hibernate:

- https://spring.io/guides/gs/accessing-data-jpa/#initial
- http://hibernate.org/orm/documentation/5.3/
- https://docs.oracle.com/javase/8/docs/api/index.html

MQTT:

- https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/persist/package-summary.html
- https://mosquitto.org/man/mqtt-7.html
- [65] https://www.eclipse.org/paho/clients/java/#

AMQP:

- https://www.rabbitmq.com/tutorials/amqp-concepts.html
- https://www.rabbitmq.com/img/tutorials/intro/exchange-direct.png

Conclusió:

- [59] https://blogs.mulesoft.com/biz/news/success-in-microservices-with-mulesoft-spotify/
- [60] https://wso2.com/products/microservices-framework-for-java/

- [61] https://konghq.com/kong-community-edition/who/
- [62] https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith
- [63] https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/
- [64] https://www.bbva.com/es/las-smart-cities/