
ScorpioBroker Documentation

NECTI

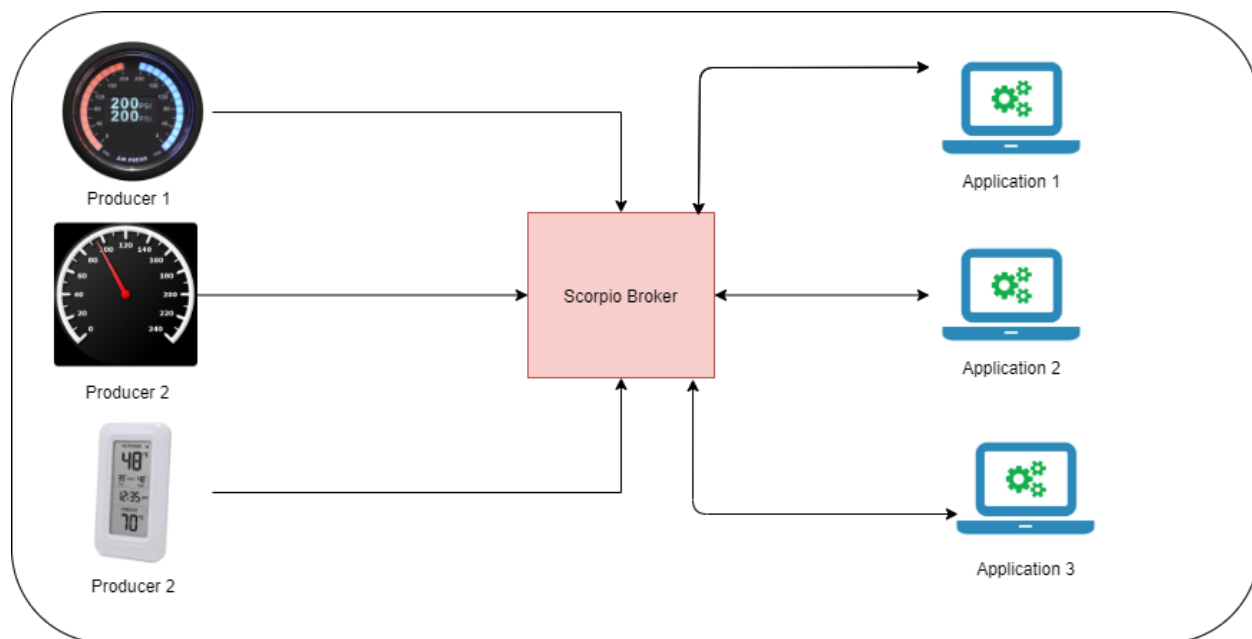
Sep 13, 2023

| | | |
|-----------|--|-----------|
| 1 | Introduction | 3 |
| 2 | Basic Guide | 5 |
| 3 | System Setup Guide | 7 |
| 4 | Starting Scorpio via docker-compose | 13 |
| 5 | Configuration Parameters | 15 |
| 6 | Building Scorpio from source | 17 |
| 7 | MQTT Notifications | 23 |
| 8 | Introduction | 27 |
| 9 | Entity creation | 31 |
| 10 | Querying & receiving entities | 35 |
| 11 | Updating an entity & appending to an entity | 43 |
| 12 | Subscriptions | 47 |
| 13 | Batch operations | 57 |
| 14 | Context Registry | 61 |
| 15 | Developer Installation Guide | 65 |
| 16 | System Requirements | 73 |
| 17 | Error Handling in Scorpio | 75 |
| 18 | Security in Scorpio | 77 |
| 19 | Hello World example | 79 |
| 20 | Multi-value Attribute | 81 |

| | |
|---|------------|
| 21 Architecture | 85 |
| 22 Deployment Architecture | 89 |
| 23 Operation flows | 91 |
| 24 Contribution guidelines | 103 |
| 25 Getting a docker container | 105 |
| 26 Config parameters for Scorpio | 107 |
| 27 Troubleshooting | 111 |
| 28 Deployment Guide for Scorpio Broker on Kubernetes | 113 |

The Scorpio Broker implements the NGSI-LD API through which context producers and consumers can interact with each other. For Example in the typical IoT based room, various sensors like temperature sensors, light sensors, etc are connected to the central application which uses those sensors output and acts as the consumer. There can be a lot of use cases for this central application i.e Scorpio.

1. Scorpio uses the NGSI-LD API and information model to model entities with their properties and relationships, thus forming a property graph with the entities as the nodes. It allows finding information by discovering entities, following relationships and filtering according to properties, relationships and related meta-information. For data not directly represented in NGSI-LD like video streams or 3D models, links can be added to the model that allows consumers to directly access this information. In this way, Scorpio can provide a graph-based index to a data lake.
2. Scorpio provides several interfaces for querying the stored data so easily analytics can be done on the stored data. like it can be used to predict the situation of an ecosystem. Example:- In a huge building there can be several fire sensors, temperature sensors, and smoke sensors. In case of a false fire alarm, it can be verified by the collected fire data, temperature data and smoke data of the particular area.
3. Scorpio can be used for determining the accuracy of any event. For example, In an automated car, the speed of the car can be known by several applications like GPS, speed camera and speedometer. Scorpio's internal data is stored in this way that any third-party application can use it and can find the accuracy and determine faulty device.



Scorpio broker is a reference implementation of **NGSI-LD standard** specifications that are compliant to **ETSI standards**. Basically Scorpio broker is a core component of **FiWARE/IoT** platform wherein IoT data-driven by dynamic context is collected, processed, notified & stored/ingested with different application usage perspectives. Scorpio broker also provides an implementation of REST API endpoints for various data context operations that conform to **NGSI-LD API** specification. Scorpio broker allows you to collect, process, notify and store the IoT data with dynamic context with the use of linked data concepts. It makes use of the **microservice-based architecture** build with the help of **spring boot**, which has its own advantages over the existing IoT brokers such as scalability, cross-technology integration, etc.

Scorpio Broker based on NGSI-LD offers a unique feature of Link data context that provides self-contained (or referenced) **dynamic schema definition** (i.e. the context) for contained data in each message/entity. Thus allows the Scorpio Broker core processing to still remain unified even it gets dynamic context-driven data as its input from different types of data sources coupled(or designed for) with different schemas.

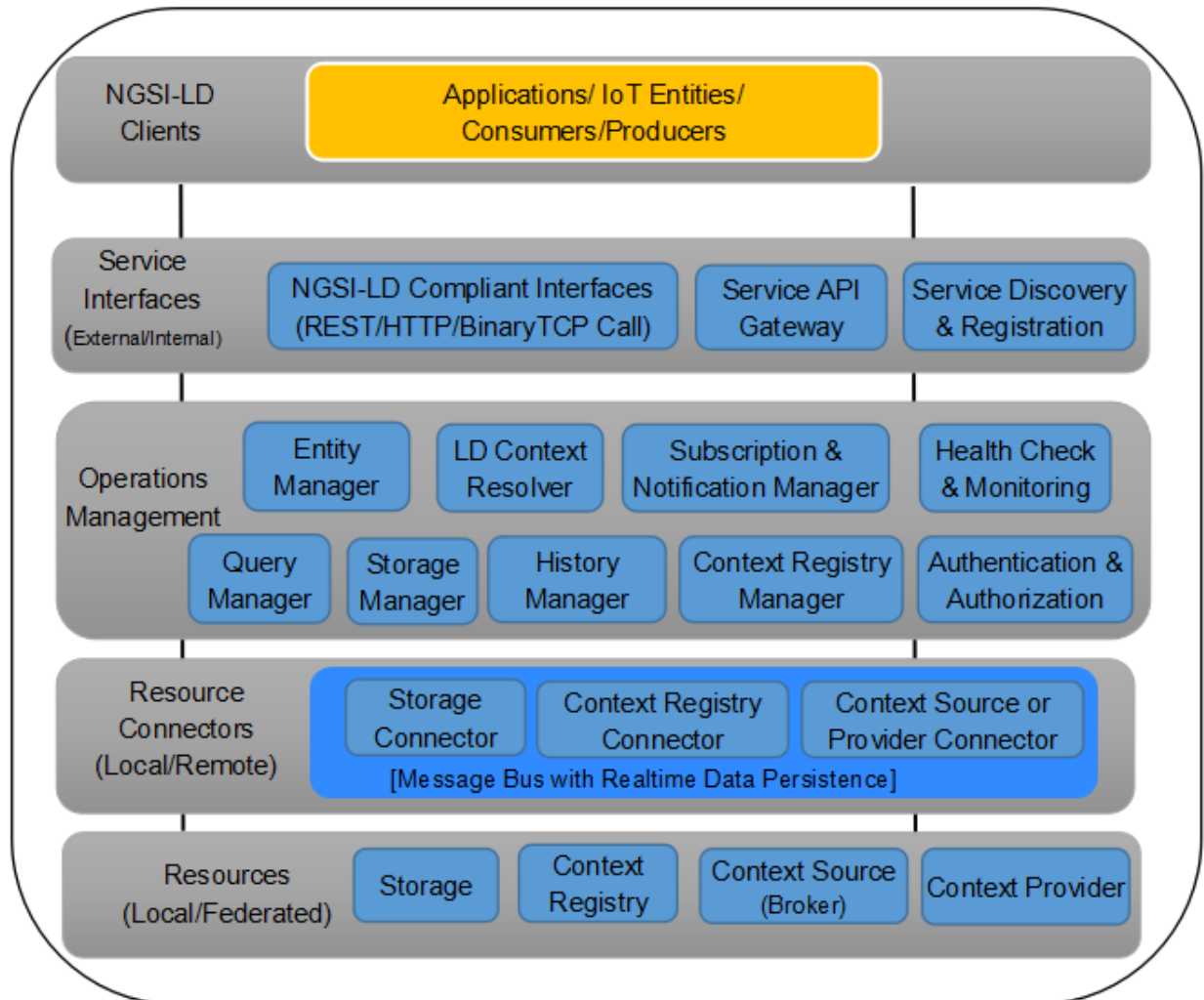
Key advantages of Scorpio Broker over other brokers:

- Uses micro-service architecture which enhances the performance drastically.
- The Scorpio Broker architecture is designed & implemented as a scalable, highly available, and load balanced.
- Use of Ld which gives us the leverage of dynamic context.
- Usage of Kafka, allowing us the robust pub-sub service with the facility of scaling with no downtime.
- It provides fail-over resiliency.
- It provides load balancing to distribute the load on distributed infrastructure.
- It is modular enough to offer low coupling and high cohesion by design.
- It offers different storage integration without changing the application logic time and again.

2.1 Architectural Overview

Scorpio Broker is a reference implementation of NGSI-LD APIs. Scorpio Broker provides an implementation of REST API endpoints for various data context operations that conform to NGSI-LD API specification. Scorpio Broker component has been implemented based on modular, Microservices oriented, scalable, secure by design, easy to monitor/debug, fault-tolerant, and highly available architecture. Scorpio Broker based on NGSI-LD offers a unique feature of Link data context that provides self-contained (or referenced) dynamic schema definition (i.e. the context) for contained data in each message/entity. Thus allows the Scorpio Broker core processing to still remain unified even it gets dynamic context-driven data as its input from different types of data sources coupled(or designed for) with different schemas.

The basic architecture of the Scorpio Broker consists of five layers, the first layer consists of the Scorpio Broker clients which act as the producers and consumers. The second layer act as an interface between the Scorpio Broker and the external world this layer comprises the NGSI-LD Compliant Interfaces, Service API Gateway, and Service Discovery & Registration. The third layer contains all the micro-services and is responsible for the majority of tasks like entity CRUD operations etc. The fourth layer acts as the interface which connects different micro-services from the storage. The fifth layer is a Resources layer which acts as the storage for Scorpio Broker.



In order to set-up the environment of Scorpio broker, the following dependency needs to be configured:-

1. Server JDK.
2. Apache Kafka.
3. PostgreSQL

3.1 Windows

3.1.1 JDK Setup

- Start the JDK installation and hit the “Change destination folder” checkbox, then click ‘Install.’

Note:- Recommended version is JDK-11. Scorpio Broker is developed and tested with this version only.

- Change the installation directory to any path without spaces in the folder name.

After you’ve installed Java in Windows, you must set the JAVA_HOME environment variable to point to the Java installation directory.

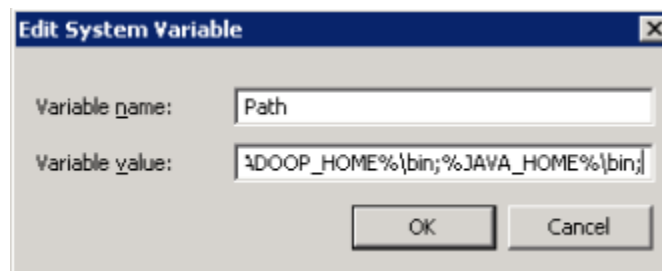
Set the JAVA_HOME Variable

To set the JAVA_HOME variable:

1. Find out where Java is installed. If you didn’t change the path during installation, it will be something like this:
C:\Program Files\Java\jdk1.version
2.
 - In Windows 7 right-click **My Computer** and select **Properties > Advanced**.
 - OR
 - In Windows 8 go to **Control Panel > System > Advanced System Settings**.
3. Click the Environment Variables button.
4. Under System Variables, click New.



5. In the User Variable Name field, enter: **JAVA_HOME**
6. In the User Variable Value field, enter your JDK path.
(Java path and version may change according to the version of Kafka you are using)
7. Now click OK.
8. Search for a Path variable in the “System Variable” section in the “Environment Variables” dialogue box you just opened.
9. Edit the path and type `;%JAVA_HOME%\bin` at the end of the text already written there, just like the image below:

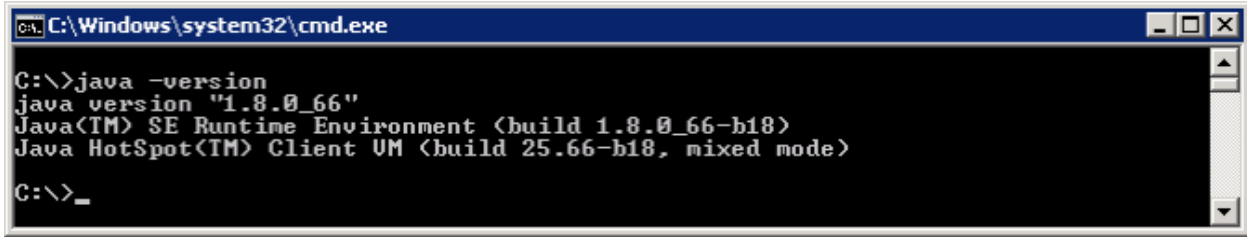


- To confirm the Java installation, just open cmd and type “java -version.” You should be able to see the version of Java you just installed.

If your command prompt somewhat looks like the image above, you are good to go. Otherwise, you need to recheck whether your setup version matches the correct OS architecture (x86, x64), or if the environment variables path is correct.

3.1.2 Setting Up Kafka

1. Go to your Kafka config directory. For example:- **C:kafka_2.12-2.1.0config**



```

C:\Windows\system32\cmd.exe

C:\>java -version
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b18)
Java HotSpot(TM) Client VM (build 25.66-b18, mixed mode)

C:\>_

```

2. Edit the file “server.properties.”
3. Find and edit the line `log.dirs=/tmp/kafka-logs` to `“log.dir= C:kafka_2.11-0.9.0.0kafka-logs.”`
4. If your ZooKeeper is running on some other machine or cluster you can edit `“zookeeper.connect:2181”` to your custom IP and port. For this demo, we are using the same machine so there’s no need to change. Also the Kafka port and broker.id are configurable in this file. Leave other settings as is.
5. Your Kafka will run on default port 9092 and connect to ZooKeeper’s default port, 2181.

Note: For running Kafka, zookeepers should run first. At the time of closing Kafka, zookeeper should be closed first than Kafka. Recommended version of kafka is `kafka_2.12-2.1.0`.

3.1.3 Running a Kafka Server

Important: Please ensure that your ZooKeeper instance is up and running before starting a Kafka server.

1. Go to your Kafka installation directory: `** C:kafka_2.11-0.9.0.0**`
2. Open a command prompt here by pressing Shift + right-click and choose the “Open command window here” option).
3. Now type `.bin\windowskafka-server-start.bat .config\server.properties` and press Enter, then
4. Type `.bin\windowskafka-server-start.bat .config\server.properties` in new command window and hit enter.

3.1.4 Setting up PostgreSQL

Step 1) Go to <https://www.postgresql.org/download>.

Note: Recommended version is postgres 10.

Step 2) You are given two options:-

1. Interactive Installer by EnterpriseDB
2. Graphical Installer by BigSQL

BigSQL currently installs pgAdmin version 3 which is deprecated. It’s best to choose EnterpriseDB which installs the latest version 4

Step 3)

1. You will be prompted to the desired Postgre version and operating system. Select the Postgres 10, as Scorpio has been tested and developed with this version.
2. Click the Download Button, Download will begin

Step 4) Open the downloaded .exe and Click next on the install welcome screen.

Step 5)

1. Change the Installation directory if required, else leave it to default

2. Click Next

Step 6)

1. You can choose the components you want to install in your system. You may uncheck Stack Builder

2. Click on Next

Step 7)

1. You can change the data location

2. Click Next

Step 8)

1. Enter the superuser password. Make a note of it

2. Click Next

Step 9)

1. Leave the port number as the default

2. Click Next

Step 10)

1. Check the pre-installation summary.

2. Click Next

Step 11) Click the next button

Step 12) Once install is complete you will see the Stack Builder prompt

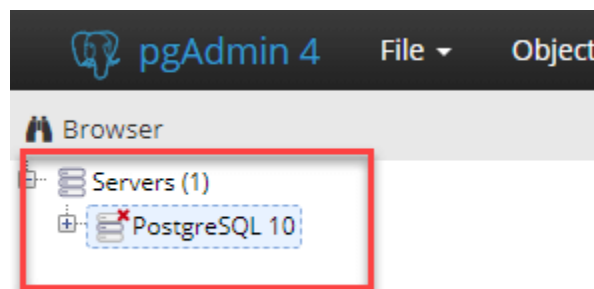
1. Uncheck that option. We will use Stack Builder in more advance tutorials

2. Click Finish

Step 13) To launch Postgre go to Start Menu and search pgAdmin 4

Step 14) You will see pgAdmin homepage

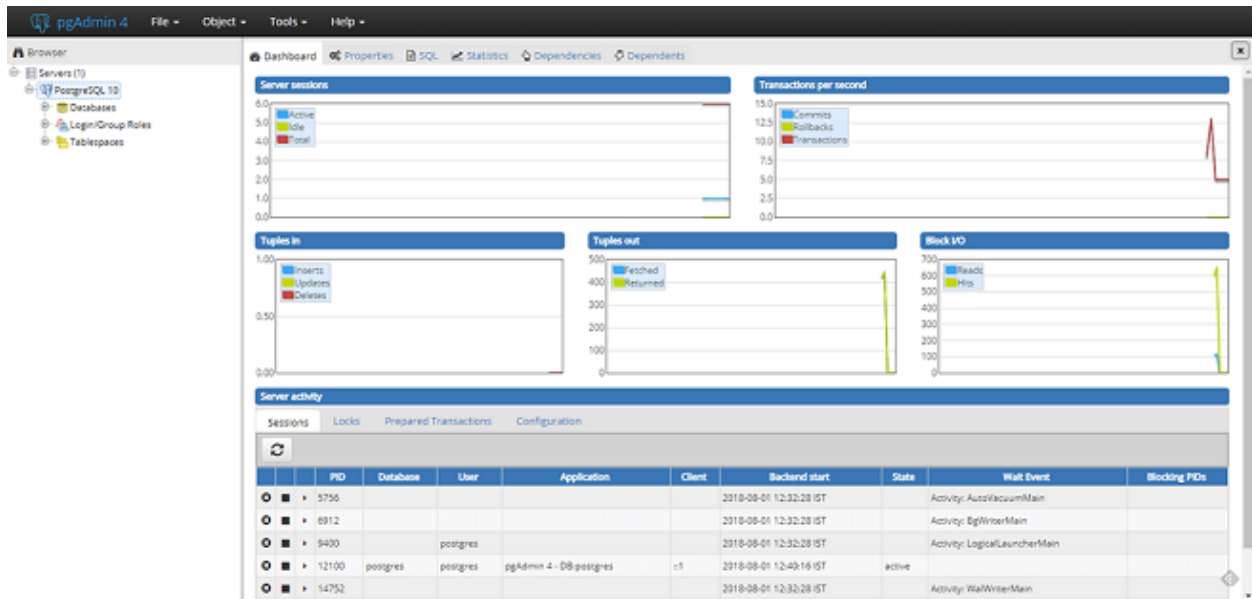
Step 15) Click on Servers > Postgre SQL 10 in the left tree



Step 16)

1. Enter superuser password set during installation

2. Click OK



Step 17) You will see the Dashboard

That's it to Postgre SQL installation.

3.2 Linux

3.2.1 JDK Setup

To create a Java environment in your machine install the JDK, for this open the terminal, and run the following commands:-

1. `sudo apt-get update`
2. `sudo apt-get install openjdk-8-jdk`

To check that JDK is properly installed in your machine, run the command **java -version** in your terminal if it returns the version of the JDK as 11 then it's working fine.

3.2.2 Setting Up Kafka

To download the Apache Kafka in your machine run the following commands one by one in your terminal.

1. `mkdir kafka`
2. `cd kafka`
3. `wget https://archive.apache.org/dist/kafka/2.2.0/kafka_2.12-2.2.0.tgz`
4. `tar -xzf kafka_2.12-2.2.0.tgz`

Once the Kafka is downloaded in your machine hit the following commands to get it run

1. `kafka_2.12-2.2.0/bin/zookeeper-server-start.sh kafka_2.12-2.2.0/config/zookeeper.properties > /dev/null 2>&1 &`

2. `kafka_2.12-2.2.0/bin/kafka-server-start.sh kafka_2.12-2.2.0/config/server.properties > /dev/null 2>&1 &`

3.2.3 Setting up PostgreSQL

In order to download the PostgreSQL in your machine run the following commands from your terminal.

1. `sudo apt update`
2. `sudo apt-get install postgresql-10`
3. `service postgresql status`

The last command will give us the status of the PostgreSQL four your machine if this matches to one in the picture then everything is properly installed else re-run the commands. .. figure:: figures/postgresTerminal

Once PostgreSQL is successfully installed in your machine create the database **ngb** and change its role by running the following commands:

1. `psql -U postgres -c "create database ngb;"`
2. `psql -U postgres -c "create user ngb with password 'ngb';"`
3. `psql -U postgres -c "alter database ngb owner to ngb;"`
4. `psql -U postgres -c "grant all privileges on database ngb to ngb;"`
5. `psql -U postgres -c "alter role ngb superuser;"`
6. `sudo apt install postgresql-10-postgis-2.4`
7. `sudo apt install postgresql-10-postgis-scripts`
8. `sudo -u postgres psql -U postgres -c "create extension postgis;"`

After this your PostgreSQL is ready to use for Scorpio Boker.

Starting Scorpio via docker-compose

4.1 Start commands to copy

Looking for the easiest way to start Scorpio? This is it.

```
curl https://raw.githubusercontent.com/ScorpioBroker/ScorpioBroker/development/docker-  
compose-aaio.yml  
sudo docker-compose -f docker-compose-aaio.yml up
```

4.2 Introduction

The easiest way to start Scorpio is to use docker-compose. We provide 2 main docker-compose files which rely on dockerhub. `docker-compose-aaio.yml` and `docker-compose-dist.yml`. You can use this files directly as they are to start Scorpio. When you want to run Scorpio in the distributed variant exchange the yml file in the command above.

4.3 docker-compose-aaio.yml

AAIO here stands for almost all in one. In this variant the core components of Scorpio and the Spring Cloud components are started within one container. Additional containers are only Kafka and Postgres. For testing and small to medium size deployments this is most likely what you want to use.

4.4 docker-compose-dist.yml

In this variant each Scorpio component is started in a different container. This makes it highly flexible and allows you to replace individual components or to start new instances of some core components.

4.5 Configure docker image via environment variables

There are multiple ways to enter environment variables into docker. We will not go through all of them but only through the docker-compose files. However the scorpio relevant parts apply to all these variants. Configuration of Scorpio is done via the Spring Cloud configuration system. For a complete overview of the used parameters and the default values have a look at the application.yml for the AllInOneRunner here, <https://github.com/ScorpioBroker/ScorpioBroker/blob/development/AllInOneRunner/src/main/resources/application-aaio.yml>. To provide a new setting you can provide those via an environment entry in the docker-compose file. The variable we want to set is called `spring_args`. Since we only want to set this option for the Scorpio container we make it a sub part of the Scorpio Container entry like this

```
scorpio:
  image: scorpiobroker/scorpio:scorpio-aaio_1.0.0
  ports:
    - "9090:9090"
  depends_on:
    - kafka
    - postgres
  environment:
    spring_args: --maxLimit=1000
```

With this we would set the maximum limit for a query reply to 1000 instead of the default 500.

4.6 Be quiet! docker

Some docker containers can be quite noisy and you don't want all of that output. The easy solution is to add this

```
logging:
driver: none
```

in the docker-compose file to respective container config. E.g. to make Kafka quite.

```
kafka:
  image: wurstmeister/kafka
  hostname: kafka
  ports:
    - "9092"
  environment:
    KAFKA_ADVERTISED_HOST_NAME: kafka
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_ADVERTISED_PORT: 9092
    KAFKA_LOG_RETENTION_MS: 10000
    KAFKA_LOG_RETENTION_CHECK_INTERVAL_MS: 5000
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  depends_on:
    - zookeeper
  logging:
    driver: none
```

Configuration Parameters

Scorpio uses the Spring Cloud/Boot configuration system. This is done via the `application.yml` files in the corresponding folders. The `AllInOneRunner` has a complete set of all available configuration options in them.

Those can be overwritten via the command line or in the docker case as described above.

| Config Option | Description | Default Value |
|----------------------------|---|---|
| atcontext.url | the url to be used for the internal context server | http://localhost:9090/ngsi-ld/contextes/ |
| bootstrap.servers | the host and port of the internal kafka | kafka:9092 (default used for docker) |
| broker.id a | unique id for the broker. needed for federation | Broker1 |
| broker.parent.location.url | url for the parent broker in a federation setup | SELF (meaning no federation) |
| broker.geo-Coverage | GeoJSON description of the coverage. used for registration in a federation setup. | empty |
| defaultLimit | The default limit for a query if no limit is provided | 50 |
| maxLimit | The maximum number of results in a query | 500 |
| reader.datasource.password | If you change the postgres setup here you set the password | ngb |
| reader.datasource.url | JDBC URL to postgres | jdbc:postgresql://postgres:5432/ngb?ApplicationName=ngb_storagemanager_reader |
| reader.datasource.username | username for the postgres db | ngb |
| writer.datasource.password | If you change the postgres setup here you set the password | ngb |
| writer.datasource.url | JDBC URL to postgres | jdbc:postgresql://postgres:5432/ngb?ApplicationName=ngb_storagemanager_writer |
| writer.datasource.username | username for the postgres db | ngb |
| spring.datasource.password | Same as above but used by flyway for database migration | ngb |
| spring.datasource.url | Same as above but used by flyway for database migration | jdbc:postgresql://postgres:5432/ngb?ApplicationName=ngb_storagemanager_writer |
| spring.datasource.username | Same as above but used by flyway for database migration | ngb |

Building Scorpio from source

Scorpio is developed in Java using SpringCloud as microservice framework and Apache Maven as build tool. Some of the tests require a running Apache Kafka messagebus (further instruction are in the Setup chapter). If you want to skip those tests you can run `mvn clean package -DskipTests` to just build the individual microservices.

6.1 General Remarks on Building

Further down this document you will get exact build commands/arguments for the different flavors. This part will give you an overview on how the different arguments work.

6.1.1 Maven Profiles

There currently three available Maven build profiles

Default

If you provide no `-P` argument Maven will produce individual jar files for the microservices and the `AllInOneRunner` with each “full” microservice packaged (this will result in ca. 500 MB size for the `AllInOneRunner`)

docker

This will trigger the Maven to build docker containers for each microservice.

docker-aaio

This will trigger the Maven to build one docker container, containing the `AllInOneRunner` and the spring cloud components (eureka, configserver and gateway)

Maven arguments

These arguments are provided via -D in the command line.

skipTests

Generally recommended if you want to speed up the build or you don't have a kafka instance running, which is required by some of the tests.

skipDefault

This is a special argument for the Scorpio build. This argument will disable springs repacking for the individual microservices and will allow for a smaller AllInOneRunner jar file. This argument should ONLY be used in combination with the docker-aaio profile.

6.1.2 Spring Profiles

Spring supports also profiles which can be activated when launching a jar file. Currently there 3 profiles actively used in Scorpio. The default profiles assume the default setup to be a individual microservices. The exception is the AllInOneRunner which as default assumes to be running in the docker-aaio setup.

Currently you should be able to run everything with a default profile except the gateway in combination with the AllInOneRunner. In order to use these two together you need to start the gateway with the aaio spring profile. This can be done by attaching this to your start command -Dspring.profiles.active=aaio.

Additionally some components have a dev profile available which is purely meant for development purposes and should only be used for such.

6.2 Setup

Scorpio requires two components to be installed.

6.2.1 Postgres

Please download the [Postgres DB](#) and the [Postgis](#) extension and follow the instructions on the websites to set them up.

Scorpio has been tested and developed with Postgres 10.

The default username and password which Scorpio uses is "ngb". If you want to use a different username or password you need to provide them as parameter when starting the StorageManager and the RegistryManager.

e.g.

```
java -jar Storage/StorageManager/target/StorageManager-<VERSIONNUMBER>-SNAPSHOT.jar --  
-reader.datasource.username=funkyusername --reader.datasource.password=funkypassword
```

OR

```
java -jar Registry/RegistryManager/target/RegistryManager-<VERSIONNUMBER>-SNAPSHOT.  
-jar --spring.datasource.username=funkyusername --spring.datasource.  
-password=funkypassword
```

Don't forget to create the corresponding user ("ngb" or the different username you chose) in postgres. It will be used by the SpringCloud services for database connection. While in terminal, log in to the psql console as postgres user:

```
sudo -u postgres psql
```

Then create a database "ngb":

```
postgres=# create database ngb;
```

Create a user "ngb" and make him a superuser:

```
postgres=# create user ngb with encrypted password 'ngb';
postgres=# alter user ngb with superuser;
```

Grant privileges on database:

```
postgres=# grant all privileges on database ngb to ngb;
```

Also create an own database/schema for the Postgis extension:

```
postgres=# CREATE DATABASE gisdb;
postgres=# \connect gisdb;
postgres=# CREATE SCHEMA postgis;
postgres=# ALTER DATABASE gisdb SET search_path=public, postgis, contrib;
postgres=# \connect gisdb;
postgres=# CREATE EXTENSION postgis SCHEMA postgis;
```

6.2.2 Apache Kafka

Scorpio uses [Apache Kafka](#) for the communication between the microservices.

Scorpio has been tested and developed with Kafka version 2.12-2.1.0

Please download [Apache Kafka](#) and follow the instructions on the website.

In order to start kafka you need to start two components: Start zookeeper with

```
<kafkafolder>/bin/[Windows]/zookeeper-server-start.[bat|sh] <kafkafolder>/config/
↪zookeeper.properties
```

Start kafkaserver with

```
<kafkafolder>/bin/[Windows]/kafka-server-start.[bat|sh] <kafkafolder>/config/server.
↪properties
```

For more details please visit the [Kafka website](#).

Getting a docker container

The current maven build supports two types of docker container generations from the build using maven profiles to trigger it.

The first profile is called 'docker' and can be called like this

```
sudo mvn clean package -DskipTests -Pdocker
```

this will generate individual docker containers for each micro service. The corresponding docker-compose file is `docker-compose-dist.yml`

The second profile is called 'docker-aaio' (for almost all in one). This will generate one single docker container for all components the broker except the kafka message bus and the postgres database.

To get the aaio version run the maven build like this

```
sudo mvn clean package -DskipTests -DskipDefault -Pdocker-aaio
```

The corresponding docker-compose file is `docker-compose-aaio.yml`

Starting the docker container

To start the docker container please use the corresponding docker-compose files. I.e.

```
sudo docker-composer -f docker-compose-aaio.yml up
```

to stop the container properly execute

```
sudo docker-composer -f docker-compose-aaio.yml down
```

General remark for the Kafka docker image and docker-compose

The Kafka docker container requires you to provide the environment variable `KAFKA_ADVERTISED_HOST_NAME`. This has to be changed in the docker-compose files to match your docker host IP. You can use `127.0.0.1` however this will disallow you to run Kafka in a cluster mode.

For further details please refer to <https://hub.docker.com/r/wurstmeister/kafka>

Running docker build outside of Maven

If you want to have the build of the jars separated from the docker build you need to provide certain VARS to docker. The following list shows all the vars and their intended value if you run docker build from the root dir

- `BUILD_DIR_ACS` = `Core/AtContextServer`
- `BUILD_DIR_SCS` = `SpringCloudModules/config-server`
- `BUILD_DIR_SES` = `SpringCloudModules/eureka`
- `BUILD_DIR_SGW` = `SpringCloudModules/gateway`
- `BUILD_DIR_HMG` = `History/HistoryManager`
- `BUILD_DIR_QMG` = `Core/QueryManager`
- `BUILD_DIR_RMG` = `Registry/RegistryManager`
- `BUILD_DIR_EMG` = `Core/EntityManager`
- `BUILD_DIR_STRMG` = `Storage/StorageManager`
- `BUILD_DIR_SUBMG` = `Core/SubscriptionManager`
- `JAR_FILE_BUILD_ACS` = `AtContextServer-${project.version}.jar`
- `JAR_FILE_BUILD_SCS` = `config-server-${project.version}.jar`
- `JAR_FILE_BUILD_SES` = `eureka-server-${project.version}.jar`

- JAR_FILE_BUILD_SGW = gateway-\${project.version}.jar
- JAR_FILE_BUILD_HMG = HistoryManager-\${project.version}.jar
- JAR_FILE_BUILD_QMG = QueryManager-\${project.version}.jar
- JAR_FILE_BUILD_RMG = RegistryManager-\${project.version}.jar
- JAR_FILE_BUILD_EMG = EntityManager-\${project.version}.jar
- JAR_FILE_BUILD_STRMG = StorageManager-\${project.version}.jar
- JAR_FILE_BUILD_SUBMG = SubscriptionManager-\${project.version}.jar
- JAR_FILE_RUN_ACS = AtContextServer.jar
- JAR_FILE_RUN_SCS = config-server.jar
- JAR_FILE_RUN_SES = eureka-server.jar
- JAR_FILE_RUN_SGW = gateway.jar
- JAR_FILE_RUN_HMG = HistoryManager.jar
- JAR_FILE_RUN_QMG = QueryManager.jar
- JAR_FILE_RUN_RMG = RegistryManager.jar
- JAR_FILE_RUN_EMG = EntityManager.jar
- JAR_FILE_RUN_STRMG = StorageManager.jar
- JAR_FILE_RUN_SUBMG = SubscriptionManager.jar

6.3 Starting of the components

After the build start the individual components as normal Jar files.

Start the SpringCloud services by running

```
java -jar SpringCloudModules/eureka/target/eureka-server-<VERSIONNUMBER>-SNAPSHOT.jar
java -jar SpringCloudModules/gateway/target/gateway-<VERSIONNUMBER>-SNAPSHOT.jar
java -jar SpringCloudModules/config-server/target/config-server-<VERSIONNUMBER>-
↪SNAPSHOT.jar
```

Start the broker components

```
java -jar Storage/StorageManager/target/StorageManager-<VERSIONNUMBER>-SNAPSHOT.jar
java -jar Core/QueryManager/target/QueryManager-<VERSIONNUMBER>-SNAPSHOT.jar
java -jar Registry/RegistryManager/target/RegistryManager-<VERSIONNUMBER>-SNAPSHOT.jar
java -jar Core/EntityManager/target/EntityManager-<VERSIONNUMBER>-SNAPSHOT.jar
java -jar History/HistoryManager/target/HistoryManager-<VERSIONNUMBER>-SNAPSHOT.jar
java -jar Core/SubscriptionManager/target/SubscriptionManager-<VERSIONNUMBER>-
↪SNAPSHOT.jar
java -jar Core/AtContextServer/target/AtContextServer-<VERSIONNUMBER>-SNAPSHOT.jar
```

6.3.1 Changing config

All configurable options are present in application.properties files. In order to change those you have two options. Either change the properties before the build or you can override configs by add --<OPTION_NAME>=<OPTION_VALUE> e.g.

```
java -jar Storage/StorageManager/target/StorageManager-<VERSIONNUMBER>-SNAPSHOT.jar --  
↪reader.datasource.username=funkyusername --reader.datasource.password=funkypassword`
```

6.3.2 Enable CORS support

You can enable cors support in the gateway by providing these configuration options - gateway.enablecors - default is False. Set to true for general enabling - gateway.enablecors.allowall - default is False. Set to true to enable CORS from all origins, allow all headers and all methods. Not secure but still very often used. - gateway.enablecors.allowedorigin - A comma separated list of allowed origins - gateway.enablecors.allowedheader - A comma separated list of allowed headers - gateway.enablecors.allowedmethods - A comma separated list of allowed methods - gateway.enablecors.allowallmethods - default is False. Set to true to allow all methods. If set to true it will override the allowmethods entry

6.4 Troubleshooting

6.4.1 Missing JAXB dependencies

When starting the eureka-server you may facing the

java.lang.TypeNotPresentException: Type javax.xml.bind.JAXBContext not present exception. It's very likely that you are running Java 11 on your machine then. Starting from Java 9 package `javax.xml.bind` has been marked deprecated and was finally completely removed in Java 11.

In order to fix this issue and get eureka-server running you need to manually add below JAXB Maven dependencies to ScorpioBroker/SpringCloudModules/eureka/pom.xml before starting:

```
...  
<dependencies>  
  ...  
  <dependency>  
    <groupId>com.sun.xml.bind</groupId>  
    <artifactId>jaxb-core</artifactId>  
    <version>2.3.0.1</version>  
  </dependency>  
  <dependency>  
    <groupId>javax.xml.bind</groupId>  
    <artifactId>jaxb-api</artifactId>  
    <version>2.3.1</version>  
  </dependency>  
  <dependency>  
    <groupId>com.sun.xml.bind</groupId>  
    <artifactId>jaxb-impl</artifactId>  
    <version>2.3.1</version>  
  </dependency>  
  ...  
</dependencies>  
...
```

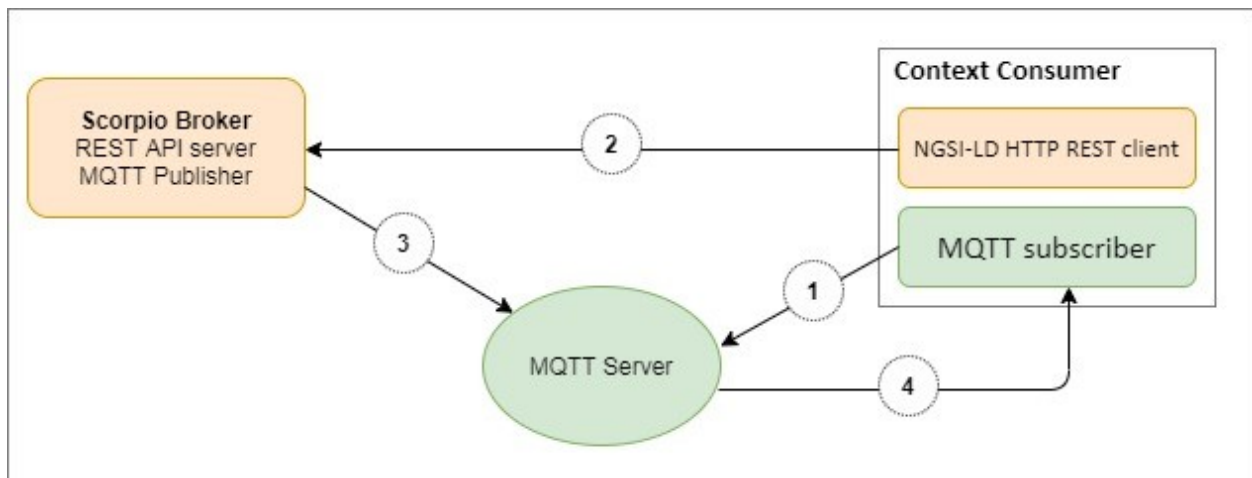
This should be fixed now using conditional dependencies.

MQTT Notifications

MQTT is a pub/sub based message bus and works with topics. For more detailed information please visit <https://mqtt.org/>. NGSI-LD allows you to receive notifications via MQTT. A subscription received via HTTP specifies an MQTT endpoint in the “notification.endpoint.uri” member of the subscription and the MQTT notification binding is supported by the NGSI-LD implementation, notifications related to this subscription shall be sent via the MQTT protocol.

The syntax of an MQTT endpoint URI is `mqtt[s]://[<username>][:<password>]@<host>[:<port>]/<topic>[/<subtopic>]*` and follows an existing convention for representing an MQTT endpoint as a URI.

Username and password can be optionally specified as part of the endpoint URI. If the port is not explicitly specified, the default MQTT port is **1883** for MQTT over TCP and **8883** for MQTTS, For the MQTT protocol, there are currently two versions supported, **MQTtv3.1.1** and **MQTtv5.0**.



The flow of Scorpio broker notification via MQTT:-

1. Subscribe to TOPIC.
2. Create NGSI-LD Subscription, with MQTT Server’s URI as a contact point to send Notifications.

3. Publish Notifications to TOPIC extracted from URI.
4. Send Notifications from the MQTT server to the MQTT subscriber.

To start the MQTT broker follow the below step:-

1. Install the MQTT broker (Mosquitto).
2. Add chrome extension MQTTlens.
3. Create the MQTT broker connection.
4. Subscribe the topic.

7.1 Operations

7.1.1 1. Entity Creation

To create the entity, hit the endpoint **http://<IP Address>:<port>/ngsi-ld/v1/entities/** with the given payload.

```
{
  "id": "urn:ngsi-ld:Vehicle:A135",
  "type": "Vehicle",
  "brandName": {
    "type": "Property",
    "value": "Mercedes"
  },
  "speed": [{
    "type": "Property",
    "value": 55,
    "datasetId": "urn:ngsi-ld:Property:speedometerA4567-speed",
    "source": {
      "type": "Property",
      "value": "Speedometer"
    }
  }],
  {
    "type": "Property",
    "value": 11,
    "datasetId": "urn:ngsi-ld:Property:gpsA4567-speed",
    "source": {
      "type": "Property",
      "value": "GPS"
    }
  },
  {
    "type": "Property",
    "value": 10,
    "source": {
      "type": "Property",
      "value": "CAMERA"
    }
  }
}]
}
```

7.1.2 2. Subscription

To subscribe to the entity, hit the endpoint **http://<IP Address>:<port>/ngsi-ld/v1/subscriptions/** with the given payload.

```
{
  "id": "urn:ngsi-ld:Subscription:16",
  "type": "Subscription",
  "entities": [{
    "id": "urn:ngsi-ld:Vehicle:A135",
    "type": "Vehicle"
  }],
  "watchedAttributes": ["brandName"],
  "q": "brandName!=Mercedes",
  "notification": {
    "attributes": ["brandName"],
    "format": "keyValues",
    "endpoint": {
      "uri": "mqtt://localhost:1883/notify",
      "accept": "application/json",
      "notifierinfo": {
        "version": "mqtt5.0",
        "qos": 0
      }
    }
  }
}
```

7.1.3 3. Notification

If we update the value of the attribute and making the PATCH request at **http://<IP Address>:<port>/ngsi-ld/v1/entities/entityId/attrs**

```
{
  "brandName": {
    "type": "Property",
    "value": "BMW"
  }
}
```

then, we get the notification.

```
{
  "metadata": {
    "link": "https://json-ld.org/contexts/person.jsonld",
    "contentType": "application/json"
  },
  "body": {
    "id": "ngsildbroker:notification:-7550927064189664633",
    "type": "Notification",
    "data": [{
      "id": "urn:ngsi-ld:Vehicle:A135",
      "type": "Vehicle",
      "brandName": {
        "type": "Property",
        "createdAt": "2020-07-29T07:19:33.872000Z",

```

(continues on next page)

(continued from previous page)

```
"value": "BMW",
"modifiedAt": "2020-07-29T07:51:21.183000Z"
},
],
"notifiedAt": "2020-07-29T07:51:22.300000Z",
"subscriptionId": "urn:ngsi-ld:Subscription:16"
}
}
```

This walkthrough adopts a practical approach that we hope will help our readers to get familiar with NGSI-LD in general and the Scorpio Broker in particular - and have some fun in the process :).

The walkthrough is based on the NGSI-LD Specification, that can be found in here [https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.02.02_60/gs_CIM009v010202p.pdf]. -> will become gs_CIM009v010301p.pdf soon ... You should also have a look at the NGSI-LD implementation notes. -> once they are available To get familiar with NGSI-LD, you may also have a look at the NGSI-LD Primer [https://www.etsi.org/deliver/etsi_gr/CIM/001_099/008/01.01.01_60/gr_CIM008v010101p.pdf] that is targeted at developers.

The main section is about context management. It describes the basic context broker functionality for context management (information about entities, such as the temperature of a car). Context source management (information not about the entities themselves, but about the sources that can provide the information in a distributed system setup) is also described as part of this document.

It is recommended to get familiar with the theoretical concepts on which the NGSI-LD model is based before starting. E.g. entities, properties, relationships etc. Have a look at the FIWARE documentation about this, e.g. this public presentation. [... find suitable presentation]

8.1 Starting the Scorpio Broker for the tutorials

In order to start the broker we recommend to use docker-compose. Get the docker-compose file from the github repo of Scorpio.

```
curl https://raw.githubusercontent.com/ScorpioBroker/ScorpioBroker/development/docker-  
compose-aaio.yml
```

and start the container with

```
sudo docker-compose -f docker-compose-aaio.yml up
```

You can also start the broker without docker. For further instructions please refer to the readme <https://github.com/ScorpioBroker/ScorpioBroker/blob/development/README.md>

8.2 Issuing commands to the broker

To issue requests to the broker, you can use the curl command line tool. curl is chosen because it is almost ubiquitous in any GNU/Linux system and simplifies including examples in this document that can easily be copied and pasted. Of course, it is not mandatory to use it, you can use any REST client tool instead (e.g. RESTClient). Indeed, in a real case, you will probably interact with the Scorpio Broker using a programming language library implementing the REST client part of your application.

The basic patterns for all the curl examples in this document are the following:

For POST: curl localhost:9090/ngsi-ld/v1/<ngsi-ld-resource-path> -s -S [headers] -d @- <<EOF [payload] EOF
For PUT: curl localhost:9090/ngsi-ld/v1/<ngsi-ld-resource-path> -s -S [headers] -X PUT -d @- <<EOF [payload] EOF
For PATCH: curl localhost:9090/ngsi-ld/v1/<ngsi-ld-resource-path> -s -S [headers] -X PATCH -d @- <<EOF [payload] EOF
For GET: curl localhost:9090/ngsi-ld/v1/<ngsi-ld-resource-path> -s -S [headers]
For DELETE: curl localhost:9090/ngsi-ld/v1/<ngsi-ld-resource-path> -s -S [headers] -X DELETE
Regarding [headers] you have to include the following ones:

Accept header to specify the payload format in which you want to receive the response. You should explicitly specify JSON or JSON-LD. curl ... -H 'Accept: application/json' ... or curl ... -H 'Accept: application/ld+json' depending on whether you want to receive the JSON-LD @context in a link header or in the body of the response (JSON-LD and the use of @context is described in the following section).

If using payload in the request (i.e. POST, PUT or PATCH), you have to supply the Context-Type HTTP header to specify the format (JSON or JSON-LD). curl ... -H 'Content-Type: application/json' ... or -H 'Content-Type: application/ld+json'

In case the JSON-LD @context is not provided as part of the request body, it has to be provided as a link header, e.g. curl ... -H 'Link: <<https://uri.etsi.org/ngsi-ld/primer/store-context.jsonld>>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"' where the @context has to be retrievable from the first URI, i.e. in this example: <https://uri.etsi.org/ngsi-ld/primer/store-context.jsonld>

Some additional remarks:

Most of the time we are using multi-line shell commands to provide the input to curl, using EOF to mark the beginning and the end of the multi-line block (here-documents). In some cases (GET and DELETE) we omit -d @- as no payload is used.

In the examples, it is assumed that the broker is listening on port 9090. Adjust this in the curl command line if you are using a different port.

In order to pretty-print JSON in responses, you can use Python with msjson.tool (examples along with tutorial are using this style):

```
(curl ... | python -mjson.tool) <<EOF ... EOF
```

Check that curl is installed in your system using:

```
which curl
```

8.3 NGSI-LD data in 3 sentences

NGSI-LD is based on JSON-LD. Your toplevel entries are NGSI-LD Entities. Entities can have Properties and Relationships and Properties and Relationships can themselves also have Properties and Relationships (meta information). All keys in the JSON-LD document must be URIs, but there is a way to shorten it.

8.4 @context

NGSI-LD builds upon JSON-LD. Coming from JSON-LD there is the concept of a mandatory @context entry which is used to ‘translate’ between expanded full URIs and a compacted short form of the URI. e.g. “Property”: “<https://uri.etsi.org/ngsi-ld/Property>”. @context entries can also be linked in via a URL in a JSON array. You can also mix this up, so this is perfectly fine.

```
{
  "@context": [{
    "myshortname": "urn:mylongname"
  },
  "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
]
```

NGSI-LD has a core context made available at <https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld>. Even though it is highly recommended to always provide a full entry of all used @context entries, Scorpio and other NGSI-LD brokers will inject the core context on any entry where it is missing.

8.5 application/json and application/ld+json

You can provide and receive data in two different ways. The main difference between application/json and application/ld+json is where you provide or receive the mandatory @context entry. If you set the accept header or the content-type header to application/ld+json the @context entry is embedded in the JSON document as a root level entry. If it is set to application/json the @context has to be provided in a link in the header entry Link like this. <<https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld>>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"

8.6 Context Management

To show the use of @context, most examples in this tutorial will be done as application/ld+json having the @context entries in the body of the payload. At the end of this section, you will have the basic knowledge to create applications (both context producers and consumers) using the Scorpio Broker with context management operations.

Entity creation

Assuming a fresh start we have an empty Scorpio Broker. First, we are going to create `house2:smartrooms:room1`. Let's assume that at entity creation time, temperature is 23 °C and it is part of `smartcity:houses:house2`.

```
curl localhost:9090/ngsi-ld/v1/entities -s -S -H 'Content-Type: application/ld+json' -  
→d @- <<EOF  
{  
  "id": "house2:smartrooms:room1",  
  "type": "Room",  
  "temperature": {  
    "value": 23,  
    "unitCode": "CEL",  
    "type": "Property",  
    "providedBy": {  
      "type": "Relationship",  
      "object": "smartbuilding:house2:sensor0815"  
    }  
  },  
  "isPartOf": {  
    "type": "Relationship",  
    "object": "smartcity:houses:house2"  
  },  
  "@context": [{"Room": "urn:mytypes:room", "temperature": "myuniqueuri:temperature",  
→ "isPartOf": "myuniqueuri:isPartOf"}, {"https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-  
→ context.jsonld"}]  
}  
EOF
```

Apart from the `id` and `type` fields (that define the ID and type of the entity), the payload contains a set of attributes. As you can see, there are two types of attributes. Properties and Relationships. Properties directly provide a value of an attribute. Additionally there is an optional parameter `unitCode` which can be used to better describe the value using unit codes described in UN/CEFACT Common Codes for Units of Measurement. UnitCodes should be seen as an additional metadata provided by the producer. They are not restrictive. There is no validation on the value field.

Relationships always point to another Entity encoded as the object of a relationship. They are used to describe the relations between various entities. Properties and Relationship can themselves have Relationships, enabling the rep-

resentation of meta information. As you can see we also added a Relationship to the temperature Property pointing to an Entity describing the sensor from which this information has been received.

Upon receipt of this request, Scorpio creates the entity in its internal database and takes care of any further handling required because of the creation, e.g. subscription handling or creating historical entries. Once the request is validated Scorpio responds with a 201 Created HTTP code.

Next, let's create house2:smartrooms:room2 in a similar way.

```
curl localhost:9090/ngsi-ld/v1/entities -s -S -H 'Content-Type: application/ld+json' -
↳d @- <<EOF
{
  "id": "house2:smartrooms:room2",
  "type": "Room",
  "temperature": {
    "value": 21,
    "unitCode": "CEL",
    "type": "Property",
    "providedBy": {
      "type": "Relationship",
      "object": "smartbuilding:house2:sensor4711"
    }
  },
  "isPartOf": {
    "type": "Relationship",
    "object": "smartcity:houses:house2"
  },
  "@context": [{"Room": "urn:mytypes:room", "temperature": "myuniqueuri:temperature",
↳ "isPartOf": "myuniqueuri:isPartOf"}, {"https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-
↳ context.jsonld"}]
}
EOF
```

Now to complete this setup we are creating an Entity describing our house with the id smartcity:houses:house2.

```
curl localhost:9090/ngsi-ld/v1/entities -s -S -H 'Content-Type: application/ld+json' -
↳d @- <<EOF
{
  "id": "smartcity:houses:house2",
  "type": "House",
  "hasRoom": [{
    "type": "Relationship",
    "object": "house2:smartrooms:room1",
    "datasetId": "somethingunique1"
  },
  {
    "type": "Relationship",
    "object": "house2:smartrooms:room2",
    "datasetId": "somethingunique2"
  }],
  "location": {
    "type": "GeoProperty",
    "value": {
      "type": "Polygon",
      "coordinates": [[[-8.5, 41.2], [-8.5000001, 41.2], [-8.
↳ 5000001, 41.2000001], [-8.5, 41.2000001], [-8.5, 41.2]]]
    }
  },
}
```

(continues on next page)

(continued from previous page)

```
"entrance": {
  "type": "GeoProperty",
  "value": {
    "type": "Point",
    "coordinates": [-8.50000005, 41.2]
  }
},
"@context": [{"House": "urn:mytypes:house", "hasRoom": "myuniqueuri:hasRoom"},
↪ "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"]
}
EOF
```

Even though you can of course model this differently, for this scenario we model the relationships of houses with rooms with a `hasRoom` entry as a multi-relationship. To uniquely identify the entries they have a `datasetId`, which is also used when updating this specific relationship. There can be at most one relationship instance per relationship without a `datasetId`, which is considered to be the “default” instance. In the case of properties, multi-properties are represented in the same way. Additionally we are using a third type of attribute here the `GeoProperty`. `GeoProperty` values are GeoJSON values, allowing the description of various shapes and forms using longitude and latitude. Here we add to entries location, describing the outline of the house, and entrance, pointing to the entrance door.

As you might have seen, we haven’t provided an `@context` entry for ‘entrance’ and unlike ‘location’ it is not part of the core context. This will result in Scorpio storing the entry using a default prefix defined in the core context. The result in this case would be “<https://uri.etsi.org/ngsi-ld/default-context/entrance>”.

Apart from simple values corresponding to JSON datatypes (i.e. numbers, strings, booleans, etc.) for attribute values, complex structures or custom metadata can be used.

CHAPTER 10

Querying & receiving entities

Taking the role of a consumer application, we want to access the context information stored in Scorpio. NGSI-LD has two ways to get entities. You can either receive a specific entity using a GET /ngsi-ld/v1/entities/{id} request. The alternative is to query for a specific set of entities using the NGSI-LD query language.

If we want to just get the house in our example we would do a GET request like this.

```
curl localhost:9090/ngsi-ld/v1/entities/smartcity%3Ahouses%3Ahouse2 -s -S -H 'Accept:↵
↵application/ld+json'
```

Mind the url encoding here, i.e. ‘.’ gets replaced by %3A. For consistency you should always encode your URLs.

Since we didn’t provide our own @context in this request, only the parts of the core context will be replaced in the reply.

```
{
  "id": "smartcity:houses:house2",
  "type": "urn:mytypes:house",
  "myuniqueuri:hasRoom": [{
    "type": "Relationship",
    "object": "house2:smartrooms:room1",
    "datasetId": "somethingunique1"
  },
  {
    "type": "Relationship",
    "object": "house2:smartrooms:room2",
    "datasetId": "somethingunique2"
  }],
  "location": {
    "type": "GeoProperty",
    "value": {
      "type": "Polygon",
      "coordinates": [[[-8.5, 41.2], [-8.5000001, 41.2], [-8.↵
↵5000001, 41.2000001], [-8.5, 41.2000001], [-8.5, 41.2]]]
    }
  },
}
```

(continues on next page)

(continued from previous page)

```

    "entrance": {
      "type": "GeoProperty",
      "value": {
        "type": "Point",
        "coordinates": [-8.50000005, 41.2]
      }
    }
    "@context": ["https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"]
  }
}

```

As you can see entrance was compacted properly since it is was prefixed from the default context specified in the core context.

Assuming we are hosting our own @context file on a webserver, we can provide it via the 'Link' header. For convenience we are using pastebin in this example Our context looks like this.

```

{
  "@context": [{
    "House": "urn:mytypes:house",
    "hasRoom": "myuniqueuri:hasRoom",
    "Room": "urn:mytypes:room",
    "temperature": "myuniqueuri:temperature",
    "isPartOf": "myuniqueuri:isPartOf"
  }, "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"]
}

```

We repeat this call providing our @context via the 'Link' like this

```

curl localhost:9090/ngsi-ld/v1/entities/smartcity%3Ahouses%3Ahouse2 -s -S -H 'Accept:↵
↵application/ld+json' -H 'Link: <https://pastebin.com/raw/Mgxv2ykn>; rel="http://www.
↵w3.org/ns/json-ld#context"; type="application/ld+json"'

```

The reply now looks like this.

```

{
  "id": "smartcity:houses:house2",
  "type": "House",
  "hasRoom": [{
    "type": "Relationship",
    "object": "house2:smartrooms:room1",
    "datasetId": "somethingunique1"
  },
  {
    "type": "Relationship",
    "object": "house2:smartrooms:room2",
    "datasetId": "somethingunique2"
  }],
  "location": {
    "type": "GeoProperty",
    "value": {
      "type": "Polygon",
      "coordinates": [[[-8.5, 41.2], [-8.5000001, 41.2], [-8.
↵5000001, 41.2000001], [-8.5, 41.2000001], [-8.5, 41.2]]]
    }
  },
  "entrance": {
    "type": "GeoProperty",

```

(continues on next page)

(continued from previous page)

```

        "value": {
            "type": "Point",
            "coordinates": [-8.50000005, 41.2]
        },
        "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
    }

```

Since we provide the core context in our own @context it is not added to the result. From here on we will use the custom @context so we can use the short names in all of our requests.

You can also request an entity with a single specified attribute, using the attrs parameter. For example, to get only the location:

```

curl localhost:9090/ngsi-ld/v1/entities/smartcity%3Ahouses%3Ahouse2/?attrs=location -
→s -S -H 'Accept: application/ld+json' -H 'Link: <https://pastebin.com/raw/Mgxv2ykn>;
→ rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"'

```

Response:

```

{
    "id": "smartcity:houses:house2",
    "type": "House",
    "location": {
        "type": "GeoProperty",
        "value": {
            "type": "Polygon",
            "coordinates": [[[-8.5, 41.2], [-8.5000001, 41.2], [-8.
→5000001, 41.2000001], [-8.5, 41.2000001], [-8.5, 41.2]]]
        }
    },
    "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}

```

10.1 Query

The second way to retrieve information is the NGSI-LD query. For this example we first add a new Room which belongs to another house.

```

curl localhost:9090/ngsi-ld/v1/entities -s -S -H 'Content-Type: application/ld+json' -
→d @- <<EOF
{
    "id": "house99:smartrooms:room42",
    "type": "Room",
    "temperature": {
        "value": 21,
        "unitCode": "CEL",
        "type": "Property",
        "providedBy": {
            "type": "Relationship",
            "object": "smartbuilding:house99:sensor36"
        }
    },
    "isPartOf": {

```

(continues on next page)

(continued from previous page)

```

        "type": "Relationship",
        "object": "smartcity:houses:house99"
    },
    "@context": [{"Room": "urn:mytypes:room", "temperature": "myuniqueuri:temperature",
↪ "isPartOf": "myuniqueuri:isPartOf"}, {"https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-
↪ context.jsonld"}]
}
EOF

```

Let's assume we want to retrieve all the rooms in Scorpio. To do that we do a GET request like this

```

curl localhost:9090/ngsi-ld/v1/entities/?type=Room -s -S -H 'Accept: application/json
↪ ' -H 'Link: <https://pastebin.com/raw/Mgxv2ykn>; rel="http://www.w3.org/ns/json-ld
↪ #context"; type="application/ld+json"'

```

Note that this request has the accept header `application/json`, i.e. the link to the `@context` is returned in a link header. The result is

```

[
{
  "id": "house2:smartrooms:room1",
  "type": "Room",
  "temperature": {
    "value": 23,
    "unitCode": "CEL",
    "type": "Property",
    "providedBy": {
      "type": "Relationship",
      "object": "smartbuilding:house2:sensor0815"
    }
  },
  "isPartOf": {
    "type": "Relationship",
    "object": "smartcity:houses:house2"
  }
},
{
  "id": "house2:smartrooms:room2",
  "type": "Room",
  "temperature": {
    "value": 21,
    "unitCode": "CEL",
    "type": "Property",
    "providedBy": {
      "type": "Relationship",
      "object": "smartbuilding:house2:sensor4711"
    }
  },
  "isPartOf": {
    "type": "Relationship",
    "object": "smartcity:houses:house2"
  }
},
{
  "id": "house99:smartrooms:room42",

```

(continues on next page)

(continued from previous page)

```

    "type": "Room",
    "temperature": {
      "value": 21,
      "unitCode": "CEL",
      "type": "Property",
      "providedBy": {
        "type": "Relationship",
        "object": "smartbuilding:house99:sensor36"
      }
    },
    "isPartOf": {
      "type": "Relationship",
      "object": "smartcity:houses:house99"
    }
  }
}
]

```

10.2 Filtering

NGSI-LD provides a lot of ways to filter Entities from query results (and subscription notifications respectively). Since we are only interested in our smartcity:houses:house2, we are using the ‘q’ filter on the Relationship isPartOf. (URL encoding “smartcity:houses:house2” becomes %22smartcity%3Ahouses%3Ahouse2%22)

```

curl localhost:9090/ngsi-ld/v1/entities/?type=Room\&q=isPartOf==%22smartcity%3Ahouses
↪%3Ahouse2%22 -s -S -H 'Accept: application/json' -H 'Link: <https://pastebin.com/
↪raw/Mgxv2ykn>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json
↪ '

```

The results now looks like this.

```

[
{
  "id": "house2:smartrooms:room1",
  "type": "Room",
  "temperature": {
    "value": 23,
    "unitCode": "CEL",
    "type": "Property",
    "providedBy": {
      "type": "Relationship",
      "object": "smartbuilding:house2:sensor0815"
    }
  },
  "isPartOf": {
    "type": "Relationship",
    "object": "smartcity:houses:house2"
  }
},
{
  "id": "house2:smartrooms:room2",
  "type": "Room",
  "temperature": {
    "value": 21,

```

(continues on next page)

(continued from previous page)

```

        "unitCode": "CEL",
        "type": "Property"
        "providedBy": {
            "type": "Relationship",
            "object": "smartbuilding:house2:sensor4711"
        }
    },
    "isPartOf": {
        "type": "Relationship",
        "object": "smartcity:houses:house2"
    }
}
]

```

Now an alternative way to get the same result would be using the `idPattern` parameter, which allows you to use regular expressions. This is possible in this case since we structured our IDs for the rooms.

```

curl localhost:9090/ngsi-ld/v1/entities/?type=Room&idPattern=house2%3Asmartrooms
↪%3Aroom.%2A -s -S -H 'Accept: application/json' -H 'Link: <https://pastebin.com/raw/
↪Mgxv2ykn>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"'
(house2%3Asmartrooms%3Aroom.%2A == house2:smartrooms:room.*)

```

10.3 Limit the attributes

Additionally we now want to limit the result to only give us the temperature. This is done by using the `attrs` parameter. `Attrs` takes a comma separated list. In our case since it's only one entry it looks like this.

```

curl localhost:9090/ngsi-ld/v1/entities/?type=Room&q=isPartOf==%22smartcity%3Ahouses
↪%3Ahouse2%22&attrs=temperature -s -S -H 'Accept: application/json' -H 'Link:
↪<https://pastebin.com/raw/Mgxv2ykn>; rel="http://www.w3.org/ns/json-ld#context";
↪type="application/ld+json"'

```

```

[
  {
    "id": "house2:smartrooms:room1",
    "type": "Room",
    "temperature": {
      "value": 23,
      "unitCode": "CEL",
      "type": "Property",
      "providedBy": {
        "type": "Relationship",
        "object": "smartbuilding:house2:sensor0815"
      }
    }
  },
  {
    "id": "house2:smartrooms:room2",
    "type": "Room",
    "temperature": {
      "value": 21,
      "unitCode": "CEL",

```

(continues on next page)

(continued from previous page)

```
    "type": "Property"
    "providedBy": {
      "type": "Relationship",
      "object": "smartbuilding:house2:sensor4711"
    }
  }
}
```

10.4 KeyValues results

Now assuming we want to limit the payload of the request even more since we are really only interested in the value of temperature and don't care about any meta information. This can be done using the keyValues option. KeyValues will return a condensed version of the Entity providing only top level attribute and their respective value or object.

```
curl localhost:9090/ngsi-ld/v1/entities/?type=Room\&q=isPartOf==%22smartcity%3Ahouses
↳%3Ahouse2%22\&attrs=temperature\&options=keyValues -s -S -H 'Accept: application/
↳json' -H 'Link: <https://pastebin.com/raw/Mgxv2ykn>; rel="http://www.w3.org/ns/json-
↳ld#context"; type="application/ld+json"'
```

Response:

```
[
{
  "id": "house2:smartrooms:room1",
  "type": "Room",
  "temperature": 23
},
{
  "id": "house2:smartrooms:room2",
  "type": "Room",
  "temperature": 21
}
]
```

Updating an entity & appending to an entity

NGSI-LD allows you to update entities (overwrite the current entry) but also to just append new attributes. Additionally you can of course just update a specific attribute. Taking the role of the Context Producer for the temperature for house2:smartrooms:room1 we will cover 5 scenarios. 1. Updating the entire entity to push new values. 2. Appending a new Property providing the humidity from the room. 3. Partially updating the value of the temperature. 4. Appending a new multi value entry to temperature providing the info in degree Kelvin 5. Updating the specific multi value entries for temperature and Fahrenheit.

11.1 Update Entity

You can basically update every part of an entity with two exceptions. The type and the id are immutable. An update in NGSI-LD overwrites the existing entry. This means if you update an entity with a payload which does not contain a currently existing attribute it will be removed. To update our room1 we will do an HTTP POST like this.

```
curl localhost:9090/ngsi-ld/v1/entities/house2%3Asmartrooms%3Aroom1 -s -S -H 'Content-
↪Type: application/json' -H 'Link: https://pastebin.com/raw/Mgxv2ykn' -d @- <<EOF
{
    "temperature": {
        "value": 25,
        "unitCode": "CEL",
        "type": "Property",
        "providedBy": {
            "type": "Relationship",
            "object": "smartbuilding:house2:sensor0815"
        }
    },
    "isPartOf": {
        "type": "Relationship",
        "object": "smartcity:houses:house2"
    }
}
EOF
```

Now this is a bit much payload to update one value and there is a risk that you might accidentally delete something and we would only recommend this entity update if you really want to update a bigger part of an entity.

11.2 Partial update attribute

To take care of a single attribute update NGSI-LD provides a partial update. This is done by a POST on `/entities/<entityId>/attrs/<attributeName>` In order to update the temperature we do a POST like this

```
curl localhost:9090/ngsi-ld/v1/entities/house2%3Asmartrooms%3Aroom1/attrs/temperature_
↪-s -S -H 'Content-Type: application/json' -H 'Link: https://pastebin.com/raw/
↪Mgxv2ykn' -d @- <<EOF
{
    "value": 26,
    "unitCode": "CEL",
    "type": "Property",
    "providedBy": {
        "type": "Relationship",
        "object": "smartbuilding:house2:sensor0815"
    }
}
EOF
```

11.3 Append attribute

In order to append a new attribute to an entity you execute an HTTP PATCH command on `/entities/<entityId>/attrs/` with the new attribute as payload. Append in NGSI-LD by default will overwrite an existing entry. If this is not desired you can add the option parameter with `noOverwrite` to the URL like this `/entities/<entityId>/attrs?options=noOverwrite`. Now if we want to add an additional entry for the humidity in room1 we do an HTTP PATCH like this.

```
curl localhost:9090/ngsi-ld/v1/entities/house2%3Asmartrooms%3Aroom1/attrs -s -S -X_
↪PATCH -H 'Content-Type: application/json' -H 'Link: https://pastebin.com/raw/
↪Mgxv2ykn' -d @- <<EOF
{
    "humidity": {
        "value": 34,
        "unitCode": "PER",
        "type": "Property",
        "providedBy": {
            "type": "Relationship",
            "object": "smartbuilding:house2:sensor2222"
        }
    }
}
```

11.4 Add a multivalue attribute

NGSI-LD also allows us to add new multi value entries. We will do this by adding a unique datasetId. If a datasetId is provided in an append it will only affect the entry with the given datasetId. Adding the temperature in Fahrenheit we do a PATCH call like this.


```
curl localhost:9090/ngsi-ld/v1/entities/house2%3Asmartrooms%3Aroom1/attrs/temperature_
↵-s -S -H 'Content-Type: application/json' -H 'Link: https://pastebin.com/raw/
↵Mgxv2ykn' -d @- <<EOF
{
    "value": 78,8,
    "unitCode": "FAH",
    "type": "Property",
    "providedBy": {
        "type": "Relationship",
        "object": "smartbuilding:house2:sensor0815"
    }
    "datasetId": "urn:fahrenheitentry:0815"
}
EOF
```

Subscriptions

NGSI-LD defines a subscription interface which allows you to get notifications on Entities. Subscriptions are on change subscriptions. This means you will not get a notification on an initial state of an entity as the result of a subscription. Subscriptions at the moment issue a notification when a matching Entity is created, updated or appended to. You will not get a notification when an Entity is deleted.

12.1 Subscribing to entities

In order to get the temperature of our rooms we will formulate a basic subscription which we can POST to the /ngsi-ld/v1/subscriptions endpoint.

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↪ld+json' -d @- <<EOF
{
  "id": "urn:subscription:1",
  "type": "Subscription",
  "entities": [{
    "type": "Room"
  }],
  "notification": {
    "endpoint": {
      "uri": "http://ptsv2.com/t/30xad-1596541146/post",
      "accept": "application/json"
    }
  },
  "@context": ["https://pastebin.com/raw/Mgxv2ykn"]
}
EOF
```

As you can see entities is an array, which allows you to define multiple matching criteria for a subscription. You can subscribe by id or idPattern (regex) if you want. However a type is always mandatory in an entities entry.

12.2 Notification Endpoint

NGSI-LD currently supports two types of endpoints for subscriptions. HTTP(S) and MQTT(S). In the notification entry of a subscription you can define your endpoint with a uri and an accept MIME type. As you can see we are using an HTTP endpoint.

12.3 Testing notification endpoint

For this example we are using Post Test Server V2 (<http://ptsv2.com/>). This is a public service without auth on our example. So be careful with your data. Also this service is meant for testing and debugging and NOT more. So be nice! They are giving us a good tool for development. Normally you can use the example just as is. However if for some reason our endpoint is deleted please just go to ptsv2.com and click on “New Random Toilet” and replace the endpoint with the POST URL provided there.

12.4 Notifications

Assuming that there is a temperature change in all of our rooms we will get 3 independent notifications, one for each change.

```
{
  "id": "ngsildbroker:notification:-5983263741316604694",
  "type": "Notification",
  "data": [
    {
      "id": "house2:smartrooms:room1",
      "type": "urn:mytypes:room",
      "createdAt": "2020-08-04T12:55:05.276000Z",
      "modifiedAt": "2020-08-07T13:53:56.781000Z",
      "myuniqueuri:isPartOf": {
        "type": "Relationship",
        "createdAt": "2020-08-04T12:55:05.276000Z",
        "object": "smartcity:houses:house2",
        "modifiedAt": "2020-08-04T12:55:05.276000Z"
      },
      "myuniqueuri:temperature": {
        "type": "Property",
        "createdAt": "2020-08-04T12:55:05.276000Z",
        "providedBy": {
          "type": "Relationship",
          "createdAt": "2020-08-04T12:55:05.276000Z",
          "object": "smartbuilding:house2:sensor0815",
          "modifiedAt": "2020-08-04T12:55:05.276000Z"
        },
        "value": 22.0,
        "modifiedAt": "2020-08-04T12:55:05.276000Z"
      }
    }
  ],
  "notifiedAt": "2020-08-07T13:53:57.640000Z",
  "subscriptionId": "urn:subscription:1"
}
```

```
{
  "id": "ngsildbroker:notification:-6853258236957905295",
  "type": "Notification",
  "data": [
    {
      "id": "house2:smartrooms:room2",
      "type": "urn:mytypes:room",
      "createdAt": "2020-08-04T11:17:28.641000Z",
      "modifiedAt": "2020-08-07T14:00:11.681000Z",
      "myuniqueuri:isPartOf": {
        "type": "Relationship",
        "createdAt": "2020-08-04T11:17:28.641000Z",
        "object": "smartcity:houses:house2",
        "modifiedAt": "2020-08-04T11:17:28.641000Z"
      },
      "myuniqueuri:temperature": {
        "type": "Property",
        "createdAt": "2020-08-04T11:17:28.641000Z",
        "providedBy": {
          "type": "Relationship",
          "createdAt": "2020-08-04T11:17:28.641000Z",
          "object": "smartbuilding:house2:sensor4711",
          "modifiedAt": "2020-08-04T11:17:28.641000Z"
        },
        "value": 23.0,
        "modifiedAt": "2020-08-04T11:17:28.641000Z"
      }
    }
  ],
  "notifiedAt": "2020-08-07T14:00:12.475000Z",
  "subscriptionId": "urn:subscription:1"
}
```

::

```
{ "id": "ngsildbroker:notification:-7761059438747425848", "type": "Notification", "data": [{
  "id": "house99:smartrooms:room42", "type": "urn:mytypes:room", "createdAt":
  "2020-08-04T13:19:17.512000Z", "modifiedAt": "2020-08-07T14:00:19.100000Z",
  "myuniqueuri:isPartOf": {
    "type": "Relationship", "createdAt": "2020-08-04T13:19:17.512000Z", "object":
    "smartcity:houses:house99", "modifiedAt": "2020-08-04T13:19:17.512000Z"
  }, "myuniqueuri:temperature": {
    "type": "Property", "createdAt": "2020-08-04T13:19:17.512000Z", "provid-
    edBy": {
      "type": "Relationship", "createdAt": "2020-08-04T13:19:17.512000Z",
      "object": "smartbuilding:house99:sensor36", "modifiedAt": "2020-08-
      04T13:19:17.512000Z"
    }, "value": 24.0, "modifiedAt": "2020-08-04T13:19:17.512000Z"
  }
}], "notifiedAt": "2020-08-07T14:00:19.897000Z", "subscriptionId": "urn:subscription:1"
```

```
}
```

As you can see we are getting now always the full Entity matching the type we defined in the subscription.

12.5 Subscribing to attributes

An alternative to get the same result in our setup is using the `watchedAttributes` parameter in a subscription.

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↪ld+json' -d @- <<EOF
{
  "id": "urn:subscription:2",
  "type": "Subscription",
  "watchedAttributes": ["temperature"],
  "notification": {
    "endpoint": {
      "uri": "http://ptsv2.com/t/30xad-1596541146/post",
      "accept": "application/json"
    }
  },
  "@context": "https://pastebin.com/raw/Mgxv2ykn"
}
EOF
```

This works in our example but you will get notifications everytime a temperature attribute changes. So in a real life scenario probably much more than we wanted. You need to have at least the `entities` parameter (with a valid entry in the array) or the `watchedAttributes` parameter for a valid subscription. But you can also combine both. So if we want to be notified on every change of “temperature” in a “Room” we subscribe like this.

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↪ld+json' -d @- <<EOF
{
  "id": "urn:subscription:3",
  "type": "Subscription",
  "entities": [{
    "type": "Room"
  }],
  "watchedAttributes": ["temperature"],
  "notification": {
    "endpoint": {
      "uri": "http://ptsv2.com/t/30xad-1596541146/post",
      "accept": "application/json"
    }
  },
  "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF
```

We can now limit further down what we exactly we want to get in the notification very similar to the query.

12.6 IdPattern

As we get now also the “Room” from `smartcity:houses:house99` but we are only interested `smartcity:houses:house2` we will use the `idPattern` parameter to limit the results. This is possible in our case because of our `namestructure`.

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↳ld+json' -d @- <<EOF
{
  "id": "urn:subscription:4",
  "type": "Subscription",
  "entities": [{
    "idPattern" : "house2:smartrooms:room.*",
    "type": "Room"
  }],
  "watchedAttributes": ["temperature"],
  "notification": {
    "endpoint": {
      "uri": "http://ptsv2.com/t/30xad-1596541146/post",
      "accept": "application/json"
    }
  },
  "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF
```

12.7 Q Filter

Similar to our query we can also use the q filter to achieve this via the isPartOf relationship. Mind here in the body there is no URL encoding.

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↳ld+json' -d @- <<EOF
{
  "id": "urn:subscription:5",
  "type": "Subscription",
  "entities": [{
    "type": "Room"
  }],
  "q": "isPartOf==smartcity.houses.house2",
  "watchedAttributes": ["temperature"],
  "notification": {
    "endpoint": {
      "uri": "http://ptsv2.com/t/30xad-1596541146/post",
      "accept": "application/json"
    }
  },
  "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF
```

12.8 Reduce attributes

Now since we still get the full Entity in our notifications we want to reduce the number of attributes. This is done by the attributes parameter in the notification entry.

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↳ld+json' -d @- <<EOF
{
  "id": "urn:subscription:6",
  "type": "Subscription",
  "entities": [{
    "type": "Room"
  }],
  "q": "isPartOf==smartcity.houses.house2",
  "watchedAttributes": ["temperature"],
  "notification": {
    "endpoint": {
      "uri": "http://ptsv2.com/t/30xad-1596541146/post",
      "accept": "application/json"
    },
    "attributes": ["temperature"]
  },
  "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF
```

As you can see, we now only get the temperature when the temperature changes.

```
{
  "id": "ngsildbroker:notification:-7761059438747425848",
  "type": "Notification",
  "data": [
    {
      "id": "house2:smartrooms:room1",
      "type": "urn:mytypes:room",
      "createdAt": "2020-08-04T13:19:17.512000Z",
      "modifiedAt": "2020-08-07T14:30:12.100000Z",
      "myuniqueuri:temperature": {
        "type": "Property",
        "createdAt": "2020-08-04T13:19:17.512000Z",
        "providedBy": {
          "type": "Relationship",
          "createdAt": "2020-08-04T13:19:17.512000Z",
          "object": "smartbuilding:house99:sensor36",
          "modifiedAt": "2020-08-04T13:19:17.512000Z"
        },
        "value": 24.0,
        "modifiedAt": "2020-08-04T13:19:17.512000Z"
      }
    }
  ],
  "notifiedAt": "2020-08-07T14:00:19.897000Z",
  "subscriptionId": "urn:subscription:6"
}
```

The attributes and the watchedAttributes parameter can very well be different. If you want to know in which house a temperature changes you would subscribe like this

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↳ld+json' -d @- <<EOF
{
  "id": "urn:subscription:7",
```

(continues on next page)

(continued from previous page)

```

"type": "Subscription",
"entities": [{
    "type": "Room"
}],
"watchedAttributes": ["temperature"],
"notification": {
    "endpoint": {
        "uri": "http://ptsv2.com/t/30xad-1596541146/post",
        "accept": "application/json"
    },
    "attributes": ["isPartOf"]
},
"@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF

```

12.9 GeoQ filter

An additional filter is the geoQ parameter allowing you to define a geo query. If, for instance, we want to be informed about all Houses near to a point we would subscribe like this.

```

curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
ld+json' -d @- <<EOF
{
  "id": "urn:subscription:8",
  "type": "Subscription",
  "entities": [{
    "type": "House"
  }],
  "geoQ": {
    "georel": {
      "near;maxDistance==2000",
      "geometry": "Point",
      "coordinates": [-8.50000005, 41.20000005]
    },
    "notification": {
      "endpoint": {
        "uri": "http://ptsv2.com/t/30xad-1596541146/post",
        "accept": "application/json"
      },
      "attributes": ["isPartOf"]
    },
    "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
  }
}
EOF

```

12.10 Additional endpoint parameters

The notification entry has two additional optional entries. receiverInfo and notifierInfo. They are both an array of a simple key value set. Practically they represent settings for Scorpios notifier (notifierInfo) and additional headers you want to be sent with every notification (receiverInfo). notifierInfo is currently only used for MQTT. If you want to, for instance, pass on an oauth token you would do a subscription like this

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↳ld+json' -d @- <<EOF
{
  "id": "urn:subscription:9",
  "type": "Subscription",
  "entities": [{
    "type": "Room"
  }],
  "notification": {
    "endpoint": {
      "uri": "http://ptsv2.com/t/30xad-1596541146/post",
      "accept": "application/json",
      "receiverInfo": [{"Authorization": "Bearer_
↳sdckqk3123ykasd723knsws"}]
    }
  },
  "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF
```

12.11 MQTT endpoint

If you have a running MQTT bus available, you can also get notifications to a topic on MQTT. However the setup of the MQTT bus and the creation of the topic is totally outside of the responsibilities of an NGSI-LD broker. An MQTT bus address must be provided via the URI notation of MQTT. `mqtt[s]://[<username>:<password>@]<mqtt_host_name>[:<mqtt_port>]/<topicname>[[/<subtopic>]...]` So a subscription would generally look like this.

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↳ld+json' -d @- <<EOF
{
  "id": "urn:subscription:10",
  "type": "Subscription",
  "entities": [{
    "type": "Room"
  }],
  "notification": {
    "endpoint": {
      "uri": "mqtt://localhost:1883/notifytopic",
      "accept": "application/json"
    }
  },
  "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF
```

12.12 MQTT parameters

MQTT has a few client settings which have to be configured. We do have some reasonable defaults here, if you don't provide it, but to be sure you better configure the client completely. These params are provided via the `notifierInfo` entry in endpoint. Currently supported is "MQTT-Version" with possible values "mqtt3.1.1" or "mqtt5.0", default

“mqtt5.0” “MQTT-QoS” with possible values 0, 1, 2. Default 1. Changing this to version 3.1.1 and QoS to 2 you would subscribe like this

```
curl localhost:9090/ngsi-ld/v1/subscriptions -s -S -H 'Content-Type: application/
↳ ld+json' -d @- <<EOF
{
  "id": "urn:subscription:11",
  "type": "Subscription",
  "entities": [{
    "type": "Room"
  }],
  "notification": {
    "endpoint": {
      "uri": "mqtt://localhost:1883/notifytopic",
      "accept": "application/json",
      "notifierInfo": [{"MQTT-Version": "mqtt3.1.1"}, {"MQTT-QoS": 2}
    ]
  }
},
  "@context": [ "https://pastebin.com/raw/Mgxv2ykn" ]
}
EOF
```

12.13 MQTT notifications

Since MQTT is missing the header that HTTP callbacks have the format of a notification is slightly changed. Consisting of a metadata and a body entry. The metadata holds what is normally delivered via HTTP headers and the body contains the normal notification payload.

```
{
  "metadata": {
    "Content-Type": "application/json"
    "somekey": "somevalue"
  },
  "body": {
    "id": "ngsildbroker:notification:-5983263741316604694
↳ ",
    "type": "Notification",
    "data": [
      {
        "id": "house2:smartrooms:room1",
        "type": "urn:mytypes:room",
        "createdAt": "2020-08-04T12:55:05.
↳ 276000Z",
        "modifiedAt": "2020-08-07T13:53:56.
↳ 781000Z",
        "myuniqueuri:isPartOf": {
          "type": "Relationship",
          "createdAt": "2020-08-
↳ 04T12:55:05.276000Z",
          "object":
↳ "smartcity:houses:house2",
          "modifiedAt": "2020-08-
↳ 04T12:55:05.276000Z"
```

(continues on next page)

(continued from previous page)

```

    },
    "myuniqueuri:temperature": {
      "type": "Property",
      "createdAt": "2020-08-
↪04T12:55:05.276000Z",
      "providedBy": {
        "type": "Relationship
↪",
        "createdAt": "2020-08-
↪04T12:55:05.276000Z",
        "object":
↪"smartbuilding:house2:sensor0815",
        "modifiedAt": "2020-
↪08-04T12:55:05.276000Z"
      },
      "value": 22.0,
      "modifiedAt": "2020-08-
↪04T12:55:05.276000Z"
    }
  },
  "notifiedAt": "2020-08-07T13:53:57.640000Z",
  "subscriptionId": "urn:subscription:1"
}

```

Batch operations

NGSI-LD defines 4 endpoints for 4 batch operations. You can create a batch of Entity creations, updates, upserts or deletes. Create, update and upsert are basically an array of the corresponding single Entity operations. Assuming we want to create a few rooms for house 99 we would create the entities like this

```
curl localhost:9090/ngsi-ld/v1/entityOperations/create -s -S -H 'Content-Type: application/ld+json' -d @- <<EOF
[{
    "id": "house99:smartrooms:room1",
    "type": "Room",

    "isPartOf": {
        "type": "Relationship",
        "object": "smartcity:houses:house99"
    },
    "@context": "https://pastebin.com/raw/Mgxv2ykn"
},
{
    "id": "house99:smartrooms:room2",
    "type": "Room",
    "isPartOf": {
        "type": "Relationship",
        "object": "smartcity:houses:house99"
    },
    "@context": "https://pastebin.com/raw/Mgxv2ykn"
},
{
    "id": "house99:smartrooms:room3",
    "type": "Room",
    "isPartOf": {
        "type": "Relationship",
        "object": "smartcity:houses:house99"
    },
}
```

(continues on next page)

(continued from previous page)

```

        "@context": "https://pastebin.com/raw/Mgxv2ykn"
    },
    {
        "id": "house99:smartrooms:room4",
        "type": "Room",
        "temperature": {
            "value": 21,
            "unitCode": "CEL",
            "type": "Property",
            "providedBy": {
                "type": "Relationship",
                "object": "smartbuilding:house99:sensor20041113"
            }
        },
        "isPartOf": {
            "type": "Relationship",
            "object": "smartcity:houses:house99"
        },
        "@context": "https://pastebin.com/raw/Mgxv2ykn"
    }
]
EOF

```

Now as we did only add one temperature entry we are going to upsert the temperature for all the rooms like this.

```

curl localhost:9090/ngsi-ld/v1/entityOperations/upsert -s -S -H 'Content-Type:
↪application/ld+json' -d @- <<EOF
[
    {
        "id": "house99:smartrooms:room1",
        "type": "Room",
        "temperature": {
            "value": 22,
            "unitCode": "CEL",
            "type": "Property",
            "providedBy": {
                "type": "Relationship",
                "object": "smartbuilding:house99:sensor19970309"
            }
        },
        "@context": "https://pastebin.com/raw/Mgxv2ykn"
    },
    {
        "id": "house99:smartrooms:room2",
        "type": "Room",
        "temperature": {
            "value": 23,
            "unitCode": "CEL",
            "type": "Property",
            "providedBy": {
                "type": "Relationship",
                "object": "smartbuilding:house99:sensor19960913"
            }
        },
        "@context": "https://pastebin.com/raw/Mgxv2ykn"
    }
]

```

(continues on next page)

(continued from previous page)

```
    },
    {
      "id": "house99:smartrooms:room3",
      "type": "Room",
      "temperature": {
        "value": 21,
        "unitCode": "CEL",
        "type": "Property",
        "providedBy": {
          "type": "Relationship",
          "object": "smartbuilding:house99:sensor19931109"
        }
      },
      "@context": "https://pastebin.com/raw/Mgxv2ykn"
    },
    {
      "id": "house99:smartrooms:room4",
      "type": "Room",
      "temperature": {
        "value": 22,
        "unitCode": "CEL",
        "type": "Property",
        "providedBy": {
          "type": "Relationship",
          "object": "smartbuilding:house99:sensor20041113"
        }
      },
      "@context": "https://pastebin.com/raw/Mgxv2ykn"
    }
  ]
EOF
```

Now as we are at the end let's clean up with a batch delete. A batch delete is an array of Entity IDs you want to delete.

```
curl localhost:9090/ngsi-ld/v1/entityOperations/delete -s -S -H 'Content-Type:↵
↵application/json' -d @- <<EOF
[
  "house99:smartrooms:room1",
  "house99:smartrooms:room2",
  "house99:smartrooms:room3",
  "house99:smartrooms:room4"
]
EOF
```


CHAPTER 14

Context Registry

Next to the create, append, update interfaces which are used by Context Producers there is another concept in NGSI-LD which is the Context Source. A Context Source is a source that provides the query and the subscription interface of NGSI-LD. For all intents and purposes an NGSI-LD Broker is by itself an NGSI-LD Context Source. This allows you a lot of flexibility when you want to have distributed setup. Now in order to discover these Context Sources, the Context Registry is used, where Context Sources are registered in Scorpio. Assuming we have an external Context Source which provides information about another house, we register it in the system like this:

```
{
  "id": "urn:ngsi-ld:ContextSourceRegistration:csrla3458",
  "type": "ContextSourceRegistration",
  "information": [
    {
      "entities": [
        {
          "type": "Room"
        }
      ]
    }
  ],
  "endpoint": "http://my.csource.org:1234",
  "location": { "type": "Polygon", "coordinates": [[[8.686752319335938, 49.
→ 359122687528746], [8.742027282714844, 49.3642654834877], [8.767433166503904, 49.
→ 398462568451485], [8.768119812011719, 49.42750021620163], [8.74305725097656, 49.
→ 44781634951542], [8.669242858886719, 49.43754770762113], [8.63525390625, 49.
→ 41968407776289], [8.637657165527344, 49.3995797187007], [8.663749694824219, 49.
→ 36851347448498], [8.686752319335938, 49.359122687528746]]] } },
  "@context": "https://pastebin.com/raw/Mgxv2ykn"
}
```

Now Scorpio will take the registered Context Sources which are have a matching registration into account on its queries and subscriptions. You can also independently query or subscribe to the context registry entries, quite similar to the normal query or subscription, and interact with the Context Sources independently. Now if we query for all registrations which provide anything of type Room like this

```
curl localhost:9090/ngsi-ld/v1/csourceRegistrations/?type=Room -s -S -H 'Accept:↵
↵application/json' -H 'Link: <https://pastebin.com/raw/Mgxv2ykn>; rel="http://www.w3.
↵org/ns/json-ld#context"; type="application/ld+json"'
```

we will get back our original registration and everything that has been registered with the type Room.

14.1 Context Registry usage for normal queries & subscriptions

A context registry entry can have multiple entries which are taken into consideration when normal queries or subscriptions arrive in Scorpio. As you can see there is an entities entry similar to the one in the subscriptions. This is the first thing to be taken into consideration. If you register a type, Scorpio will only forward a request which is matching that type. Similarly the location is used to decide if a query with geo query part should be forwarded. While you shouldn't overdo it, the more details you provide in a registration the more efficiently your system will be able to determine to which context source a request should be forwarded to. Below you see an example with more properties set.

```
{
  "id": "urn:ngsi-ld:ContextSourceRegistration:csrla3459",
  "type": "ContextSourceRegistration",
  "name": "NameExample",
  "description": "DescriptionExample",
  "information": [
    {
      "entities": [
        {
          "type": "Vehicle"
        }
      ],
      "properties": [
        "brandName",
        "speed"
      ],
      "relationships": [
        "isParked"
      ]
    },
    {
      "entities": [
        {
          "idPattern": ".*downtown$",
          "type": "OffStreetParking"
        }
      ]
    }
  ],
  "endpoint": "http://my.csource.org:1026",
  "location": "{ \"type\": \"Polygon\", \"coordinates\": [[ [8.686752319335938, 49.
↵359122687528746], [8.742027282714844, 49.3642654834877], [8.767433166503904, 49.
↵398462568451485], [8.768119812011719, 49.42750021620163], [8.74305725097656, 49.
↵44781634951542], [8.669242858886719, 49.43754770762113], [8.63525390625, 49.
↵41968407776289], [8.637657165527344, 49.3995797187007], [8.663749694824219, 49.
↵36851347448498], [8.686752319335938, 49.359122687528746]] ] }"
```

There are two entries in the information part. In the first you can see there are two additional entries describing the two properties and one relationship provided by that source. That means any query which asks for type Vehicle, without

an attribute filter, will be forwarded to this source and if there is an attribute filter it will only be forwarded if the registered properties or relationships match. The second entry means that this source can provide Entities of type `OffStreetParking`, which have an Entity ID ending with “downtown”.

In order to set-up the environment of Scorpio broker, the following dependency needs to be configured:-

1. Eclipse.
2. Server JDK.
3. Apache Kafka.
4. PostgreSQL

15.1 Windows

15.1.1 Eclipse installation

- **Download the Eclipse Installer.:**

Download Eclipse Installer from <http://www.eclipse.org/downloads>. Eclipse is hosted on many mirrors around the world. Please select the one closest to you and start to download the Installer.

- **Start the Eclipse Installer executable:**

For Windows users, after the Eclipse Installer, the executable has finished downloading it should be available in your download directory. Start the Eclipse Installer executable. You may get a security warning to run this file. If the Eclipse Foundation is the Publisher, you are good to select Run.

For Mac and Linux users, you will still need to unzip the download to create the Installer. Start the Installer once it is available.

- **Select the package to install:**

The new Eclipse Installer shows the packages available to Eclipse users. You can search for the package you want to install or scroll through the list. Select and click on the package you want to install.

- **Select your installation folder**

Specify the folder where you want Eclipse to be installed. The default folder will be in your User directory. Select the 'Install' button to begin the installation.

- **Launch Eclipse**

Once the installation is complete you can now launch Eclipse. The Eclipse Installer has done its work. Happy coding.

15.1.2 JDK Setup

- Start the JDK installation and hit the "Change destination folder" checkbox, then click 'Install.'

Note:- Recommended version is JDK-11. Scorpio Broker is developed and tested with this version only.



- Change the installation directory to any path without spaces in the folder name.

After you've installed Java in Windows, you must set the `JAVA_HOME` environment variable to point to the Java installation directory.

Set the `JAVA_HOME` Variable

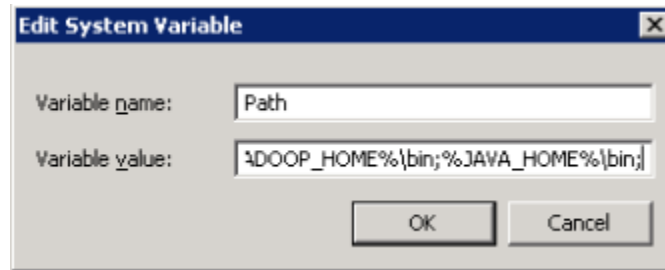
To set the `JAVA_HOME` variable:

1. Find out where Java is installed. If you didn't change the path during installation, it will be something like this:
C:\Program Files\Java\jdk1.version_detail
2.
 - In Windows 8/10 go to **Control Panel > System > Advanced System Settings**.
 - OR
 - In Windows 7 right-click **My Computer** and select **Properties > Advanced**.
3. Click the Environment Variables button.
4. Under System Variables, click New.
5. In the User Variable Name field, enter: **JAVA_HOME**

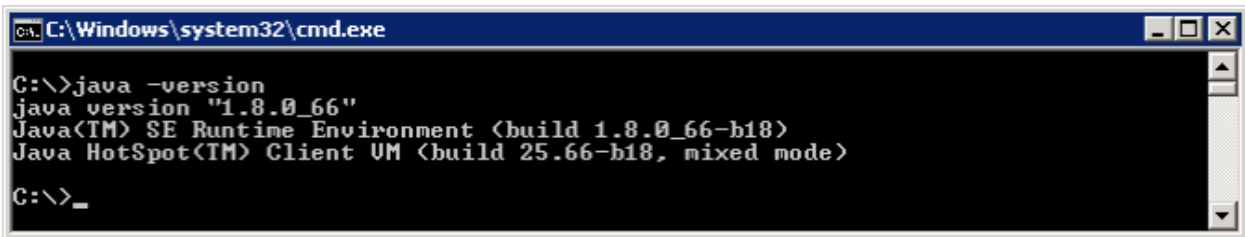
6. In the User Variable Value field, enter your JDK path.

(Java path and version may change according to the version of Kafka you are using)

7. Now click OK.
8. Search for a Path variable in the “System Variable” section in the “Environment Variables” dialogue box you just opened.
9. Edit the path and type `;%JAVA_HOME%bin` at the end of the text already written there, just like the image below:



- To confirm the Java installation, just open cmd and type “java -version.” You should be able to see the version of Java you just installed.



If your command prompt somewhat looks like the image above, you are good to go. Otherwise, you need to recheck whether your setup version matches the correct OS architecture (x86, x64), or if the environment variables path is correct.

15.1.3 Setting Up Kafka

1. Go to your Kafka config directory. For example:- **C:kafka_2.11-0.9.0.0config**
2. Edit the file “server.properties.”
3. Find and edit the line `log.dirs=/tmp/kafka-logs` to `log.dir= C:kafka_2.11-0.9.0.0kafka-logs`.
4. If your ZooKeeper is running on some other machine or cluster you can edit “`zookeeper.connect:2181`” to your custom IP and port. For this demo, we are using the same machine so there’s no need to change. Also the Kafka port and broker.id are configurable in this file. Leave other settings as is.
5. Your Kafka will run on default port 9092 and connect to ZooKeeper’s default port, 2181.

Note: For running Kafka, zookeepers should run first. At the time of closing Kafka, zookeeper should be closed first than Kafka. Recommended version of kafka is `kafka_2.12-2.1.0`.

15.1.4 Running a Kafka Server

Important: Please ensure that your ZooKeeper instance is up and running before starting a Kafka server.

1. Go to your Kafka installation directory:** C:kafka_2.11-0.9.0.0**
2. Open a command prompt here by pressing Shift + right-click and choose the “Open command window here” option).
3. Now type **.bin\windowskafka-server-start.bat .config\server.properties** and press Enter,then
4. Type **.bin\windowskafka-server-start.bat .config\server.properties** in new command window and hit enter.

15.1.5 Setting up PostgreSQL

Step 1) Go to <https://www.postgresql.org/download>.

Step 2) You are given two options:-

1. Interactive Installer by EnterpriseDB
2. Graphical Installer by BigSQL

BigSQL currently installs pgAdmin version 3 which is deprecated. It's best to choose EnterpriseDB which installs the latest version 4

Step 3)

1. You will be prompted to the desired Postgre version and operating system. Select the Postgres 10, as Scorpio has been tested and developed with this version.

2. Click the Download Button, Download will begin

Step 4) Open the downloaded .exe and Click next on the install welcome screen.

Step 5)

1. Change the Installation directory if required, else leave it to default
2. Click Next

Step 6)

1. You can choose the components you want to install in your system. You may uncheck Stack Builder
2. Click on Next

Step 7)

1. You can change the data location
2. Click Next

Step 8)

1. Enter the superuser password. Make a note of it
2. Click Next

Step 9)

1. Leave the port number as the default
2. Click Next

Step 10)

1. Check the pre-installation summary.
2. Click Next

Step 11) Click the next button

Step 12) Once install is complete you will see the Stack Builder prompt

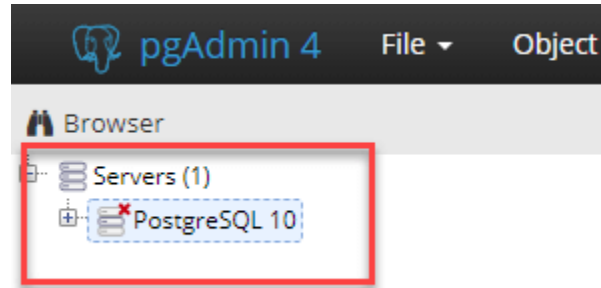
1. Uncheck that option. We will use Stack Builder in more advance tutorials

2. Click Finish

Step 13) To launch Postgre go to Start Menu and search pgAdmin 4

Step 14) You will see pgAdmin homepage

Step 15) Click on Servers > Postgre SQL 10 in the left tree

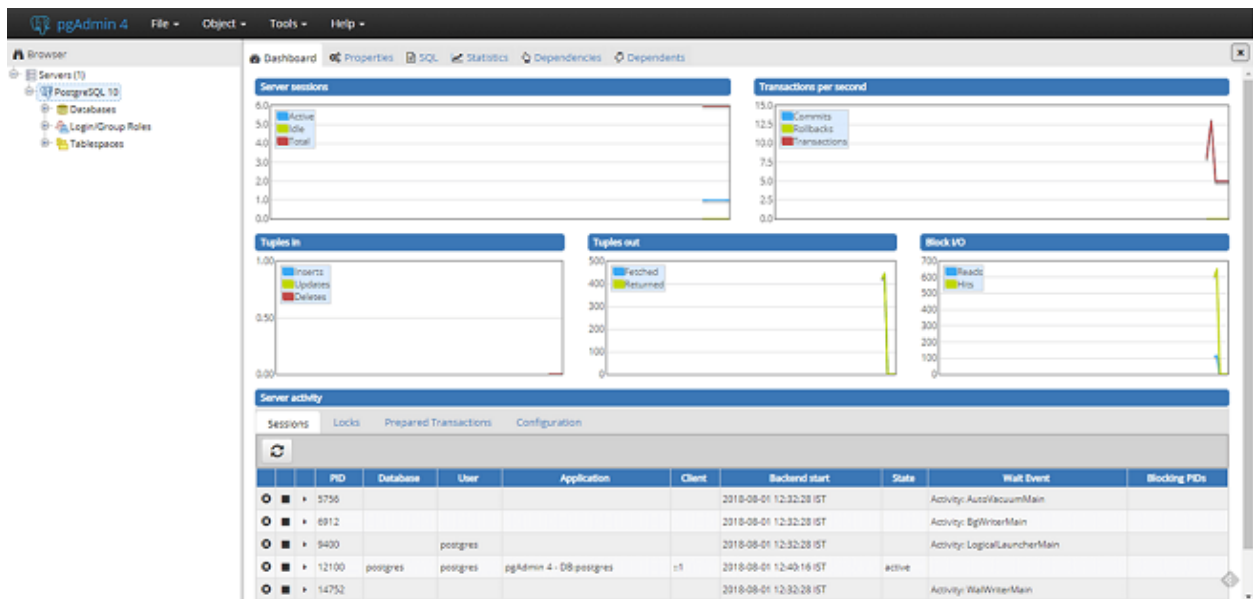


Step 16)

1. Enter superuser password set during installation

2. Click OK

Step 17) You will see the Dashboard



That's it to Postgre SQL installation.

15.2 Linux

15.2.1 JDK Setup

To create a Java environment in your machine install the JDK, for this open the terminal, and run the following commands:-

1. `sudo apt-get update`
2. `sudo apt-get install openjdk-8-jdk`

To check that JDK is properly installed in your machine, run the command **java -version** in your terminal if it returns the version of the JDK as 11 then it's working fine.

15.2.2 Eclipse installation

To install the eclipse in your linux machine first, visit the link <https://www.eclipse.org/downloads/> and select the version of eclipse based on the flavor of your linux machine.

15.2.3 Setting Up Kafka

To download the Apache Kafka in your machine run the following commands one by one in your terminal.

1. `mkdir kafka`
2. `cd kafka`
3. `wget https://archive.apache.org/dist/kafka/2.2.0/kafka_2.12-2.2.0.tgz`
4. `tar -xzf kafka_2.12-2.2.0.tgz`

Once the Kafka is downloaded in your machine hit the following commands to get it run

1. `kafka_2.12-2.2.0/bin/zookeeper-server-start.sh kafka_2.12-2.2.0/config/zookeeper.properties > /dev/null 2>&1 &`
2. `kafka_2.12-2.2.0/bin/kafka-server-start.sh kafka_2.12-2.2.0/config/server.properties > /dev/null 2>&1 &`

15.2.4 Setting up PostgreSQL

In order to download the PostgreSQL in your machine run the following commands from your terminal.

1. `sudo apt update`
2. `sudo apt-get install postgresql-10`
3. `service postgresql status`

The last command will give us the status of the PostgreSQL four your machine if this matches to one in the picture then everything is properly installed else re-run the commands. .. figure:: figures/postgresTerminal

Once PostgreSQL is successfully installed in your machine create the database **ngb** and change its role by running the following commands:

1. `psql -U postgres -c "create database ngb;"`
2. `psql -U postgres -c "create user ngb with password 'ngb';"`

3. `psql -U postgres -c "alter database ngb owner to ngb;"`
 4. `psql -U postgres -c "grant all privileges on database ngb to ngb;"`
 5. `psql -U postgres -c "alter role ngb superuser;"`
 6. `sudo apt install postgresql-10-postgis-2.4`
 7. `sudo apt install postgresql-10-postgis-scripts`
 8. `sudo -u postgres psql -U postgres -c "create extension postgis;"`
- After this your PostgreSQL is ready to use for Scorpio Boker.

16.1 Java 8 System Requirements

Windows

- Windows 10 (8u51 and above)
- Windows 8.x (Desktop)
- Windows 7 SP1
- Windows Vista SP2
- Windows Server 2008 R2 SP1 (64-bit)
- Windows Server 2012 and 2012 R2 (64-bit)
- RAM: 128 MB
- Disk space: 124 MB for JRE; 2 MB for Java Update
- Processor: Minimum Pentium 2 266 MHz processor
- Browsers: Internet Explorer 9 and above, Firefox

Mac OS X

- Intel-based Mac running Mac OS X 10.8.3+, 10.9+
- Administrator privileges for installation
- 64-bit browser
- A 64-bit browser (Safari, for example) is required to run Oracle Java on Mac.

Linux

- Oracle Linux 5.5+1
- Oracle Linux 6.x (32-bit), 6.x (64-bit)2
- Oracle Linux 7.x (64-bit)2 (8u20 and above)

- Red Hat Enterprise Linux 5.5+1, 6.x (32-bit), 6.x (64-bit)2
- Red Hat Enterprise Linux 7.x (64-bit)2 (8u20 and above)
- Suse Linux Enterprise Server 10 SP2+, 11.x
- Suse Linux Enterprise Server 12.x (64-bit)2 (8u31 and above)
- Ubuntu Linux 12.04 LTS, 13.x
- Ubuntu Linux 14.x (8u25 and above)
- Ubuntu Linux 15.04 (8u45 and above)
- Ubuntu Linux 15.10 (8u65 and above)
- Browsers: Firefox

16.2 ZooKeeper Requirements

ZooKeeper runs in Java, release 1.6 or greater (JDK 6 or greater). It runs as an ensemble of ZooKeeper servers. Three ZooKeeper servers are the minimum recommended size for an ensemble, and we also recommend that they run on separate machines. At Yahoo!, ZooKeeper is usually deployed on dedicated RHEL boxes, with dual-core processors, 2GB of RAM, and 80GB IDE hard drives.

16.3 Recommendations for Kafka

Kafka brokers use both the JVM heap and the OS page cache. The JVM heap is used for the replication of partitions between brokers and for log compaction. Replication requires 1MB (default `replica.max.fetch.size`) for each partition on the broker. In Apache Kafka 0.10.1 (Confluent Platform 3.1), we added a new configuration (`replica.fetch.response.max.bytes`) that limits the total RAM used for replication to 10MB, to avoid memory and garbage collection issues when the number of partitions on a broker is high. For log compaction, calculating the required memory is more complicated and we recommend referring to the Kafka documentation if you are using this feature. For small to medium-sized deployments, 4GB heap size is usually sufficient. In addition, it is highly recommended that consumers always read from memory, i.e. from data that was written to Kafka and is still stored in the OS page cache. The amount of memory this requires depends on the rate at which this data is written and how far behind you expect consumers to get. If you write 20GB per hour per broker and you allow brokers to fall 3 hours behind in normal scenario, you will want to reserve 60GB to the OS page cache. In cases where consumers are forced to read from disk, performance will drop significantly

Kafka Connect itself does not use much memory, but some connectors buffer data internally for efficiency. If you run multiple connectors that use buffering, you will want to increase the JVM heap size to 1GB or higher.

Consumers use at least 2MB per consumer and up to 64MB in cases of large responses from brokers (typical for bursty traffic). Producers will have a buffer of 64MB each. Start by allocating 1GB RAM and add 64MB for each producer and 16MB for each consumer planned.

CHAPTER 17

Error Handling in Scorpio

This section will provide info on the error handling mechanism for the Scorpio Broker system.

Listed below are the events of the system

Table 1: Error Handling

| S.No. | Operation/Event | Scenario Description | Responsible Module | Error Code/ Re-sponse | Action |
|-------|-----------------------|--|--------------------|-----------------------|--------------------------------------|
| 1. | InvalidRequest | The request associated to the operation is syntactically invalid or includes wrong content | REST Controller | HTTP 400 | Log the error & notify the requestor |
| 2. | BadRequestData | The request includes input data which does not meet the requirements of the operation | REST Controller | HTTP 400 | Log the error & notify the requestor |
| 3. | AlreadyExists | The referred element already exists | REST Controller | HTTP 409 | Log the error & notify the requestor |
| 4. | OperationNotSupported | The operation is not supported | REST Controller | HTTP 422 | Log the error & notify the requestor |
| 5. | ResourceNotFound | The referred resource has not been found | REST Controller | HTTP 404 | Log the error & notify the requestor |
| 6. | InternalServerError | There has been an error during the operation execution | REST Controller | HTTP 500 | Log the error & notify the requestor |
| 7. | Method Not Allowed | There has been an error when a client invokes a wrong HTTP verb over a resource | REST Controller | HTTP 405 | Log the error & notify the requestor |

Please note the errors can also be categorized into following categories for different exceptions that can occur internally to the implementation logic as well:

1. Low criticality is those which involve the errors that should be handled by the software logic, and are due to some configuration issues and should not require anything like reset, a reboot of the system.
2. Medium Criticality is those which will be tried for the software logic handling but it may need system reset, chip reset and may interrupt system significantly.
3. High Criticality is the hardware-based error that should not occur and if occur may need system reset.

Fail-safe mechanisms for the different category of errors:

- a. For the Low criticality of the errors, logging will be performed, the retry will be performed and error will be handled by means of rollback and sending failure to the upper layers.
- b. For the High Criticality errors, emergency errors will be logged further recommending a reboot.
- c. For the Medium criticality errors logging, retry mechanisms will be implemented further logging emergency logs to the system and recommend a reboot to the administrator.

During the initialization, failure will be logged as emergency and error will be returned to the calling program

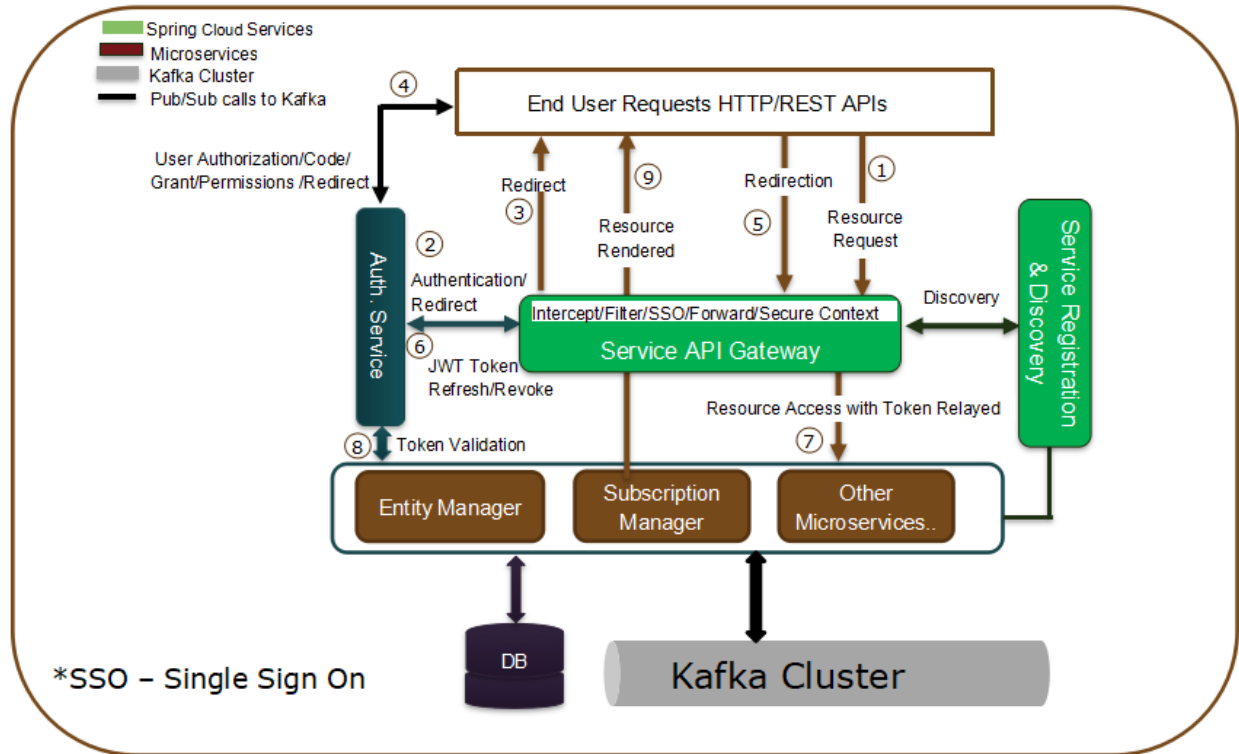
18.1 Security Architecture

Scorpio Broker system will also be responsible for any of the Identity & authentication management security. This will include authentication & authorization of requests, users, role base protected resources to access in Scorpio Broker security realm.

A new Authentication & Authorization service compliant to OAuth2.0 specs has been introduced that will provide the application layer security to the entire Scorpio Broker components & services.

18.2 Security - Functional Request Flow

1. Browser/end user sends a resource request which is protected to the Scorpio Broker system using the API gateway REST interface.
2. API Gateway checks if the security feature is enabled.
 - a. If yes then, it checks if the request is already authenticated and already has some existing session.
 - If it does not find any session, then it forwards the request to Authentication & Authorization services. Or
 - If it finds any existing session than it reuses the same session for the authentication purpose and routes the request to the back-end resource service.
 - b. If no security is enabled then, it bypasses security check and routes the request to the back-end resource service which is responsible to render the resource against the given request.
3. Now when the request comes at Authentication & Authorization (Auth in short) service, it responds to the original requester i.e. user/browser with a login form to present their identity based on credentials it has been issued to access the resource.
4. So now the user submits the login form with its credential to Auth service. Auth services validate the user credentials based on its Account details and now responded with successful login auth code & also the redirect U to which the user can redirect to fetch its resource request.



5. User/Browser now redirects at the redirect URL which is in our case is again the API gateway URL with the auth_code that it has received from the Auth service.
6. Now API gateway again checks the session where it finds the existing session context but now this time since it receives the auth_code in the request so it uses that auth_code and requests the token from Auth service acting as a client on user's behalf. Auth service based on auth code recognized that it is already logged-in validated user and reverts back with the access token to the API gateway.
7. The API gateway upon receiving the token (with in the same security session context), now relays/routes to the back-end resource service for the original requested resource/operation.
8. The back-end resource service is also enabled with security features (if not error will be thrown for the coming secure request). It receives the request and reads the security context out of it and now validates (based on some extracted info) the same with the Auth service to know if this is a valid token/request with the given privileges. Auth service response backs and back-end service decide now whether the local security configuration and the auth service-based access permissions are matching.
9. If the access permissions/privileges are matched for the incoming request, then it responds back with the requested resources to the user/browser. In case, it does not match the security criteria than it reverts with the error message and reason why it's being denied.

CHAPTER 19

Hello World example

Generally speaking you can Create entities which is like the hello world program for Scorpio Broker by sending an HTTP POST request to `http://localhost:9090/ngsi-ld/v1/entities/` with a payload like this

```
curl localhost:9090/ngsi-ld/v1/entities -s -S -H 'Content-Type: application/json' -d@-
{
  "id": "urn:ngsi-ld:testunit:123",
  "type": "AirQualityObserved",
  "dateObserved": {
    "type": "Property",
    "value": {
      "@type": "DateTime",
      "@value": "2018-08-07T12:00:00Z"
    }
  },
  "NO2": {
    "type": "Property",
    "value": 22,
    "unitCode": "GP",
    "accuracy": {
      "type": "Property",
      "value": 0.95
    }
  },
  "refPointOfInterest": {
    "type": "Relationship",
    "object": "urn:ngsi-ld:PointOfInterest:RZ:MainSquare"
  },
  "@context": [
    "https://schema.lab.fiware.org/ld/context",
    "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
  ]
}
```

In the given example the @context is in the payload therefore you have to set the ContentType header to applica-

tion/ld+json

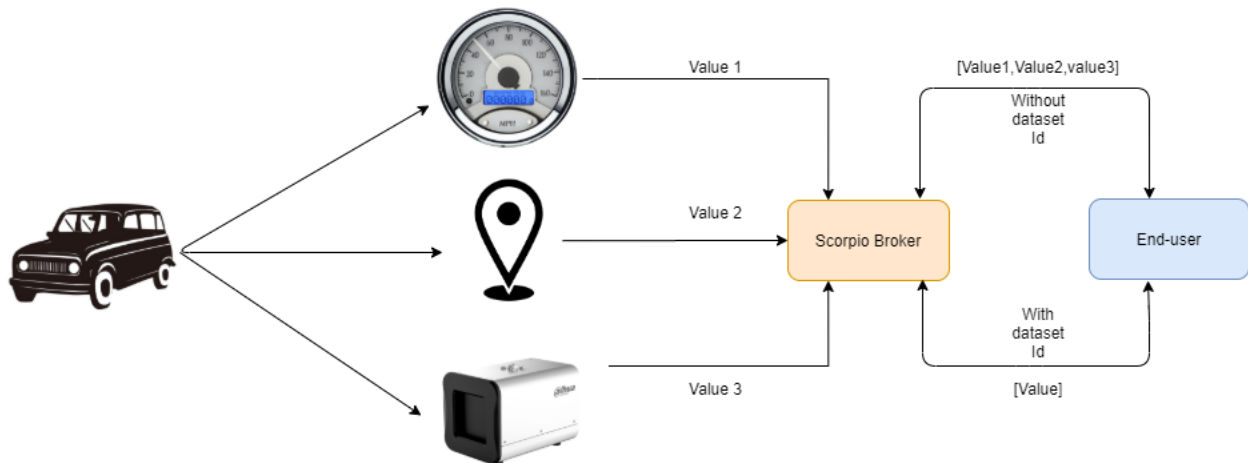
To receive entities you can send an HTTP GET to

`http://localhost:9090/ngsi-ld/v1/entities/<entityId>`

Multi-value Attribute

Multi-value Attribute is a feature through which an Entity can simultaneously have Attributes with more than one instance. In the case of Properties, there may be more than one source at a time that provides a Property value, e.g. based on independent sensor measurements with different quality characteristics.

For example: take a speedometer and a GPS both providing the current speed of a car or take a thermometer or an infrared camera both provides the temperature of the body.



In the case of Relationships, there may be non-functional Relationships, e.g. for a room, there may be multiple “contains” Relationships to all sorts of objects currently in the room that have been put there by different people and which are dynamically changing over time. To be able to explicitly manage such multi-attributes, the optional `datasetId` property is used, which is of datatype `URI`.

20.1 CRUD Operations

If a `datasetId` is provided when creating, updating, appending or deleting Attributes, only instances with the same `datasetId` are affected, leaving instances with another `datasetId` or an instance without a `datasetId` untouched. If no `datasetId` is provided, it is considered as the default Attribute instance. It is not required to explicitly provide this default `datasetId`, but even if not present it is treated as if this default `datasetId` was present in the request(s). Thus the creation, updating, appending or deleting of Attributes without providing a `datasetId` only affects the default property instance.

Note:-There can only be one default Attribute instance for an Attribute with a given Attribute Name in any request or response.

When requesting Entity information, if there are multiple instances of matching Attributes these are returned as arrays of Attributes respectively, instead of a single Attribute element. The `datasetId` of the default Attribute instance is never explicitly included in responses. In case of conflicting information for an Attribute, where a `datasetId` is duplicated, but there are differences in the other attribute data, the one with the most recent `observedAt DateTime`, if present, and otherwise the one with the most recent `modifiedAt DateTime` shall be provided.

20.1.1 1. Create Operation

In order to create the entity with the multi-value attribute, we can hit the endpoint **`http://<IP Address>:<port>/ngsi-ld/v1/entities/`** with the given payload.

```
{
  "id": "urn:ngsi-ld:Vehicle:A135",
  "type": "Vehicle",
  "brandName": {
    "type": "Property",
    "value": "Mercedes"
  },
  "speed": [{
    "type": "Property",
    "value": 55,
    "datasetId": "urn:ngsi-ld:Property:speedometerA4567-speed",
    "source": {
      "type": "Property",
      "value": "Speedometer"
    }
  }],
  {
    "type": "Property",
    "value": 11,
    "datasetId": "urn:ngsi-ld:Property:gpsA4567-speed",
    "source": {
      "type": "Property",
      "value": "GPS"
    }
  },
  {
    "type": "Property",
    "value": 10,
    "source": {
      "type": "Property",
      "value": "CAMERA"
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
}]
}
```

20.1.2 2. Update Operation

- **Update the attribute instance value based on datasetId**

We can update the value of the particular instance by sending the datasetId in the body and making the PATCH request at **http://<IP Address>:<port>/ngsi-ld/v1/entities/entityId/attrs/attrsId**

```
{
  "value": "27",
  "datasetId": "urn:ngsi-ld:Property:speedometerA4567-speed"
}
```

- **Update the default attribute instance value based on attribute name**

We can update the value of the default instance by making the PATCH request at **http://<IP Address>:<port>/ngsi-ld/v1/entities/entityId/attrs/attrsId** with only updated value in the payload.

```
{
  "value": "27"
}
```

20.1.3 3. Delete Operation

- **Delete the default attribute instance**

In order to delete the default attribute instance, make the DELETE request with URL **http://<IP Address>:<port>/ngsi-ld/v1/entities/entityId/attrs/attrsId** this will delete the default instance of the attribute.

- **Delete the attribute instance with datasetId**

To delete the particular attribute instance, make a DELETE request with URL **http://<IP Address>:<port>/ngsi-ld/v1/entities/entityId/attrs/attrsId?datasetId={{datasetId}}** where datasetId is the id of the instance which we require to be deleted.

- **Delete all the attribute instance with the given attribute name**

If we want to delete all the attribute instance with the given attribute name, then we need to make DELETE request with the URL **http://<IP Address>:<port>/ngsi-ld/v1/entities/entityId/attrs/attrsId?deleteAll=true**.

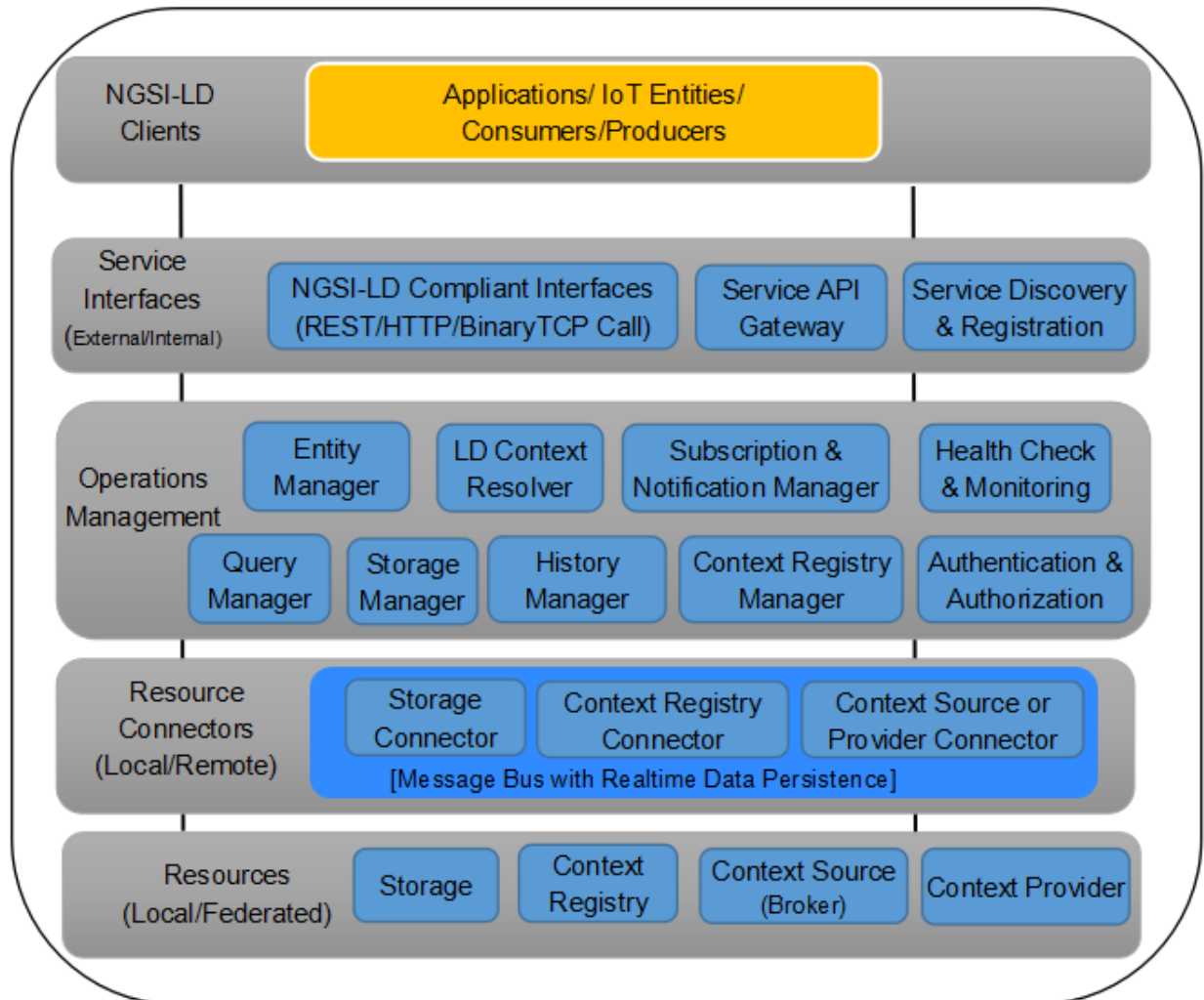
20.1.4 4. Query Operation

In order to retrieve the entity details, make a GET request with URL **http://<IP Address>:<port>/ngsi-ld/v1/entities/**, and we will get all the instance of the required attribute.

The deployment architecture leverages the Spring Cloud framework that addresses lots of Micro-services concerns(e.g. scaling, monitoring, fault-tolerant, highly available, secure, decoupled, etc.) and Kafka based distributed and scalable message queue infrastructure to provide high performance on message processing for a huge number of context requests which is usual in the IoT domain.

It covers the high-level operations (HTTP based REST with method POST/GET/DELETE/PATCH) request flow from the external world to the Scorpio Broker system. The external request is served through a unified service API gateway interface that exposes a single IP/port combination to be used for all services that the Scorpio Broker system can provide. In reality, each of the Scorpio Broker services have been implemented as a micro-service that can be deployed as an independent standalone unit in a distributed computing environment. The API gateway routes all the incoming requests to the specific Micro-services with the help of THE registration & discovery service. Once the request reaches a micro-service based on the operation requirement it uses(pub/sub) Kafka topics (message queues) for real-time storage and for providing intercommunication among different micro-services (based on requirement) over message queues.

- **Application:** End-user/domain applications leverage Scorpio Broker to provide the required information about IoT infrastructure. This application can query, subscribe, update context information to/from the Scorpio Broker as per their requirements.
- **Consumers:** These are the IoT entities or applications that consume the data of Scorpio Broker.
- **Producers:** These are the IoT entities, context source, or applications that produce the context data to the Scorpio Broker.
- **Service API Gateway:** This is the proxy gateway for the external world to access the internal services of the Scorpio Broker system exposed via REST-based HTTP interfaces. All internal Scorpio Broker related services can be accessed through this service gateway using its single IP & port (which are usually static) and extending the service name in the URL. Thus the user does not need to take care of (or learn or use) the IP and Port of every service which often changes dynamically. This makes life easier, especially in a case when multiple services (or micro-service) are running under one system. This is easily solved by the use of proxy gateway(i.e. service API gateway) for all the back-end services.
- **Rest Interface:** These are the HTTP based interfaces for the external entities/applications to consume in order to execute certain operations on Scorpio Broker. The external interface would be visible through the Service



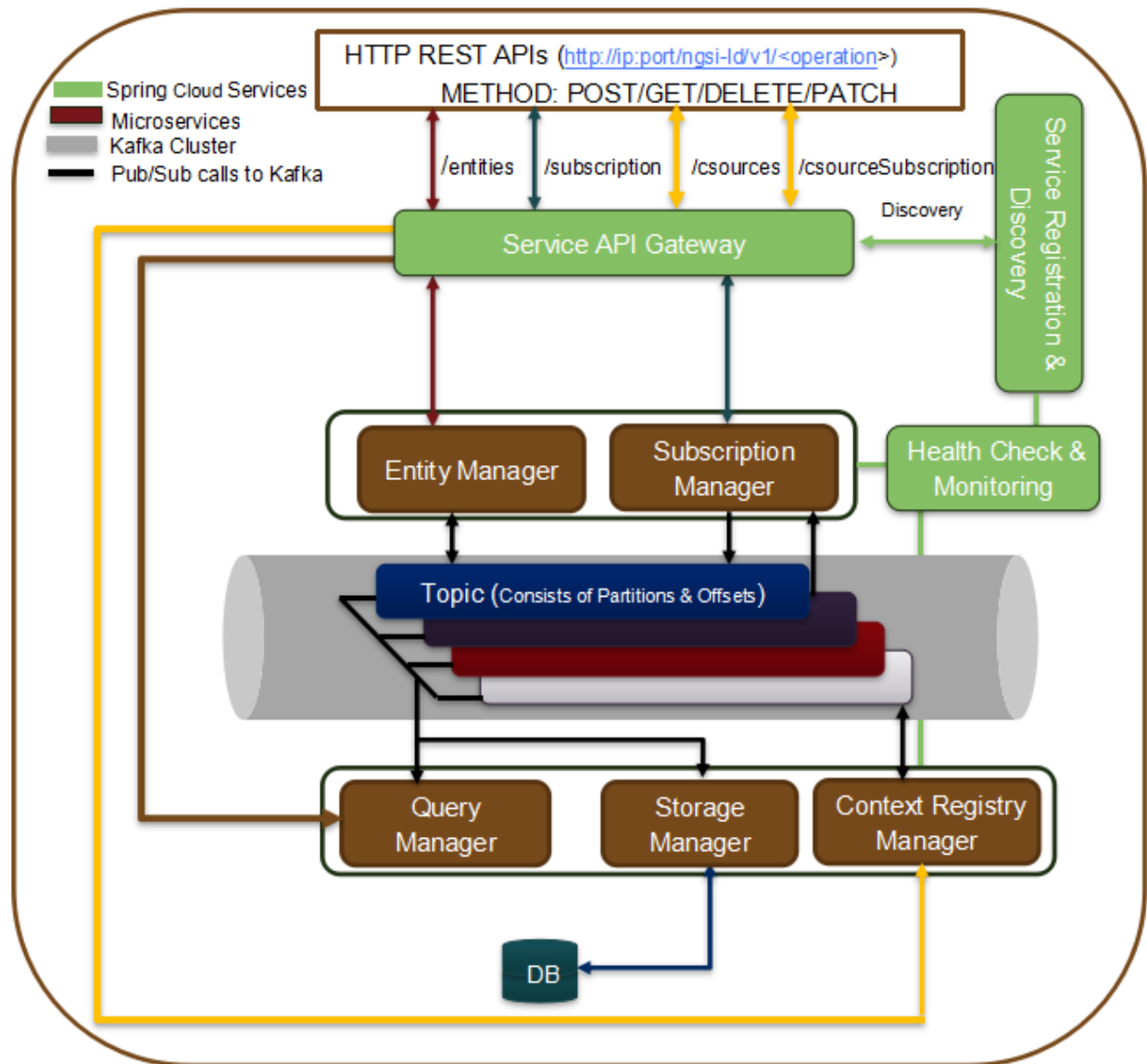
API gateway and internal interface mapping to each requested service would be discovered through the service registration & discovery module.

- **Service Discovery & Registration:** This component allows registration of any service (web service/micro-service) with it so that any client using discovery functionality of this component can determine the location of a service instance to which it wants to send requests. So in short, a service registry & discovery implements a database of services, their instances, and their locations. Service instances get registered with the service registry on startup and deregistered on shutdown. A client of the service, query the service registry, which discovers the available instances of a service. A service registry might also invoke a service instance's health check API to verify that it is able to handle requests.
- **Entity Manager:** This component handles all entity related CRUD operations with the help of other components of the Scorpio Broker.
- **LD Context Resolver:** This component is responsible for expanding the NGSI-LD document based on the JSON-LD @context for further processing by the other components of the Scorpio Broker.
- **Subscription & Notification Manager:** This component is responsible for handling CRUD operations related to entities and/or csource subscription & notification.
- **Query Manager:** This component handles simple or complex queries (e.g. geo-query) to the Scorpio Broker.
- **Storage Manager:** This component is responsible for fetching data from the message broker and then transforming them into relevant schema format in order to persist in DB tables. Additionally, this manager also provides interfaces for complex queries to the DB e.g. Geo query or cross-domain entity context relationship queries.
- **Context Registry Manager:** This component is responsible for providing interfaces for CRUD operations of csource registration/query/ subscription.
- **Health Check & Monitoring:** This component is responsible for monitoring the health of running services & infrastructure.
- **Message Bus Handler:** Every module of the Scorpio Broker may need to communicate with the bus for the inter-module exchange of messages. This interface is provided by the message bus handler.
- **Storage Connectors:** The Scorpio Broker needs to store certain information in different DB formats. So storage connectors (using any type of message broker methodology) provide the way to connect to those storage systems (which may be present locally or remotely). For example, the entity information could be stored in/streamed to a different types of storage systems e.g. MySQL, PostgreSQL, Bigdata, etc. These connectors could also be implemented for storage resiliency purposes.
- **Context Registry Connector:** Scorpio Broker needs to communicate to the context registry in order to know about the registered context sources (brokers/providers) and the type of data model they support. The context registry connector allows the message broker mechanism to connect to the context registry that may be running locally or remotely in federated mode.
- **Storage:** This is the actual storage (e.g. Postgres/Postgis) where data is persisted.
- **Context Registry:** This is the component which is responsible for saving the registration of the context sources/producers.

Deployment Architecture

This section is covering the deployment architecture of the Scorpio Broker which is using different technologies stack.

The deployment architecture leverages the Spring Cloud framework that addresses lots of Micro-services concerns(e.g. scaling, monitoring, fault-tolerant, highly available, secure, decoupled, etc.) and Kafka based distributed and scalable message queue infrastructure to provide high performance on message processing for a huge number of context requests which is usual in the IoT domain. The deployment architecture covers the high-level operations (Http based REST with method POST/GET/DELETE/PATCH) request flow from the external world to the Scorpio Broker system. The external request is served through a unified service API gateway interface that exposes a single IP/port combination to be used for all services that the Scorpio Broker system can provide. In reality, each of the Scorpio Broker services will be implemented as a micro-service that can be deployed as an independent standalone unit in a distributed computing environment. That API gateway routes all the incoming requests to the specific Micro-services with the help of registration & discovery service. Once the request reaches at micro-service based on the operation requirement it uses(pub/sub) Kafka topics (message queues) for real-time storage and for providing intercommunication among different micro-services (based on requirement) over message queues.

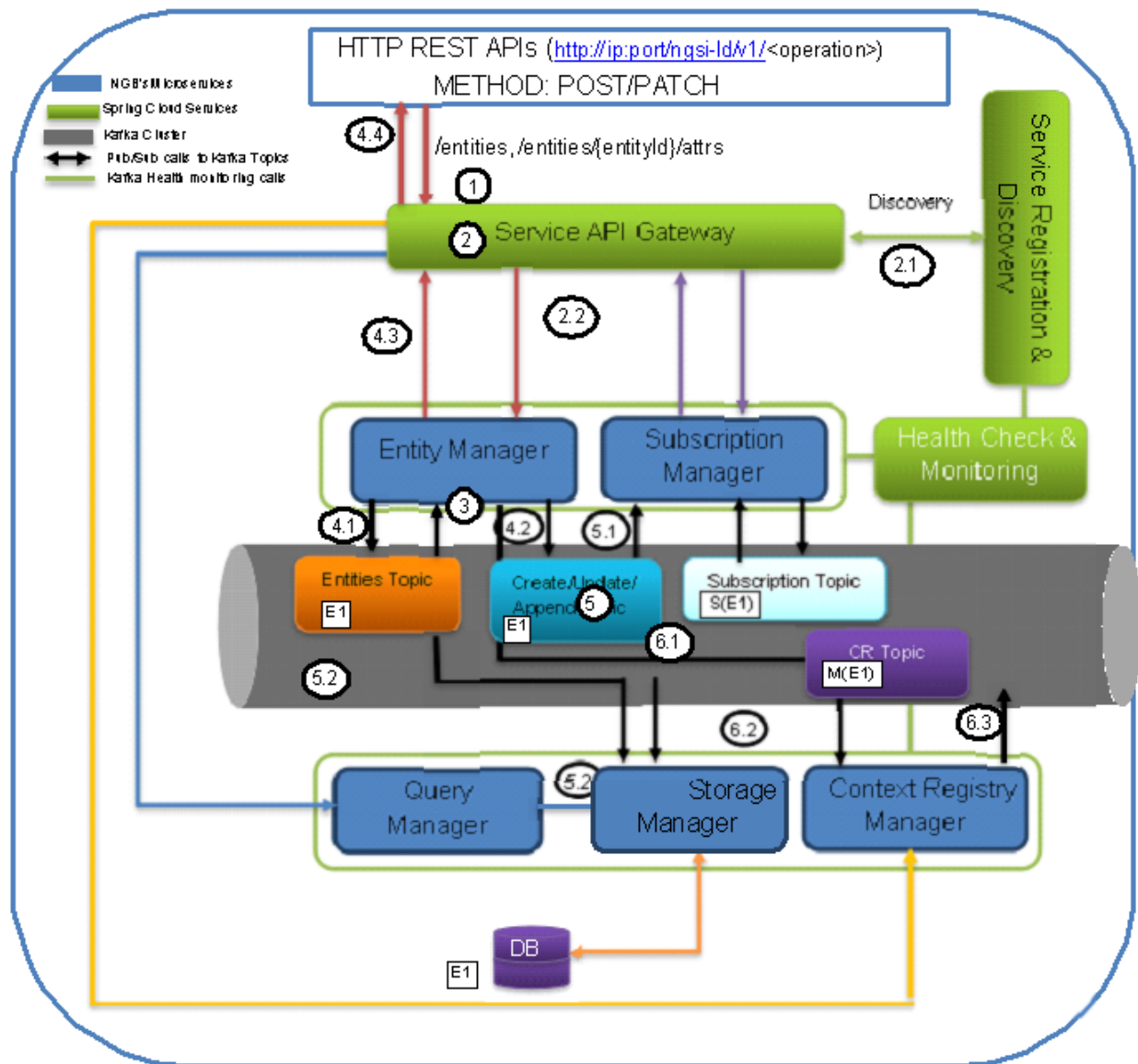


23.1 Entity Create/Update/Append

The Figure is showing the operational flow of entity create/update/append in the Scorpio Broker system. Following are the marked steps interpretation:

1. An application calls the NGSI-LD compliant interface (exposed by service API gateway) to create/update/append an entity in the form of the HTTP POST request.
2. The request enters in service API gateway.
 - 2.1. The service API gateway discovers the actual serving micro-service endpoints (where the incoming requests need to be forwarded) from discovery & registry service.
 - 2.2. The service API gateway forwards the HTTP request to the Entity Manager micro-service.
3. The entity Manager internally calls an LDContext resolver service to resolve the payload with the given context sent along with the POST request. Once the payload is resolved with context, it now fetches the previously stored data/entities from the Topic “Entities” and validates the requested entity against the existing stored entities based on EntityID.
 - If the entity is already present (or with all the attributes and values that are requested to be modified), an error message (“already exists”) will be responded for the same and no further step will be executed.
 - Else it will move for further processing.
4. The Entity Manager (EM) will do publish/store and send the response to the requester for the requested Entity(E1) creation operation given as follows:
 - 4.1. EM publishes the E1 in the Kafka under Topic “Entities”.
 - 4.2. EM publishes the E1 in the Kafka under Topic “Entity_Create/Update/Append” as well.
 - 4.3. Upon successful pub operation, EM will send the response back.

Note: “Entities” topic will save all changes of an entity done over a period of time by any of the create/update/append operations of an entity. However, “Entity_Create/Update/Append” Topic (specific to CREATE operation) will only store the data changes of entity create operation only. Having different topics per operation will avoid ambiguity



situations among different consumers different requirements. E.g. the subscription manager may need to subscribe for the whole entity, a set of specific attributes, or might be some value change of certain attributes. So, managing all these requirements would be hard if a separate topic per operation is not maintained and would be very simplified to provide direct delta change in data for the given entity at any point in time if separate topics per operation are maintained. Therefore, putting all operations data in a single topic cannot offer the required decoupling, simplification, and flexibility to subscribe/manage at operations, data, or delta data level requirements. So that's why creating separate topics per operation and one common topic for recording all changes (require to validate the whole entity changes for all operations over a period of time) of all operations to the given entity is the favorable design choice. The context for the given payload is being stored by the LDContext resolver service in the Kafka topic under the name AtContext.

5. When a message gets published to Kafka Topics, the consumers of that topic will get notified who has subscribed or listening to those topics. In this case, the consumers of "Entity Create/Update/Append" topic upon receiving notification will do the following:

- 5.1. Subscription Manager when getting a notification for the related entity it will check for the notification validation for the current entity and checks if the notification needs to be sent accordingly.

- 5.2. Storage Manager, upon notification from Entities & CR Topics, will trigger the further operations to store/modify the entity related changes in the DB tables.

6. Now entity manager also prepares for registration of the entity data model in the Context Registry. Following are the further functions it performs to achieve the same:

- 6.1. So it prepares the csource registration payload (as per NGSI_LD spec section C.3) from the entity payload and fills the necessary field (like id, endpoint as broker IP, location, etc.). Afterword entity manager writes this created csource payload in the CR Topic.

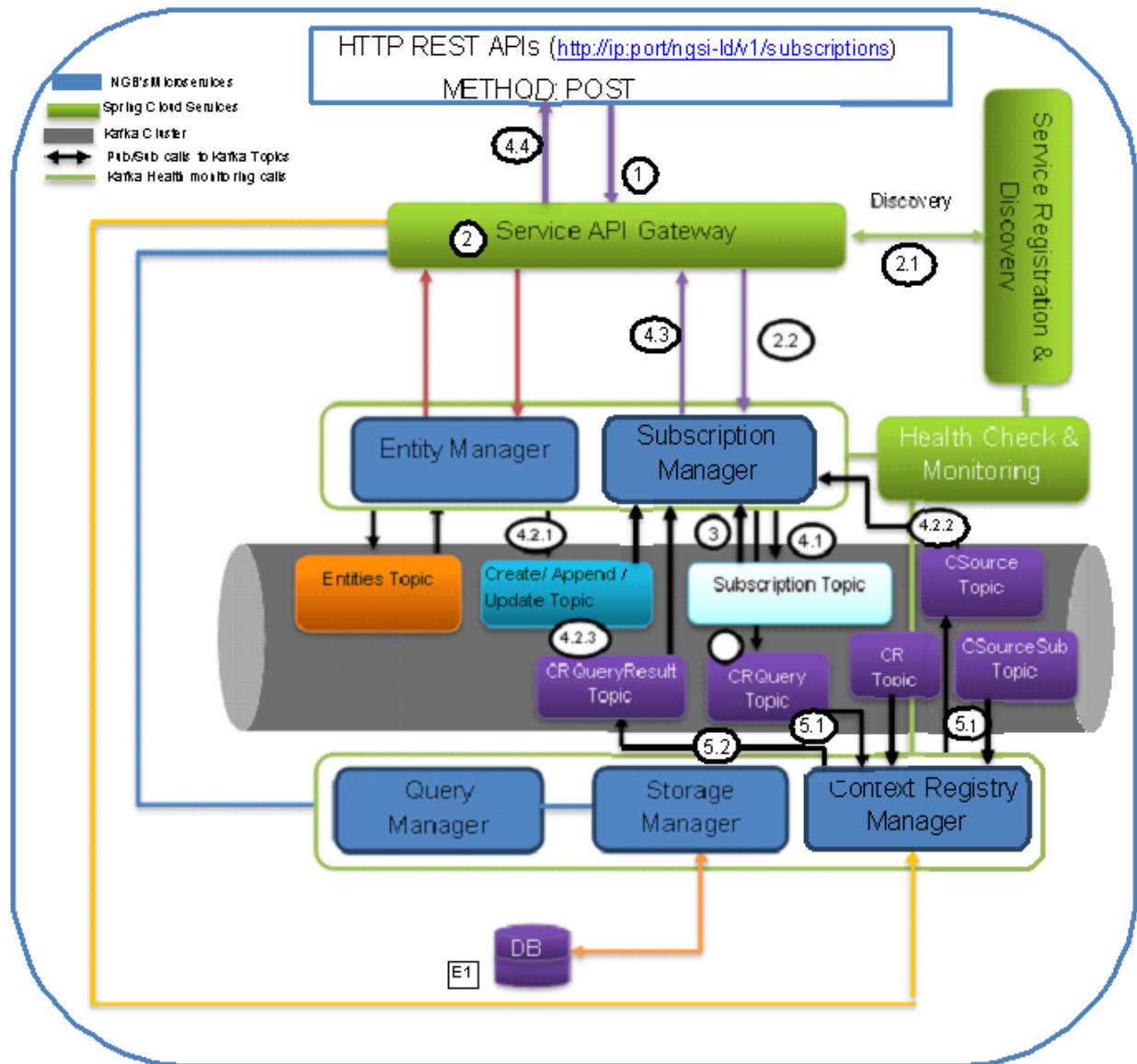
- 6.2. CR Manager listens to this CR topic and then able to know that some entity has registered.

- 6.3. CR manager writes the updates, if any are there, into the Csource Topic.

23.2 Entity Subscription

The Figure is showing the operational flow of entity subscription in the Scorpio Broker system. Following are the marked steps interpretation:

1. An application calls the NGSI-LD compliant interface (exposed by service API gateway) to subscribe for an entity (or attribute) in the form of the HTTP POST request.
2. The request enters in service API gateway.
 - 2.1. The service API gateway discovers the actual serving micro-service endpoints (where the incoming requests need to be forwarded) from discovery & registry service.
 - 2.2. The service API gateway forwards the HTTP request to the Subscription Manager micro-service.
3. The Subscription Manager internally calls an LDContext resolver service to resolve the payload with the given context sent along with the POST request. The subscription manager then fetches the previously stored data/entities from the Topic "Subscription" and validates the requested entity against the existing stored values based on EntityID.
 - If the data for the current request is already present, an error message will be responded for the same and no further step will be executed.
 - Else it will move for further processing.
4. The Subscription Manager (SM) will publish/store and send the response to the requestor for the requested operation given as follows:



4.1.SM publish the subscription S(E1) in the Kafka under Topic “Subscription”

4.2.SM will start the notification functionality and will start/keep listening for related subscription on.

4.2.1. Entity related topics “Create/Update/Append”

4.2.2.Context source related topic i.e. “CSource” topic for any future registration of context sources. Doing this it avoids the need to query CR explicitly for csources for already subscribed items/entities.

4.2.3.CRQueryResult Topic for gathering results of the raised specific queries, if any are there.

4.2.4. Upon successful subscription condition of subscription request, SM will notify the subscribed entity to the given endpoint back. And also do the remote subscriptions to the context sources provided by the context registry.

4.3. Upon successful pub operation, SM will send the response back

5.SM optionally may raise the query to CR by posting in the CRQuery Topic for each of the subscription requests received (only once per each subscription request). When a message gets published to CRQuery Topic, the consumer CR will be notified who has subscribed or listening on this topic. Now, CR will do the following:

5.1. CR will receive the notification and checks for the list of context sources by pulling data from CR Topic and/or CSourceSub Topic for whom this subscription may valid.

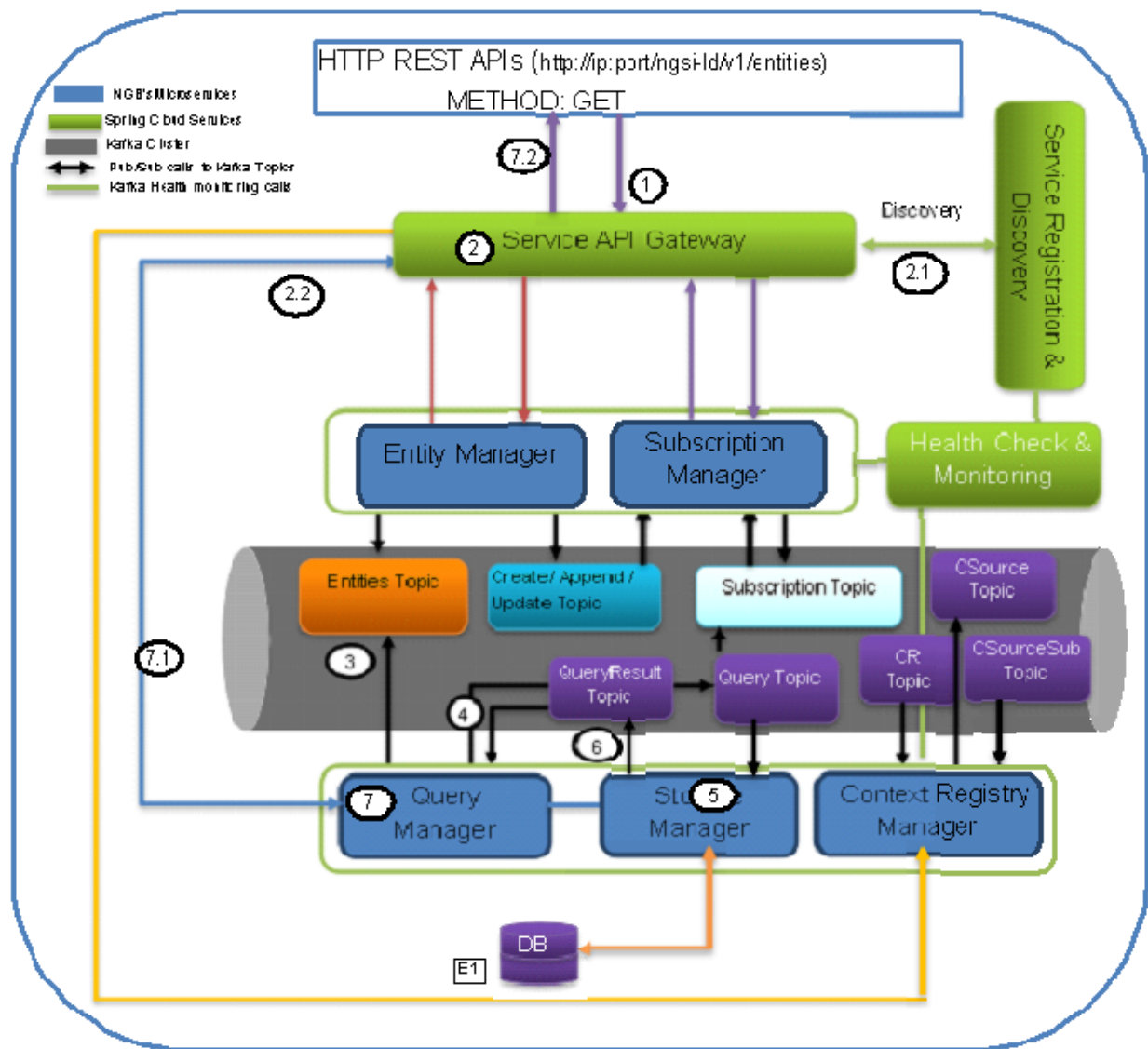
5.2.CR publishes the list of context sources into the CRQueryResult topic upon which the SM would have already started listening and repeat steps 4.2.3 and 4.2.4.

Note: CSource Topic will contain the list of context sources registered through Csource registration interface directly. CR Topic will contain the map of Entity Data model (maintained as an entity ID) created based on entity creation request (through IoT broker interface) and/or provider/data source of that entity model. Limitation: In the first release of Scorpio Broker, Csource query is not supported instead csource query is based on the internal messaging queue mechanism. In the future, both the message queue and Rest based csource query would be supported.

23.3 Query

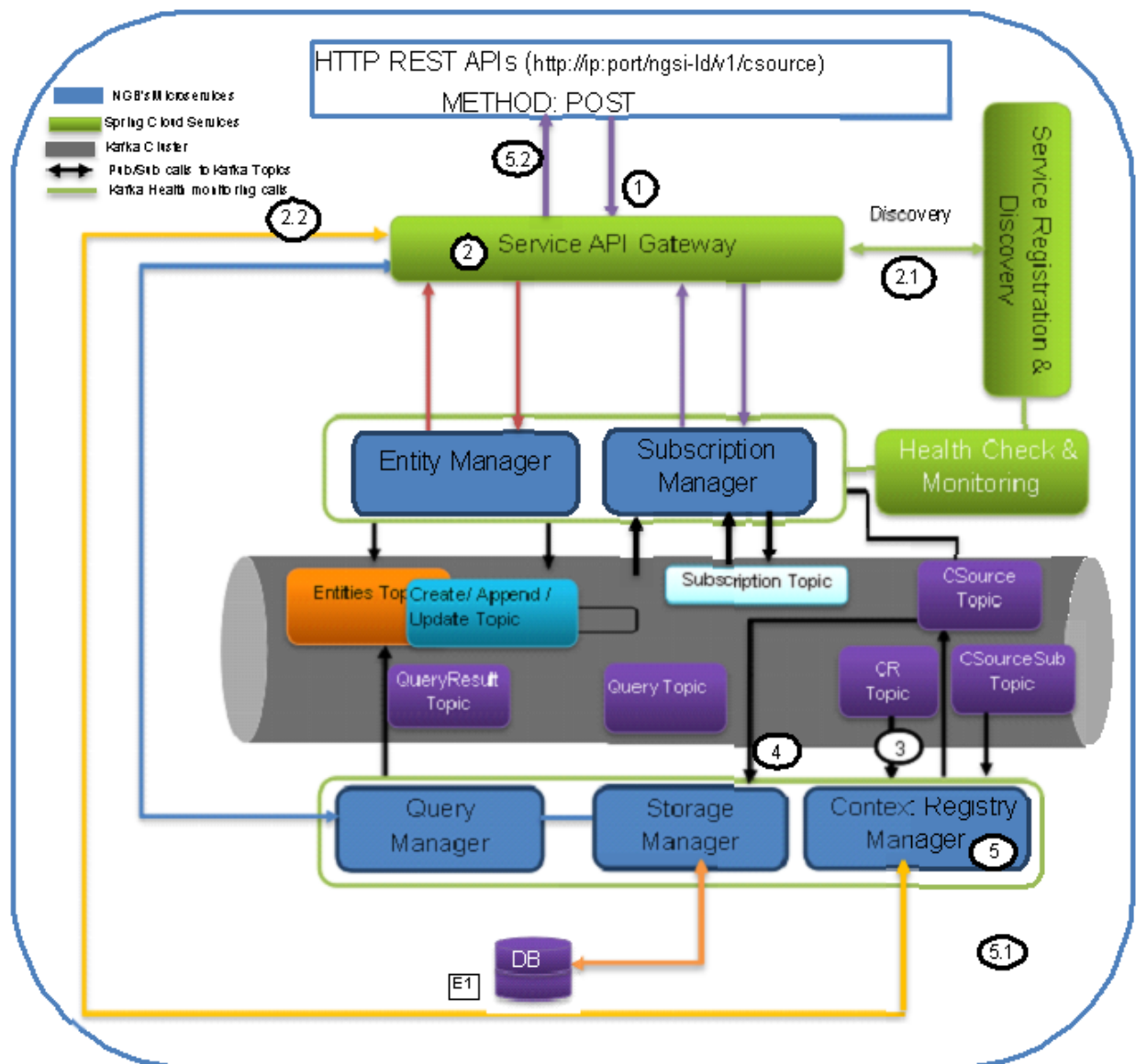
The Figure is showing the operational flow of entity subscription in the Scorpio Broker system. Following are the marked steps interpretation:

1. An application calls the NGSI-LD compliant interface (exposed by service API gateway) to query for entities/an entity/attribute in the form of an HTTP GET request.
2. The request enters in service API gateway.
- 2.1. The service API gateway discovers the actual serving micro-service endpoints (where the incoming requests need to be forwarded) from discovery & registry service.
- 2.2. The service API gateway forwards the HTTP request to the Query Manager micro-service.
3. The query manager now fetches the previously stored data/entities from the Topic “Entities”.
 - If the query is for all entities or specific entities with id and/or attribute is requested, this will be directly served based on Kafka Entity topic data by query manager without involving the storage manager. In short simpler queries like non-geo queries or without regular expression queries associated with entity or entities can be served directly. In this case, the response will be sent back and processing jumps to step 7.2.
 - For complex queries, the query manager will take help from the storage manager as mention in the following steps.
4. The Query Manager (in case of complex queries) will publish the query (embedding a used in the message and other metadata) into the Query topic which is being listened by the Storage manager.
5. The storage manager gets the notification for the requested query and starts processing the query over the DB data and builds the query response.



6. The storage manager publishes the response of query in the Query topic which is being listened by Query manager.
7. The QM receives the notification from the QueryResult topic.
- 7.1. It sends the HTTP response back to the API gateway.
- 7.2. API gateway sends back the response to the end-user/requestor.

23.4 Context Source Registration



The Figure is showing the operational flow of context source registration in the Scorpio Broker system. Following are the marked steps interpretation:

1. An application calls the NGSI-LD compliant interface (exposed by service API gateway) to csource registration for in the form of an HTTP POST request.

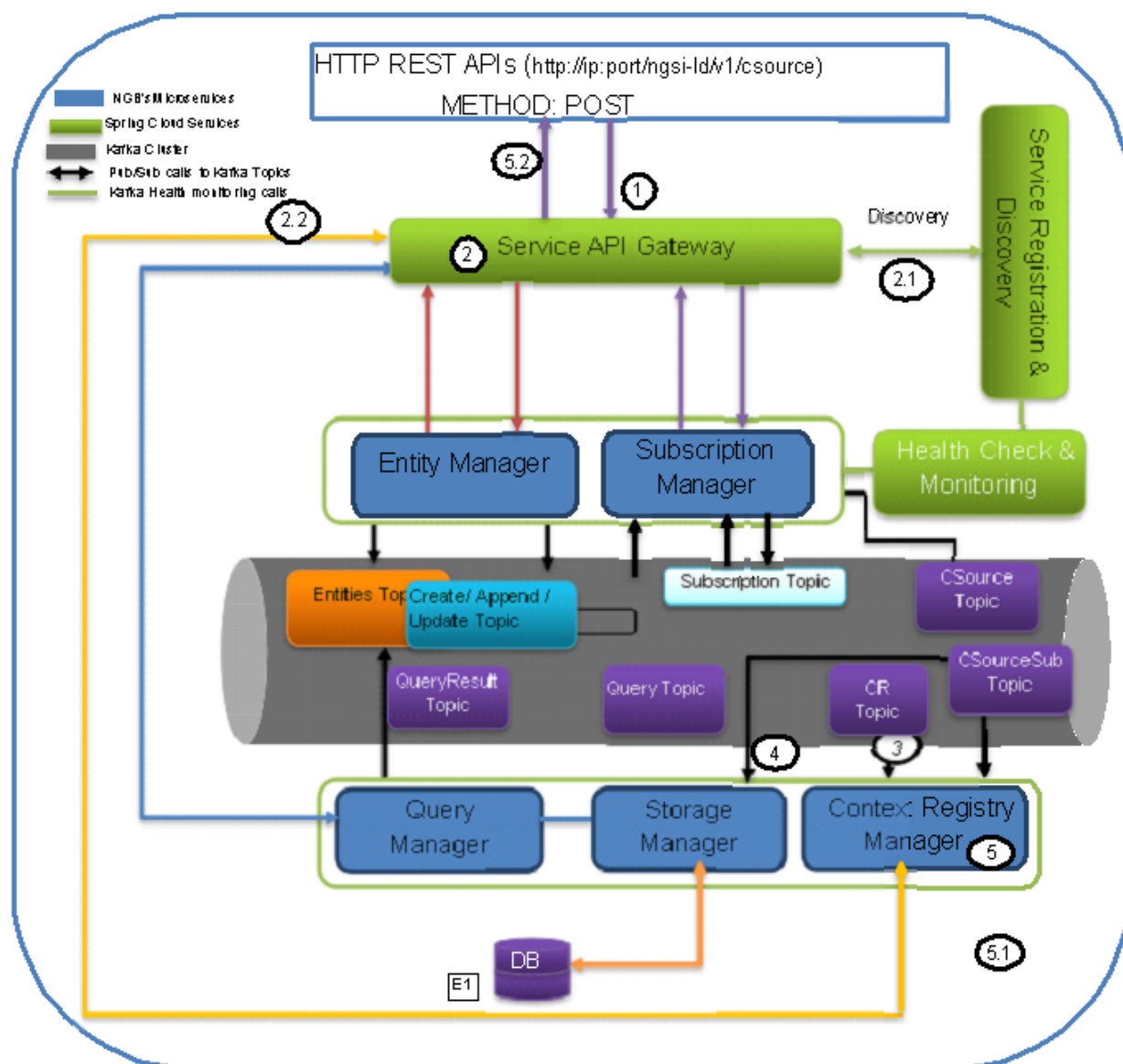
2. The request enters in service API gateway.
 - a. The service API gateway discovers the actual serving micro-service endpoints (where the incoming requests need to be forwarded) from discovery & registry service.
 - b. The service API gateway forwards the HTTP request to the Context Registry (CR) Manager micro-service.
3. The CR manager now fetches the previously stored data/entities from the Topic “CSource”.
 - a. If the entry for the request csource is already present it exits the processing and informing the same to the requester. If it is not present, then it continues for further processing.
 - b. Now the CR manager performs some basic validation to check if this is a valid request with the valid payload.
 - c. CR manager now writes this payload into the Csource Topic.
4. The Storage Manager will keep listening for the Csource topic and for any new entry write it perform the relative operation in the database.
5. The CR manager prepares the response for csource request and
 - 5.1 sends the Http response back to the API gateway.
 - 5.2 API gateway sends back the response to the end-user/requester.

Note: For Conext Source Update request only the payload will get changes and in step 3 upon validation for the existing entity it will not exit rather it will update the retrieved entity and write it back into the Kafka. The rest of the flow will remain mostly the same.

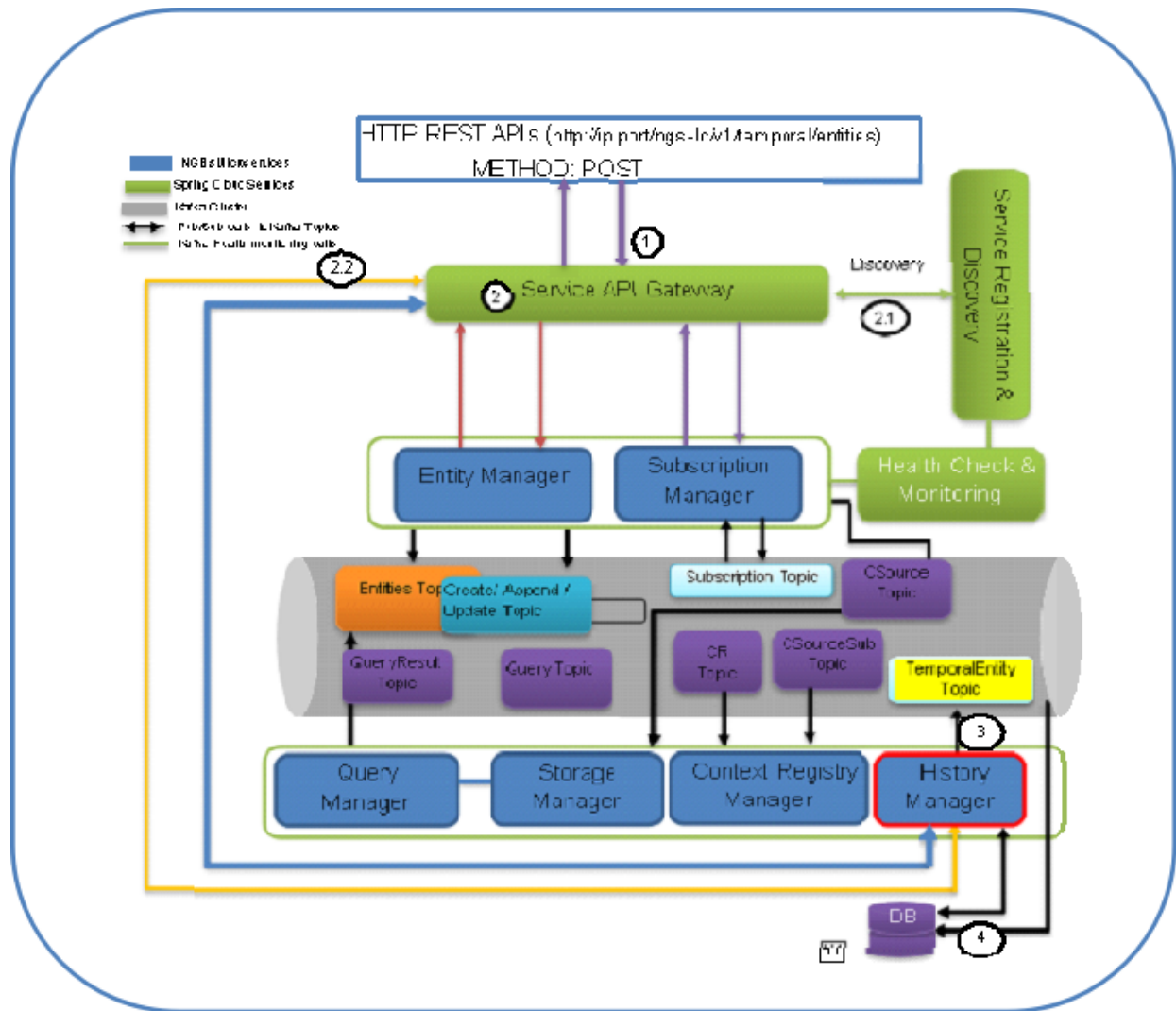
23.5 Context Source Subscription

The Figure Scorpio Broker Context Source Subscription Flow is showing the operational flow of context source subscriptions in the Scorpio Broker system. Following are the marked steps interpretation:

1. An application calls the NGS-LD compliant interface (exposed by service API gateway) to csource updates in the form of an HTTP POST request.
2. The request enters in service API gateway.
 - a. The service API gateway discovers the actual serving micro-service endpoints (where the incoming requests need to be forwarded) from discovery & registry service.
 - b. The service API gateway forwards the HTTP request to the Context Registry (CR) Manager micro-service.
3. The CR manager now fetches the previously stored data/entities from the Topic “CSourceSub”.
 - a. Now the CR manager performs some basic validation to check if this is a valid request with the valid payload.
 - b. If the entry for the request csource subscription is already present it exits the processing and informing the same to the requester. If it is not present, then it continues for further processing.
 - c. CR manager now writes this payload into the CsourceSub Topic.
 - d. In parallel, it will also start an independent thread to listen Csource Topic for the requested subscription and upon the successful condition, the notification will be sent to the registered endpoint provided under subscription payload.
4. The Storage Manager will keep listening for the CsourceSub topic and for any new/updated entry write it perform the relative operation in the database.
5. The CR manager prepares the response for csource subscription request and
 - 5.1 sends the Http response back to the API gateway.
 - 5.2 API gateway sends back the response to the end-user/requester.



23.6 History



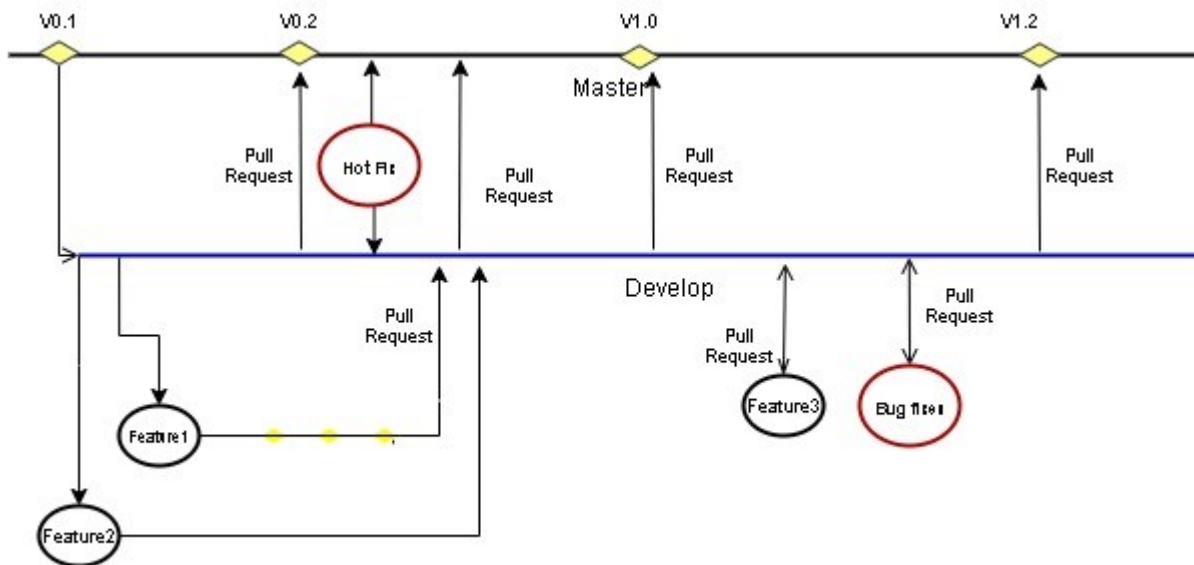
The Figure is showing the operational flow of entity subscription in the Scorpio Broker system. Following are the marked steps interpretation:

1. An application calls the NGSI-LD compliant interface (exposed by service API gateway) to the history manager in the form of an HTTP POST request.
2. The request enters in service API gateway.
 - a. The service API gateway discovers the actual serving micro-service endpoints (where the incoming requests need to be forwarded) from discovery & registry service.
 - b. The service API gateway forwards the HTTP request to the History Manager micro-service.
3. The history manager now executes the EVA algorithm approach on the received payload and push payload attributes to Kafka topic "TEMPORALENTITY".

Note: History Manager must walk through each attribute at the root level of the object (except @id and @type). Inside each attribute, it must walk through each instance (array element). Then, it sends the current object to the Kafka topic TEMPORALENTITY.

4. The history manager will keep listening to the “TEMPORALENTITY” topic and for any new entry and performs the relative operation in the database.

24.1 Branch Management Guidelines



The community can have two main branches with an infinite lifetime:

1. **Master branch:** This is a highly stable branch that is always production-ready and contains the last release version of source code in production.
2. **Development branch:** Derived from the master branch, the development branch serves as a branch for integrating different features planned for an upcoming release. This branch may or may not be as stable as the master branch. It is where developers collaborate and merge feature branches. All of the changes should be merged back into the master somehow and then tagged with a release number.

Apart from those two primary branches, there are other branches in the workflow:

- **Feature Branch:** Forked from the development branch for feature development i.e. enhancement or documentation. Merged back to the development branch after feature development or enhancement implementation.
- **Bug Branch:** Ramify from the development branch. Merged back to the development branch after bug fixing.
- **Hotfix branch:** Hotfix branches are created from the master branch. It is the current production release running live and causing troubles due to a severe bug. But changes in development are yet unstable. We may then branch off a hotfix branch and start fixing the problem. It should be the rarest occasion, in case only critical bugs.

Note: Only NLE and NECTI members have the privilege to create and merge the Hotfix branch.

| Branch | Branches naming guideline | Remarks |
|------------------|--|---|
| Feature branches | Must branch from: <i>development</i> . Must merge back into: <i>development</i> . Branch naming convention: <i>feature-feature_id</i> | <i>feature_id</i> is the Github issue id from https://github.com/ScorpioBroker/ScorpioBroker/issues |
| Bug Branches | Must branch from: <i>development</i> . Must merge back into: <i>development</i> . Branch naming convention: <i>bug-bug_id</i> | <i>bug_id</i> is the Github issue id from https://github.com/ScorpioBroker/ScorpioBroker/issues |
| Hotfix Branches | Must branch from: <i>master branch</i> . Must merge back into: <i>master branch</i> . Branch naming convention: <i>hotfix-bug number</i> | <i>Bug number</i> is the Github issue id from https://github.com/ScorpioBroker/ScorpioBroker/issues |

24.1.1 Permissions to the branches

- **Master** - We tend to very strict that only NLE members and privileged members of NECTI can merge on the Master branch and accept the pull requests. Pull requests to master can be raised by only NECTI OR NLE members.
- **Development** - Any community member can raise the pull request to the development branch but it should be reviewed by NLE or NECTI members. Development branches commits will be moved to the master branch only when all the test cases written under NGSI-LD test suites, will run successfully.

Getting a docker container

The current maven build supports two types of docker container generations from the build using maven profiles to trigger it.

The first profile is called 'docker' and can be called like this

```
mvn clean package -DskipTests -Pdocker
```

this will generate individual docker containers for each microservice. The corresponding docker-compose file is *docker-compose-dist.yml*

The second profile is called 'docker-aaio' (for almost all in one). This will generate one single docker container for all components of the broker except the Kafka message bus and the Postgres database.

To get the aaio version run the maven build like this

```
mvn clean package -DskipTests -Pdocker-aaio
```

The corresponding docker-compose file is *docker-compose-aaio.yml*

25.1 General remark for the Kafka docker image and docker-compose

The Kafka docker container requires you to provide the environment variable *KAFKA_ADVERTISED_HOST_NAME*. This has to be changed in the docker-compose files to match your docker host IP. You can use *127.0.0.1* however this will disallow you to run Kafka in a cluster mode.

For further details please refer to <https://hub.docker.com/r/wurstmeister/kafka>

25.2 Running docker build outside of Maven

If you want to have the build of the jars separated from the docker build you need to provide certain VARS to docker. The following list shows all the vars and their intended value if you run docker build from the root dir

- BUILD_DIR_ACS = Core/AtContextServer
- BUILD_DIR_SCS = SpringCloudModules/config-server
- BUILD_DIR_SES = SpringCloudModules/eureka
- BUILD_DIR_SGW = SpringCloudModules/gateway
- BUILD_DIR_HMG = History/HistoryManager
- BUILD_DIR_QMG = Core/QueryManager
- BUILD_DIR_RMG = Registry/RegistryManager
- BUILD_DIR_EMG = Core/EntityManager
- BUILD_DIR_STRMG = Storage/StorageManager
- BUILD_DIR_SUBMG = Core/SubscriptionManager
- JAR_FILE_BUILD_ACS = AtContextServer-\${project.version}.jar
- JAR_FILE_BUILD_SCS = config-server-\${project.version}.jar
- JAR_FILE_BUILD_SES = eureka-server-\${project.version}.jar
- JAR_FILE_BUILD_SGW = gateway-\${project.version}.jar
- JAR_FILE_BUILD_HMG = HistoryManager-\${project.version}.jar
- JAR_FILE_BUILD_QMG = QueryManager-\${project.version}.jar
- JAR_FILE_BUILD_RMG = RegistryManager-\${project.version}.jar
- JAR_FILE_BUILD_EMG = EntityManager-\${project.version}.jar
- JAR_FILE_BUILD_STRMG = StorageManager-\${project.version}.jar
- JAR_FILE_BUILD_SUBMG = SubscriptionManager-\${project.version}.jar
- JAR_FILE_RUN_ACS = AtContextServer.jar
- JAR_FILE_RUN_SCS = config-server.jar
- JAR_FILE_RUN_SES = eureka-server.jar
- JAR_FILE_RUN_SGW = gateway.jar
- JAR_FILE_RUN_HMG = HistoryManager.jar
- JAR_FILE_RUN_QMG = QueryManager.jar
- JAR_FILE_RUN_RMG = RegistryManager.jar
- JAR_FILE_RUN_EMG = EntityManager.jar
- JAR_FILE_RUN_STRMG = StorageManager.jar
- JAR_FILE_RUN_SUBMG = SubscriptionManager.jar

Config parameters for Scorpio

This section covers all the basic configuration needed for the Scorpio broker. This can be used as the basic template for the various micro-services of the Scorpio.

26.1 Description of various configuration parameters

1. **server:-** In this, the user can define the various server related parameters like **port** and the maximum **number of threads** for the internal tomcat server. This is related to the microservice communication. Be careful with changes.

```
server:
  port: XXXX
  tomcat:
    max:
      threads: XX
```

2. **Entity Topics:-** These are the topics which are used for the internal communication of Scorpio on Kafka. If you change this you need to change things in the source code too.

```
entity:
  topic: XYZ
  create:
    topic: XYZ
  append:
    topic: XYZ
  update:
    topic: XYZ
  delete:
    topic: XYZ
  index:
    topic: XYZ
```

3. **batchoperations:-** Used to define the limit for the batch operations defined by NGSI-LD operations. This is http server config and hardware related. Change with caution.

```
batchoperations:
  maxnumber:
    create: XXXX
    update: XXXX
    upsert: XXXX
    delete: XXXX
```

4. **bootstrap:-** Used to define the URL for the Kafka broker. Change only if you have changed the setup of Kafka

```
bootstrap:
  servers: URL
```

5. **csources Topics:-** These are the topics which are used for the internal communication of Scorpio on Kafka. If you change this you need to change things in the source code too.

```
registration:
  topic: CONTEXT_REGISTRY
```

6. **append:-** Used to define the entity append overwrite option. Change with only with extreme caution.

```
append:
  overwrite: noOverwrite
```

7. **spring:-** Used to define the basic details of the project like service name as well as to provide the configuration details for Kafka, flyway, data source, and cloud. **DO NOT CHANGE THOSE UNLESS YOU KNOW WHAT YOU ARE DOING!**

```
spring:
  application:
    name: serviceName
  main:
    lazy-initialization: true
  kafka:
    admin:
      properties:
        cleanup:
          policy: compact
  flyway:
    baselineOnMigrate: true
  cloud:
    stream:
      kafka:
        binder:
          brokers: localhost:9092
      bindings:
        ATCONTEXT_WRITE_CHANNEL:
          destination: ATCONTEXT
          contentType: application/json
  datasource:
    url: "jdbc:postgresql://127.0.0.1:5432/ngb?ApplicationName=ngb_querymanager"
    username: ngb
    password: ngb
    hikari:
      minimumIdle: 5
      maximumPoolSize: 20
```

(continues on next page)

(continued from previous page)

```

idleTimeout: 30000
poolName: SpringBootHikariCP
maxLifetime: 2000000
connectionTimeout: 30000

```

8. **query Topics:-** These are the topics which are used for the internal communication of Scorpio on Kafka. If you change this you need to change things in the source code too.

```

query:
  topic: QUERY
  result:
    topic: QUERY_RESULT

```

9. **atcontext:-** Used to define the URL for served context by scorpio for scenarios where a mixed context is provided via a header.

```

atcontext:
  url: http://<ScorpioHost>:<ScorpioPort>/ngsi-ld/contextes/

```

10. **Key:-** Used to define the file for the deserialization. DO NOT CHANGE!

```

key:
  deserializer: org.apache.kafka.common.serialization.StringDeserializer

```

11. **reader:-** Used to configure the database to the Scorpio broker, required to perform all the read operations. This example is based on the default config for a local installed Postgres DB

```

reader:
  enabled: true
  datasource:
    url: "jdbc:postgresql://localhost:5432/ngb?ApplicationName=ngb_storagemanager_
↪reader"
    username: ngb
    password: ngb
    hikari:
      minimumIdle: 5
      maximumPoolSize: 20
      idleTimeout: 30000
      poolName: SpringBootHikariCP_Reader
      maxLifetime: 2000000
      connectionTimeout: 30000

```

12. **writer:-** Used to configure the database to the Scorpio broker, required to perform all the write operations. This example is based on the default config for a local installed Postgres DB.

```

writer:
  enabled: true
  datasource:
    url: "jdbc:postgresql://localhost:5432/ngb?ApplicationName=ngb_storagemanager_
↪writer"
    username: ngb
    password: ngb
    hikari:
      minimumIdle: 5
      maximumPoolSize: 20
      idleTimeout: 30000

```

(continues on next page)

(continued from previous page)

```
poolName: SpringBootHikariCP_Writer  
maxLifetime: 2000000  
connectionTimeout: 30000
```

27.1 Missing JAXB dependencies

When starting the eureka-server you may be facing the **java.lang.TypeNotPresentException: Type javax.xml.bind.JAXBContext not present** exception. It's very likely that you are running Java 11 on your machine then. Starting from Java 9 package *javax.xml.bind* has been marked deprecated and was finally completely removed in Java 11.

In order to fix this issue and get eureka-server running you need to manually add below JAXB Maven dependencies to *ScorpioBroker/SpringCloudModules/eureka/pom.xml* before starting:

```

... <dependencies>      ...      <dependency>      <groupId>com.sun.xml.bind</groupId>
<artifactId>jaxb-core</artifactId>      <version>2.3.0.1</version>      </dependency>      <dependency>
<groupId>javax.xml.bind</groupId>      <artifactId>jaxb-api</artifactId>      <version>2.3.1</version>
</dependency>      <dependency>      <groupId>com.sun.xml.bind</groupId>      <artifactId>jaxb-
impl</artifactId>      <version>2.3.1</version>      </dependency>      ...      </dependencies>
...

```

Deployment Guide for Scorpio Broker on Kubernetes

In order to deploy the Scorpio broker on the Kubernetes, the following dependency needs to be deployed:-

1. Postgres.
2. Kafka and Zookeeper.
3. Scorpio Broker microservices.

28.1 For Quick Deployment

28.1.1 Postgres

Please follow the steps to deploy Postgres in your Kubernetes setup:-

1. Firstly the user needs to deploy the Postgres volume files. For this user can clone the Kubernetes files(ScorpioBroker > KubernetesFile > dependencies > volumes > postgres-storage.yaml) or can create a new file with the following code inside it.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv-volume
  labels:
    type: local
    app: postgres
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/db"
```

(continues on next page)

(continued from previous page)

```

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim
  labels:
    app: postgres
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi

```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create a persistent volume and persistent volume claim for Postgres and the user will get the message.

```

persistentvolume/postgres-pv-volume created
persistentvolumeclaim/postgres-pv-claim created

```

User can check these by running the commands:

```

kubectl get pv
kubectl get pvc

```

```

root@master:~# kubectl create -f postgres-storage.yaml
persistentvolume/postgres-pv-volume created
persistentvolumeclaim/postgres-pv-claim created
root@master:~# kubectl get pv
NAME                CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM                STORAGECLASS  REASON  AGE
postgres-pv-volume  5Gi       RWX           Retain          Bound   default/postgres-pv-claim  manual                          28s
root@master:~# kubectl get pvc
NAME                STATUS  VOLUME                CAPACITY  ACCESS MODES  STORAGECLASS  AGE
postgres-pv-claim   Bound   postgres-pv-volume    5Gi       RWX           manual        39s
root@master:~#

```

2. After the persistent volume and persistent volume claim are successfully created user needs to deploy the Postgres deployment file.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: postgres
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      component: postgres
  strategy: {}
  template:
    metadata:
      labels:
        component: postgres

```

(continues on next page)

(continued from previous page)

```

spec:
  containers:
  - env:
    - name: POSTGRES_DB
      value: ngb
    - name: POSTGRES_PASSWORD
      value: ngb
    - name: POSTGRES_USER
      value: ngb
    image: mdillon/postgis
    imagePullPolicy: ""
    name: postgres
    ports:
    - containerPort: 5432
    resources: {}
    volumeMounts:
    - mountPath: /var/lib/postgresql/data
      name: postgreddb
    restartPolicy: Always
    serviceAccountName: ""
    volumes:
    - name: postgreddb
      persistentVolumeClaim:
        claimName: postgres-pv-claim
status: {}

```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create an instance of Postgres and the user will get the message.

```
deployment.apps/postgres created
```

User can check this by running the commands:

```
kubectl get deployments
```

```

root@master: kubectl create -f postgres-deployment.yaml
deployment.apps/postgres created
root@master: kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
postgres  1/1     1            1           16s
root@master:

```

3. Lastly user needs to deploy the service file.

```

apiVersion: v1
kind: Service
metadata:
  labels:
    component: postgres
    name: postgres
spec:
  ports:
  - name: "5432"
    port: 5432

```

(continues on next page)

(continued from previous page)

```
targetPort: 5432
selector:
  component: postgres
status:
  loadBalancer: {}
```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create a clusterIp service of Postgres and the user will get the message.

```
service/postgres created
```

User can check this by running the commands:

```
kubectl get svc
```

```
root@master: kubectl create -f postgres-service.yaml
service/postgres created
root@master: kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP  PORT(S)    AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    23m
postgres     ClusterIP   10.107.255.182   <none>        5432/TCP   9s
root@master:
```

28.1.2 Kafka and zookeeper

To quickly deploy the Kafka and zookeeper, the user can use the deployment files present in the dependencies folder of the Kubernetes files. To deploy these files please follow the following steps:

1. Deploy the zookeeper deployment file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: zookeeper
    name: zookeeper
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      component: zookeeper
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        component: zookeeper
    spec:
```

(continues on next page)

(continued from previous page)

```

containers:
- image: zookeeper
  imagePullPolicy: Always
  name: zookeeper
  ports:
  - containerPort: 2181
    protocol: TCP
  resources:
    limits:
      cpu: 500m
      memory: 128Mi
    requests:
      cpu: 250m
      memory: 64Mi
  terminationMessagePath: /dev/termination-log
  terminationMessagePolicy: File
dnsPolicy: ClusterFirst
restartPolicy: Always
schedulerName: default-scheduler
securityContext: {}
terminationGracePeriodSeconds: 30
status: {}

```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create an instance of Zookeeper and the user will get the message.

```
deployment.apps/zookeeper created
```

User can check this by running the commands:

```
kubectl get deployments
```

```

root@master: kubectl create -f zookeeper-deployment.yaml
deployment.apps/zookeeper created
root@master: kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
postgres      1/1     1            1           8m57s
zookeeper     1/1     1            1           13s
root@master:

```

2. Once the deployment is up and running, deploy the service using the service file.

```

apiVersion: v1
kind: Service
metadata:
  labels:
    component: zookeeper
  name: zookeeper
spec:
  ports:
  - name: "2181"
    port: 2181
    targetPort: 2181
  selector:
    component: zookeeper

```

(continues on next page)

(continued from previous page)

```
status:
  loadBalancer: {}
```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create an instance of Zookeeper and the user will get the message.

```
service/zookeeper created
```

User can check this by running the commands:

```
kubectl get svc
```

```
root@master: kubectl create -f zookeeper-service.yaml
service/zookeeper created
root@master: kubectl get svc
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP  PORT(S)    AGE
kubernetes ClusterIP   10.96.0.1        <none>       443/TCP    36m
postgres  ClusterIP   10.107.255.182   <none>       5432/TCP   13m
zookeeper ClusterIP   10.111.9.216     <none>       2181/TCP    9s
root@master: █
```

3. After the zookeeper service file is successfully deployed, create the PV and PVC for the Kafka using the Kafka storage file present in the dependencies folder.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: kafka-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  labels:
    component: kafka-claim0
  name: kafka-claim0
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
status: {}
```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create a persistent volume and persistent volume claim for Postgres and the user will get the message.

```
persistentvolume/kafka-pv-volume created
persistentvolumeclaim/kafka-claim0 created
```

User can check these by running the commands:

```
kubectl get pv
kubectl get pvc
```

```
root@master: kubectl create -f kafka-pv.yaml
persistentvolume/kafka-pv-volume created
persistentvolumeclaim/kafka-claim0 created
root@master: kubectl get pv
NAME                CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM                STORAGECLASS  REASON  AGE
kafka-pv-volume      1Gi       RWO           Retain          Bound   default/kafka-claim0  manual              7s
postgres-pv-volume   5Gi       RWX           Retain          Bound   default/postgres-pv-claim  manual            36m
root@master: kubectl get pvc
NAME                STATUS  VOLUME                CAPACITY  ACCESS MODES  STORAGECLASS  AGE
kafka-claim0        Bound   kafka-pv-volume       1Gi       RWO           manual         11s
postgres-pv-claim   Bound   postgres-pv-volume    5Gi       RWX           manual         36m
root@master:
```

4. Now deploy the Kafka using the Kafka deployment files.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka
spec:
  replicas: 1
  selector:
    matchLabels:
      component: kafka
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        component: kafka
    spec:
      containers:
        - name: kafka
          image: wurstmeister/kafka
          ports:
            - containerPort: 9092
          resources: {}
          volumeMounts:
            - mountPath: /var/run/docker.sock
              name: kafka-claim0
      env:
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: KAFKA_ADVERTISED_PORT
          value: "9092"
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: zookeeper:2181
```

(continues on next page)

(continued from previous page)

```

- name: KAFKA_ADVERTISED_PORT
  value: "9092"
- name: KAFKA_ADVERTISED_HOST_NAME
  value: $(MY_POD_IP)
hostname: kafka
restartPolicy: Always
serviceAccountName: ""
volumes:
- name: kafka-claim0
  persistentVolumeClaim:
    claimName: kafka-claim0
status: {}

```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create an instance of Postgres and the user will get the message.

```
deployment.apps/kafka created
```

User can check this by running the commands:

```
kubectl get deployments
```

```

root@master: kubectl create -f kafka-deployment.yaml
deployment.apps/kafka created
root@master: kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
kafka     1/1     1            1           9s
postgres  1/1     1            1           27m
zookeeper 1/1     1            1           19m
root@master:

```

5. Finally deploy the Kafka service file. (Only once Kafka deployment moved to running state else sometimes it throws error).

```

apiVersion: v1
kind: Service
metadata:
  labels:
    component: kafka
  name: kafka
spec:
  ports:
  - name: "9092"
    port: 9092
    targetPort: 9092
  selector:
    component: kafka
status:
  loadBalancer: {}

```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create a clusterIp service of Postgres and the user will get the message.

```
service/kafka created
```

User can check this by running the commands:

```
kubectl get svc
```

```
root@master: kubectl create -f kafka-service.yaml
service/kafka created
root@master: kubectl get svc
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP  PORT(S)    AGE
kafka     ClusterIP   10.105.43.62    <none>       9092/TCP   7s
kubernetes ClusterIP   10.96.0.1       <none>       443/TCP    50m
postgres  ClusterIP   10.107.255.182  <none>       5432/TCP   26m
zookeeper ClusterIP   10.111.9.216    <none>       2181/TCP   13m
```

28.1.3 Scorpio Broker

For testing and other lite usage, users can use the All-in-one-deployment(aaio) files(in this all the micro-services are deployed in the single docker container). For this user have two options:

1. **Deployment through helm:** The first step is to get the helm chart of aaio deployment of the Scorpio broker, please download the helm package from GitHub(ScorpioBroker > KubernetesFile > aaio-deployment-files > helm).

Now run the command

helm install {release_name} <helm folder name>

2. **Deployment through YAML files:** user can use the YAML files present in the aaio deployment section and follow the steps:
 - a. Make sure Kafka and Postgres are running, after that deploy the deployment file or the given configuration using the command

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scorpio
  name: scorpio
spec:
  replicas: 2
  selector:
    matchLabels:
      component: scorpio
  strategy: {}
  template:
    metadata:
      labels:
        component: scorpio
    spec:
      containers:
        - image: scorpiobroker/scorpio:scorpio-aaio_1.0.0
          imagePullPolicy: ""
          name: scorpio
          ports:
            - containerPort: 9090
          resources: {}
      restartPolicy: Always
      serviceAccountName: ""
```

(continues on next page)

(continued from previous page)

```
volumes: null
status: {}
```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create an instance of Scorpio Broker and the user will get the message.

```
deployment.apps/scorpio created
```

User can check this by running the commands:

```
kubectl get deployments
```

```
root@master: kubectl create -f scorpio-deployment.yaml
deployment.apps/scorpio created
root@master: kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
kafka     1/1     1            1           23m
postgres  1/1     1            1           51m
scorpio    2/2     2            2           74s
zookeeper 1/1     1            1           42m
root@master: █
```

- b. Once the deployment is up and running create the clusterIP or node port service as per the need.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    component: scorpio
  name: scorpio
spec:
  ports:
    - name: "9090"
      port: 9090
      targetPort: 9090
  selector:
    component: scorpio
status:
  loadBalancer: {}
---
apiVersion: v1
kind: Service
metadata:
  labels:
    component: scorpio
  name: scorpio-node-port
spec:
  type: NodePort
  ports:
    - port: 9090
      targetPort: 9090
      nodePort: 30000
  selector:
    component: scorpio
```

once the file is created apply it through the command:

```
kubectl create -f <filename>
```

This will create an instance of Postgres and the user will get the message.

```
service/scorpio created
service/scorpio-node-port created
```

User can check this by running the commands:

```
kubectl get deployments
```

```
root@master: kubectl create -f scorpio-service.yaml
service/scorpio created
root@master: kubectl create -f scorpio-node-pod-svc.yaml
service/scorpio-node-port created
root@master: kubectl get svc
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|-------------------|-----------|----------------|-------------|----------------|-----|
| kafka | ClusterIP | 10.105.43.62 | <none> | 9092/TCP | 31m |
| kubernetes | ClusterIP | 10.96.0.1 | <none> | 443/TCP | 81m |
| postgres | ClusterIP | 10.107.255.182 | <none> | 5432/TCP | 58m |
| scorpio | ClusterIP | 10.102.76.158 | <none> | 9090/TCP | 59s |
| scorpio-node-port | NodePort | 10.110.84.232 | <none> | 9090:30000/TCP | 34s |
| zookeeper | ClusterIP | 10.111.9.216 | <none> | 2181/TCP | 44m |

```
root@master: █
```

Now, if user have deployed the node post service user can access Scorpio Broker at

```
<ip address of master>:30000
```

28.2 For Production Deployment

28.2.1 Postgres

In order to achieve high availability and to reduce the effort of managing the multiple instances, we opted for crunchy data (<https://www.crunchydata.com/>) to fulfill all our needs for PostgreSQL.

To deploy the crunchy data Postgres follow the link <https://www.crunchydata.com/>

Once we get the running instance of Postgres, logged into the Postgres using the superuser and run the following commands to create a database and the required role:

1. create database ngb;
2. create user ngb with password 'ngb';
3. alter database ngb owner to ngb;
4. grant all privileges on database ngb to ngb;
5. alter role ngb superuser;

After this, your PostgreSQL is ready to use for Scorpio Boker.

Note: Create a cluster using the command:

```
pgo create cluster postgres --ccp-image=crunchy-postgres-gis-ha --ccp-image-tag=centos7-12.5-3.0-4.5.1
```

28.2.2 Kafka and zookeeper

To deploy a Kafka on production, we prefer to use helm since helm provides a hassle-free deployment experience for Kubernetes

To install helm in your Kubernetes cluster follow the link (<https://helm.sh/docs/intro/install/>). The preferred version is helm version 3+.

Once helm is installed use the following command to get the running Kafka cluster:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install kafka bitnami/kafka
```

For more information follow the link (<https://artifacthub.io/packages/helm/bitnami/kafka>)

28.2.3 Scorpio Broker

Once we get the running instance of PostgreSQL as well as Kafka, we are ready to deploy the Scorpio broker.

The first step is to get the helm chart of Scorpio broker, for this download the helm package from GitHub.(user can also use the YAML files if needed) Now run the command

helm install {release_name} <helm folder name>

Now run the **kubectl get pods --all-namespace** to verify that all the microservice of the Scorpio broker are in the running state.

Note: Please use only the latest docker images for the deployment since some older docker images might not work properly with Kubernetes.

Now you are ready to use the Scorpio broker.