BIG DATA & SEMANTICS

Corso di: Big Data Architectures

Dott. Marco Fanfani

DISIT Lab, Sistemi Distribuiti e Tecnologie Internet

http://www.disit.dinfo.unifi.it/

Dipartimento di Ingegneria dell'Informazione

Università degli Studi di Firenze

CC BY SA

INTRODUCTION AND OBJECTIVES

The Computer Science Core: Addressing the logic behind multi-domain data integration.

Beyond Connectivity: Moving from "getting the data" to "interpreting the data."

Key Objectives:

- Identify barriers to data flow.
- Compare geospatial standards.
- Understand the role of semantics and ontologies as the "glue" of the data.

THE MULTI-DOMAIN INTEGRATION CHALLENGE

Interoperability Complexity: Integrating data from disparate domains into a single unified platform.

Example: Urban Domains

- Traffic & Mobility.
- Environment & Air Quality.
- Buildings (BIM).
- Energy Grids & Consumption.
- Public Services (Waste, Water).

The Goal: Creating cross-domain insights (e.g., how mobility affects air quality)

DATA FRAGMENTATION

Industrial Silos: Systems often operate on isolated, proprietary platforms

Data Lock-in: Limited access to information due to non-interoperable industrial standards

Management Silos: Separate departments (energy, water, traffic) managing data with minimal coordination

Fragmented Governance: The lack of a shared digital vision leads to redundant efforts and wasted resources

SYNTACTIC VS. SEMANTIC INTEROPERABILITY

Syntactic Interoperability:

- Focus on data exchange formats (e.g., JSON, XML, CSV).
- Ensures that systems can "read" the message structure.

Semantic Interoperability:

- Focus on the meaning and logic of data (e.g., Ontologies, Knowledge Graphs)
- Ensures that systems "understand" the concept behind the data

The "Significance" Gap: Systems may exchange a JSON file but disagree on the definition of a "Building" or a "Sensor."

BARRIERS TO SCALABILITY

Lack of Universal Protocols: Absence of globally recognized standards hinders replicability

Sporadic Standardization: Current efforts are often domain-specific (e.g., manufacturing, networking)

Performance Degradation: Handling massive cross-domain data at scale remains a technical hurdle

High Engineering Costs: Porting a platform from one scenario to another often requires manual re-coding (e.g., porting a SCDT from a city to another one).

DATA HETEROGENEITY

Format Diversity: Handling JSON, XML, CSV, and high-frequency Binary streams

Data Types:

- **Static**: 3D meshes, CAD, Shapefiles.
- **Dynamic**: Real-time IoT sensor readings.
- **Unstructured**: Social media feeds, text reports, etc.

Coordinate Systems: Challenges in aligning different geodetic reference systems (e.g., ETRS89 to WGS 84)

REAL-TIME MULTI-DATA INTEGRATION

Synchronization Hurdles: Aligning live sensor feeds with static data (e.g., 3D geometries)

Data Ingestion: Managing the volume and velocity of incoming IoT event-driven streams

Visual Integrity: Ensuring that real-time updates (e.g., sensors' update) are rendered accurately without lagging the interface

Situational Awareness: Moving from offline analysis to "on-the-fly" monitoring

SCDT INTEROPERABILITY REQUIREMENTS

Integrated Solutions: Cohesive data gathering, indexing, and distribution

Replicability: Frameworks that can be deployed across different scenarios with minimal re-engineering

Affordability: Preference for open-source foundations to avoid vendor lock-in

Semantic Consistency: Use of shared vocabularies and ontologies (e.g., Km4City) to describe entities

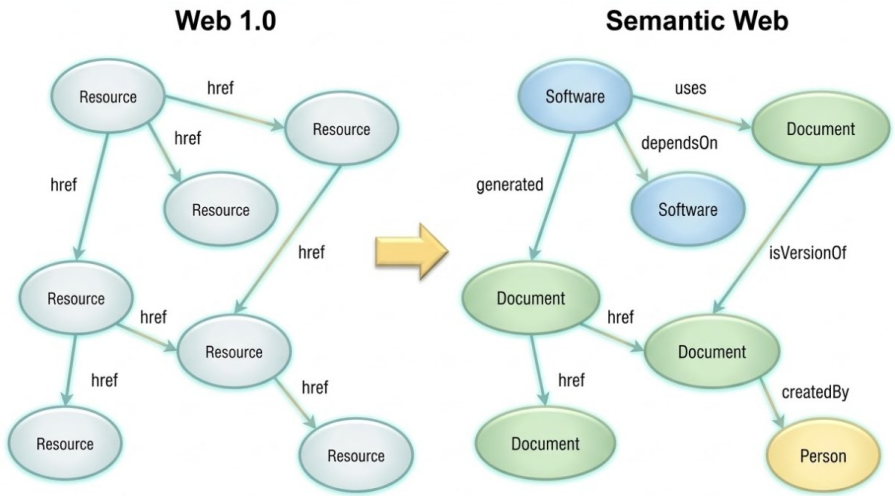
(QUICK) INTRODUCTION TO THE SEMANTIC WEB

Machine-Readable Ecosystem: A web that interconnects vast, distributed data so that software can read and infer from it

Suitability for Inference: Providing the underlying structure for logical reasoning across different data domains

Core Technologies: Foundation built on standard languages (RDF, OWL) for global interoperability

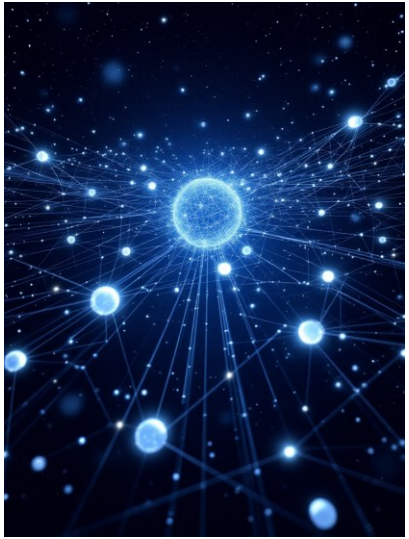
Agent Intelligence: Enabling automated systems to process city information without constant human oversight



SEMANTIC WEB TECHNOLOGIES

The Semantic Web is not intended to be a separate entity from the existing Web, but rather a **functional extension of it**.

The primary goal is to provide data with **well-defined meaning**, enabling humans and computers to work in closer cooperation, by **leveraging semantics-enriched data**



THE SEMANTIC INTEROPERABILITY BRIDGE

Connecting Silos: Ontologies serve as the "glue" that connects disparate platforms through shared vocabularies

Inter-Layer Interface: Acting as a semantic interface between the different layers of the platform architecture (e.g., storages, analytics, interfaces/dashboards)

Unifying Heterogeneous Data: Providing a single logic to manage information from different sensors/sources

Semantics is one of the critical open issue for the future of (Digital Twin) platforms

ONTOLOGY

An ontology is defined as a **formal**, **structured**, and **systematic** representation of knowledge within a specific domain of interest to provide a machine-processable definition of a specific domain

It represents the **schema of knowledge**, not the actual data. It serves as a **blueprint** composed by these building blocks:

- **Classes (types)**: The category of entity that exist (e.g., a building, a sensor)
- **Relationships**: The logical links between classes (e.g., a sensor 'is located in' a room)
- **Rules and Constraint**: Domain-specific knowledge (e.g., 'a building must have an entrance')

FROM ONTOLOGY TO KNOWLEDGE GRAPH

Ontology vs Knowledge Graph

The ontology provides the schema.

The Knowledge Graph contains the actual instances.

Relational Systems	Semantic Systems
Database Schema	Ontology
Database Records	Knowledge Graph
Tables	Classes
Rows	Instances

```
Building
   |
hasSensor
   |
Sensor
```

```
Building_123
   |
hasSensor
   |
Sensor_A45
```

KNOWLEDGE GRAPHS

A Knowledge Graph is a network of real-world entities connected through semantic relationships defined by an ontology.

Purpose

- Integrate heterogeneous data sources
- Represent contextual knowledge
- Enable semantic querying
- Support reasoning and inference

THE GRAPH STRUCTURE

Nodes: Representing entities of interest (e.g., buildings, IoT sensors, vehicles, etc.)

Edges: Defining potentially different and complex relations between these entities

Triple Logic: Every urban fact is stored as a *(Subject)-(Predicate)-(Object)* statement

Non-Linearity: Enabling **flexible** data representation that traditional relational tables cannot support

Advantages

- Flexible structure
- Easy integration of heterogeneous data
- Explicit relationships
- Machine-readable semantics

DYNAMIC KNOWLEDGE GRAPHS

The Knowledge Graph becomes a **continuously updated semantic mirror** of the physical word.

Real-time Evolution: Graphs that evolve and update based on continuous changes in the physical world

State-Awareness: Capturing the current "pulse" of assets through IoT integration

Event-Driven Updates: Automatically modifying relationships as entities change their status (e.g., a road closing due to an accident)

Synchronized Mirroring: Maintaining high fidelity between the virtual graph and reality

KG vs DATABASES

An ontology represents an evolutionary step beyond a traditional database by adding a **layer of rich vocabulary and formal semantics** to facts

While standard databases focus primarily on the structure and storage of data, **ontologies capture the inherent meaning and nuances of relationships**

Databases manage data;

Knowledge Graphs manage knowledge.

Example:

- a database might simply link *a building* to *a street*, and miss that the street is in a city,
- an ontology can formalize that *a street is part of a city* allowing for deductive reasoning (or inference)

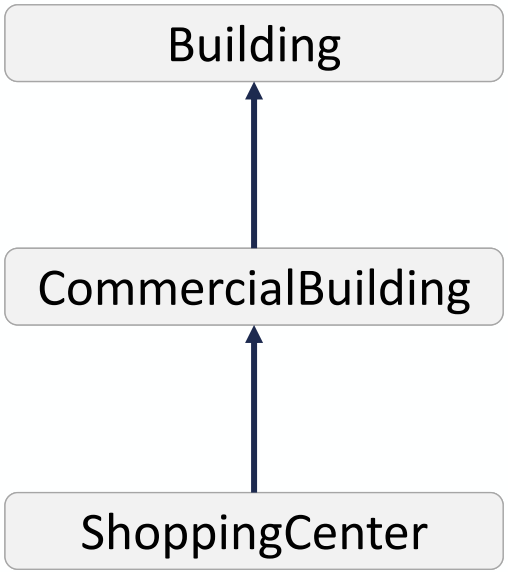
CLASSES AND TAXONOMIES

Classes, also known as Concepts, represent the general categories or types of entities found within a domain, such as "*Building*," "*City*," or "*Road*"

These concepts are often organized into **hierarchical taxonomies** using *parent-child relationship*, which allows for a structured classification of knowledge

A critical feature of this structure is **inheritance**:

- Subclasses automatically **inherit all logical restrictions and properties** from their parent classes in the hierarchy



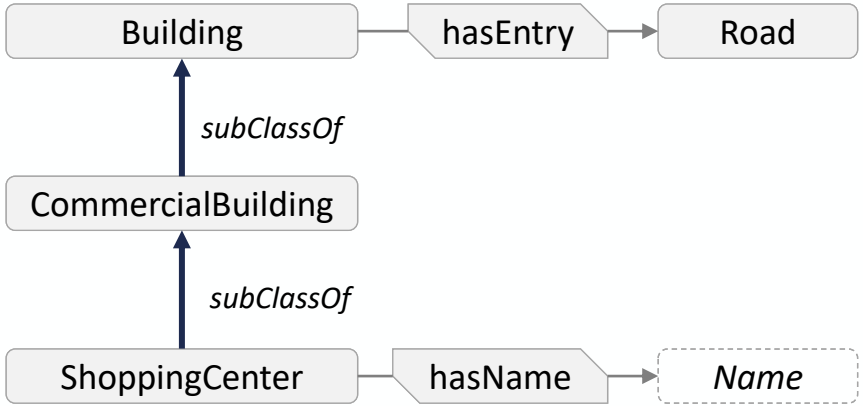
PROPERTIES

Properties allow us to move beyond simple categorization to describe the **internal structure** of concepts and the **complex relationships** between concepts

In the Semantic Web framework, properties are categorized into two main types:

- **Object Properties**: These define binary relations between two individuals
- **Datatype Properties**: These link an individual to a specific data value or "Literal"

Labels, comments, versioning info, etc. can also be specified



TOP-LEVEL ONTOLOGIES

Cross-Domain Foundation: Using ontologies like Basic Formal Ontology (BFO) to provide a shared logic

Interoperability Catalyst: Easing the integration of disparate domains by providing common 'parent' concepts

Refinement Process: Domain-specific concepts are built as 'children' of TLO concepts to ensure structural alignment

Standardization Target: Reducing the fragmentation of models by adhering to high-level formal standards

DOMAIN-SPECIFIC ONTOLOGIES

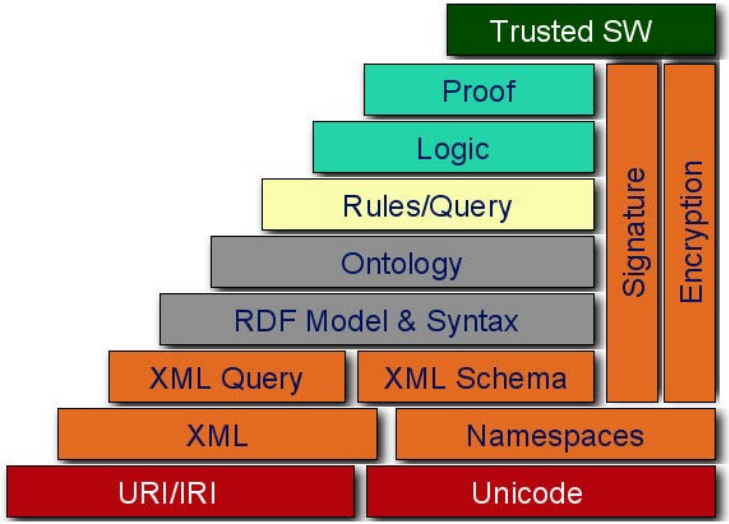
Vertical Specialization: Models tailored to specific subsystems like IoT, mobility, energy, construction, …

Contextual Depth: Describing physical components (e.g., floors, industrial machines, plant parts) with domain-specific accuracy

Data Aggregation: Unifying information from diverse sources within a single urban vertical

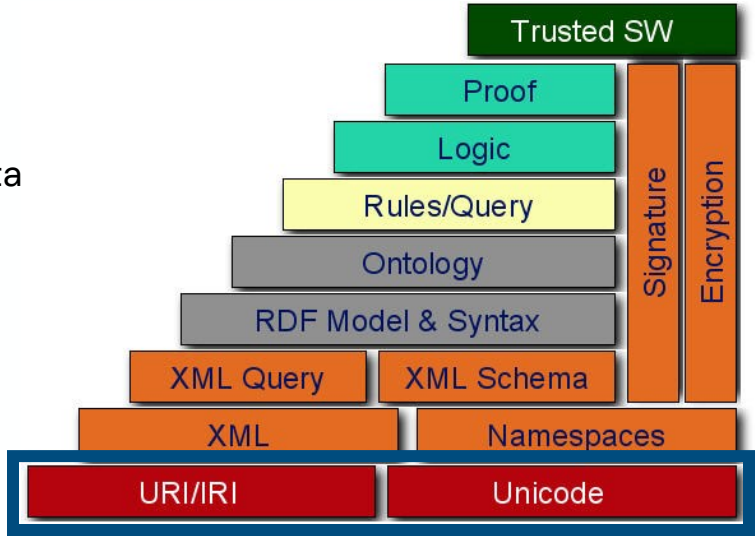
SEMANTIC WEB STACK

The Semantic Web Stack represents a hierarchical suite of open, standardized W3C technology specifications that collectively define the infrastructure for a global Web of Data



SEMANTIC WEB STACK

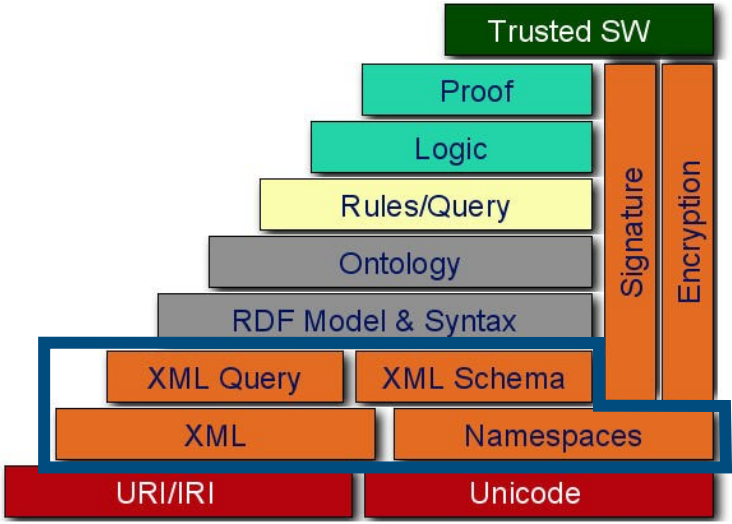
The Semantic Web Stack represents a hierarchical suite of open, standardized W3C technology specifications that collectively define the infrastructure for a global Web of Data



At the foundational level, the stack relies on **URIs (Uniform Resource Identifiers)** and **Unicode** to provide a persistent, global mechanism for identifying real-world entities

SEMANTIC WEB STACK

The Semantic Web Stack represents a hierarchical suite of open, standardized W3C technology specifications that collectively define the infrastructure for a global Web of Data

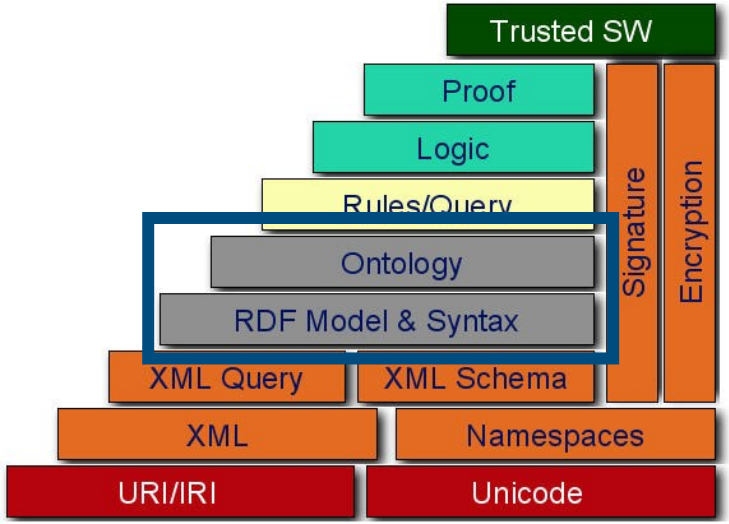


XML serves as a **common syntax** for document structure

At the foundational level, the stack relies on **URIs (Uniform Resource Identifiers)** and **Unicode** to provide a persistent, global mechanism for identifying real-world entities

SEMANTIC WEB STACK

The Semantic Web Stack represents a hierarchical suite of open, standardized W3C technology specifications that collectively define the infrastructure for a global Web of Data



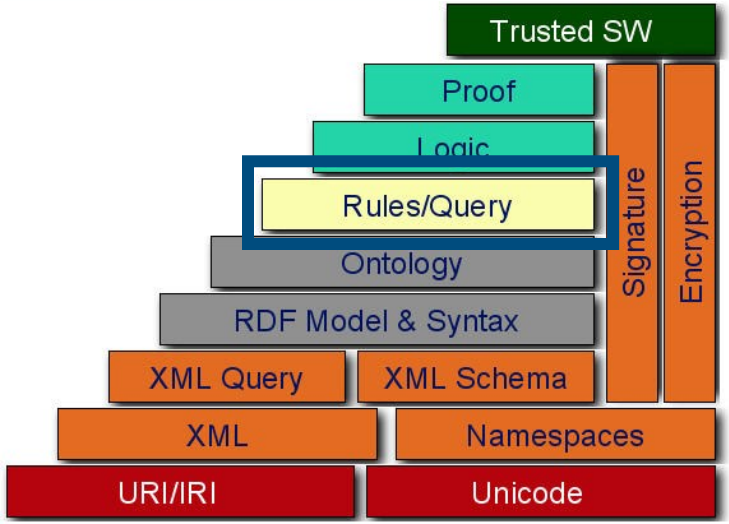
RDF Schema (**RDFS**) introduces **basic taxonomies** and **class hierarchies**, while the Web Ontology Language (**OWL**) provides a rich, logic-based **knowledge representation language**

XML serves as a **common syntax** for document structure

At the foundational level, the stack relies on **URIs (Uniform Resource Identifiers)** and **Unicode** to provide a persistent, global mechanism for identifying real-world entities

SEMANTIC WEB STACK

The Semantic Web Stack represents a hierarchical suite of open, standardized W3C technology specifications that collectively define the infrastructure for a global Web of Data



To **access** and **manipulate** this structured knowledge, **SPARQL** serves as the standardized semantic query language

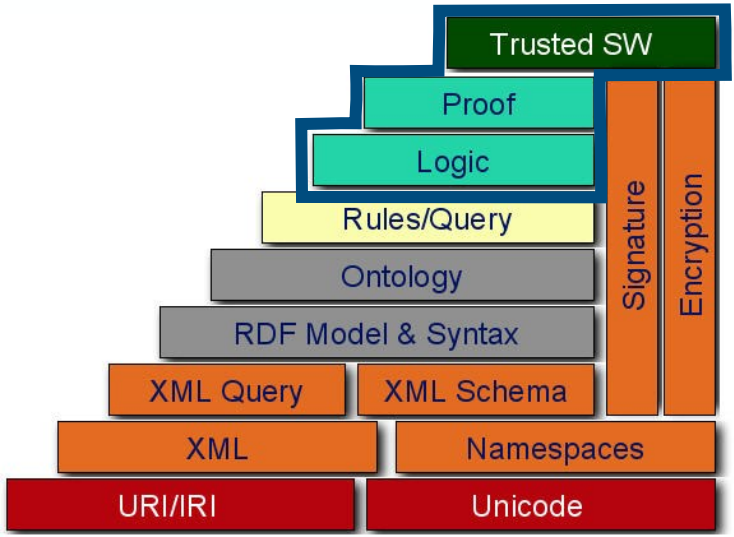
RDF Schema (**RDFS**) introduces **basic taxonomies** and **class hierarchies**, while the Web Ontology Language (**OWL**) provides a rich, logic-based **knowledge representation language**

XML serves as a **common syntax** for document structure

At the foundational level, the stack relies on **URIs (Uniform Resource Identifiers)** and **Unicode** to provide a persistent, global mechanism for identifying real-world entities

SEMANTIC WEB STACK

The Semantic Web Stack represents a hierarchical suite of open, standardized W3C technology specifications that collectively define the infrastructure for a global Web of Data



Unifying Logic, Proof, and Trust--represent the **ongoing research**

To **access** and **manipulate** this structured knowledge, **SPARQL** serves as the standardized semantic query language

RDF Schema (**RDFS**) introduces **basic taxonomies** and **class hierarchies**, while the Web Ontology Language (**OWL**) provides a rich, logic-based **knowledge representation language**

XML serves as a **common syntax** for document structure

At the foundational level, the stack relies on **URIs (Uniform Resource Identifiers)** and **Unicode** to provide a persistent, global mechanism for identifying real-world entities

URIS AND URLS

In the Semantic Web architecture, every unique entity, concept, or relationship **must be identified** by a global and persistent identifier.

The standard mechanism for this is the **Uniform Resource Identifier** (URI)

- URIs can take the form of URNs (**Uniform Resource Names**),
- URLs (**Uniform Resource Locators**) are generally used by the industry

By using the HTTP protocol, these identifiers allow software agents to **retrieve a machine-readable description** of the resource directly from the web

RDF – THE RESOURCE DESCRIPTION FRAMEWORK

The Resource Description Framework (RDF) is a W3C recommendation that serves as the **foundational data model** for interchange on the web

Unlike the relational model, which is based on tables, RDF is inherently **graph-based**. This structure is highly effective for modelling complex, real-world knowledge where entities are interconnected in a non-linear fashion



RDF TRIPLE

All information in an RDF-based Knowledge Graph is represented through a simple, atomic structure known as a **triple**.

A triple consists of three components:

- a **Subject** (denotes the entity being described)
- a **Predicate** (specifies the type of relationship or attribute)
- an **Object** (indicates the related entity or a literal value)

For example:

School of Engineering is located in Via Santa Marta

[SchoolOfEngineering] --[isLocatedIn]--> [ViaSantaMarta]

(Subject) (Predicate) (Object)

DIRECTED LABELLED GRAPHS

While individual triples represent isolated facts, a collection of triples naturally forms a **directed labelled graph**

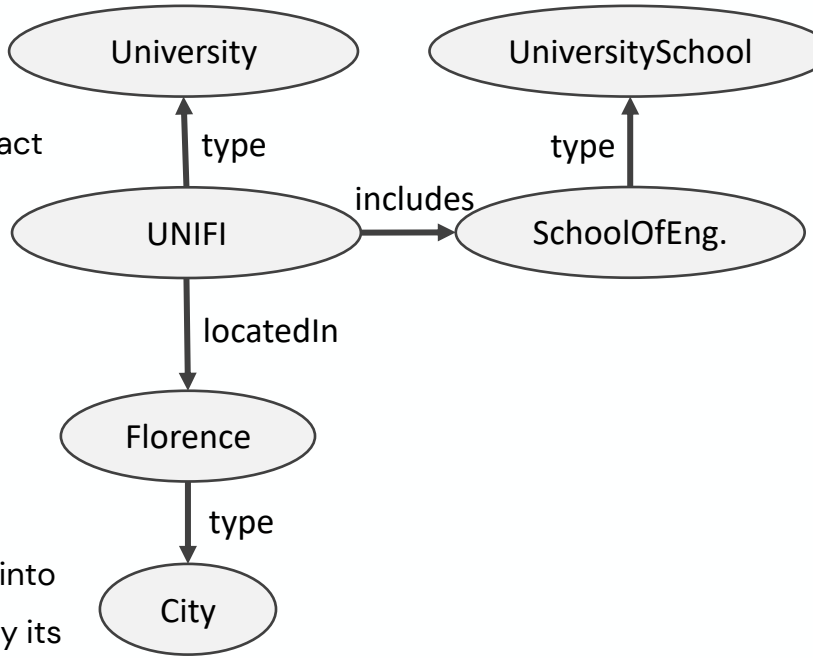
The subjects and objects serve as **nodes**, while the predicates act as **directed**, **labelled edges**

This graph-based approach allows for **multi-attribute** representation

[UNIFI] --[locatedIn]--> [Florence]

[UNIFI] --[includes]--> [SchoolOfEng.]

This interconnectedness is what transforms a simple database into a **Knowledge Graph** where the context of an entity is defined by its position and its surrounding relationships within the network



RDFS – ADDING SCHEMA TO RDF

While RDF provides the data model for triples, it lacks the vocabulary to define **categories** or **hierarchies**

RDF Schema (**RDFS**) was introduced to fill this gap

RDFS adds **basic inference semantics** to RDF. It allows us to define "Classes" (types of things) and organize them into hierarchies

Crucially, *RDFS allows a reasoner to understand that if a resource belongs to a specific class, it also automatically belongs to all parent classes in the hierarchy*

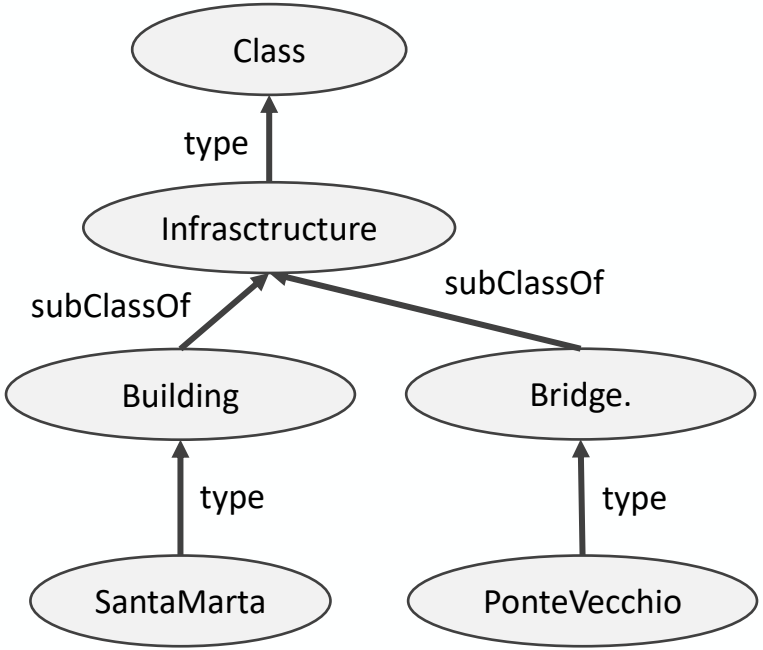
RDFS CLASS HIERARCHIES AND SUBCLASSOF

The ***rdfs:subClassOf*** property is the cornerstone of taxonomic modelling in the Semantic Web.

It defines a relationship where every instance of Class A (child) is also an instance of Class B (father)

This enables the **creation of deep inheritance trees**

This mechanism supports **knowledge reuse**: domain experts can link their specific local classes to well-known global classes, ensuring **interoperability** across different Knowledge Graphs



RDFS PROPERTY HIERARCHIES AND DOMAIN/RANGE

RDFS extends hierarchical logic to properties via *rdfs:**subPropertyOf***. This allows for more granular relations to inherit the semantics of broader ones (e.g., "*paints*" could be a subproperty of "*creates*")

Furthermore, RDFS introduces *rdfs:**domain*** and *rdfs:**range*** to constrain how properties are used

```
[locatedIn] --(domain)--> [University]

[locatedIn] --(range)--> [City]
```

These constraints act as **powerful inference tools**:

if a property has a domain of `[University]` *and it is applied to a subject, e.g.* `[UNIFI]`*, the system automatically infers that* `[UNIFI]` *is an instance of a* `[University]`*, even if not explicitly stated*

WEB ONTOLOGY LANGUAGE (OWL)

While RDF Schema (RDFS) provides a solid foundation for defining **classes and property hierarchies**, its knowledge expressivity is often too limited for complex, real-world domains

The transition to OWL is driven by the need for a **more expressive language** that can formalize common-sense domain rules and handle requirements that RDFS cannot address, such as specifying that two classes are mutually exclusive

Technically, OWL builds upon RDFS by adopting its hierarchical constructs and **introducing many additional constructs**, all of which are represented as RDF triples within the knowledge graph

WEB ONTOLOGY LANGUAGE (OWL)

The *owl:equivalentClass* predicate is used to state that two classes have the **exact same set of individuals as members**. Similarly, *owl:equivalentProperty* is used to define that two property names are semantically identical.

The *owl:disjointWith* axiom is used to declare that two classes are **mutually exclusive**, meaning no individual can be a member of both classes simultaneously.

OWL include also **construct working on properties**. For example:

- *owl:TransitiveProperty* allows for **chains of relations** to be inferred (A is in B, B is in C -> A is in C)
- *owl:SymmetricProperty* ensures that **bidirectional relations** (like "isAdjacentTo")
- *owl:inverseOf* links two properties as **opposites** (e.g., "hasRoadSegment" and "belongsToRoad")

These axioms provide the "glue" that allows machines to navigate and understand complex relationship networks

Class Axioms	rdfs:subClassOf	Defines a hierarchical relationship where every instance of a subclass is also an instance of the superclass.
	owl:equivalentClass	Declares that two class descriptions have exactly the same set of individuals as members.
	owl:disjointWith	States that two classes are mutually exclusive; no individual can belong to both simultaneously.
	owl:AllDisjointClasses	A concise way to declare that a large set of classes are all pairwise disjoint.
Property Axioms	rdfs:subPropertyOf	Establishes a hierarchy for properties, where one relation is a more specific type of another.
	owl:equivalentProperty	Declares that two different property names are semantically identical.
	owl:inverseOf	Defines two properties as opposites (e.g., hasChild is the inverse of hasParent).
	rdfs:domain	Restricts the class of individuals that can act as the **subject** of a property.
	rdfs:range	Restricts the class of individuals or data types that can act as the **object** of a property.
Property Characteristics	owl:TransitiveProperty	If A is related to B, and B to C, then A is automatically inferred to be related to C.
	owl:SymmetricProperty	Defines a bidirectional relationship; if A is related to B, then B is related to A.
	owl:AsymmetricProperty	Explicitly forbids a bidirectional relationship, enabling consistency checks for logic violations.
	owl:FunctionalProperty	Constrains a property so an individual can be related to at most one unique value through it.
	owl:InverseFunctionalProperty	States that the inverse of the property is functional (often used for unique identifiers).
	owl:ReflexiveProperty	States that every individual is related to itself via this property.
	owl:IrreflexiveProperty	Forbids an individual from being related to itself via this property.
	owl:propertyChainAxiom	Allows defining a property based on a sequence of other properties (e.g., parent's parent is grandparent).

Identity Axioms	owl:sameAs	States that two different URIs refer to the exact same real-world individual
	owl:differentFrom	Explicitly declares that two individuals are distinct to prevent incorrect reasoning merges.
	owl:AllDifferent	Declares that all members of a specific set of individuals are distinct from one another.
Restrictions & Constructors	owl:someValuesFrom	**Existential Restriction:** An individual must have at least one relationship of this type to a specific class.
	owl:allValuesFrom	**Universal Restriction:** All relationships of this type for an individual must point to members of a specific class.
	owl:hasValue	Restricts a class to individuals that are linked to a specific, unique individual or literal value.
	owl:cardinality	Specifies the exact number of relationships an individual must have via a specific property.
	owl:intersectionOf	Defines a class as the logical **AND** (overlap) of multiple other classes.
	owl:unionOf	Defines a class as the logical **OR** (combination) of multiple other classes.
	owl:complementOf	Defines a class containing everything that is **NOT** a member of the specified class.

RDF vs RDFS vs OWL

RDF (Resource Description Framework)
- fundamental data model
- based on a triples (*subject*, *predicate*, *object*)
- graph model
- no inherent semantics
- no logical inference

RDFS (RDF Schema)
- provide a basic vocabulary to structure RDF data
- introduces classes (*rdfs:Class*) and hierarchies (*rdfs:subClassOf, rdfs:subPropertyOf)*
- specify subjects (*rdfs:domain*) and objects (*rdfs:range*) of a given property
- Basic Inference Semantics

OWL (Web Ontology Language)
- represent knowledge in a rich and complex way
- based on Descriptive Logics
- constraint declaration
  - disjunction,
  - cardinality
  - property characteristics
- can infer new knowledge
- can verify the logical coherence

The Graph

The Hierarchy

Logic Rigor

ONTOLOGY REUSE CHALLENGES

The Fragmentation Paradox: Persistent issue of creating new ontologies instead of reusing or matching existing ones

Inaccessibility of Sources: Many research articles fail to share source files openly, hindering replicability

Task-Specific Narrowness: Ontologies created for single industrial tasks (e.g., one specific machine) are often difficult to generalize to a city domain

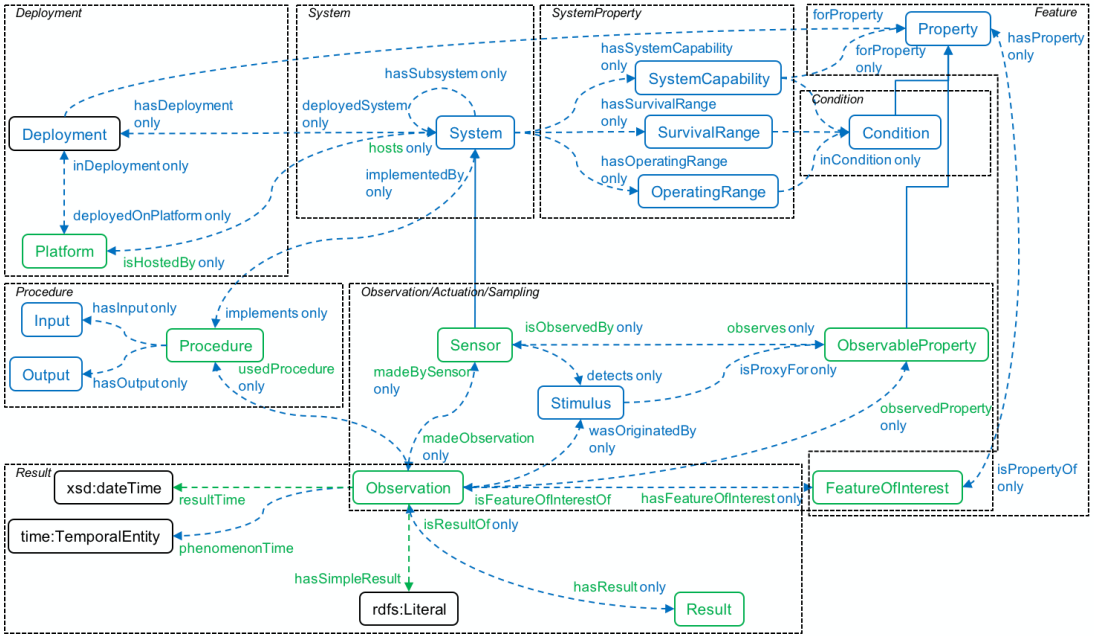
Call for Open Repositories: Necessity of sharing OWL/RDF files in public databases to foster SCDT evolution

FOUNDATIONAL ONTOLOGIES FOR URBAN DIGITAL TWINS

SSN / SOSA

Models:

- Sensors
- Observations
- Measurements
- Actuators



Credit: https://www.w3.org/TR/vocab-ssn/

FOUNDATIONAL ONTOLOGIES FOR URBAN DIGITAL TWINS

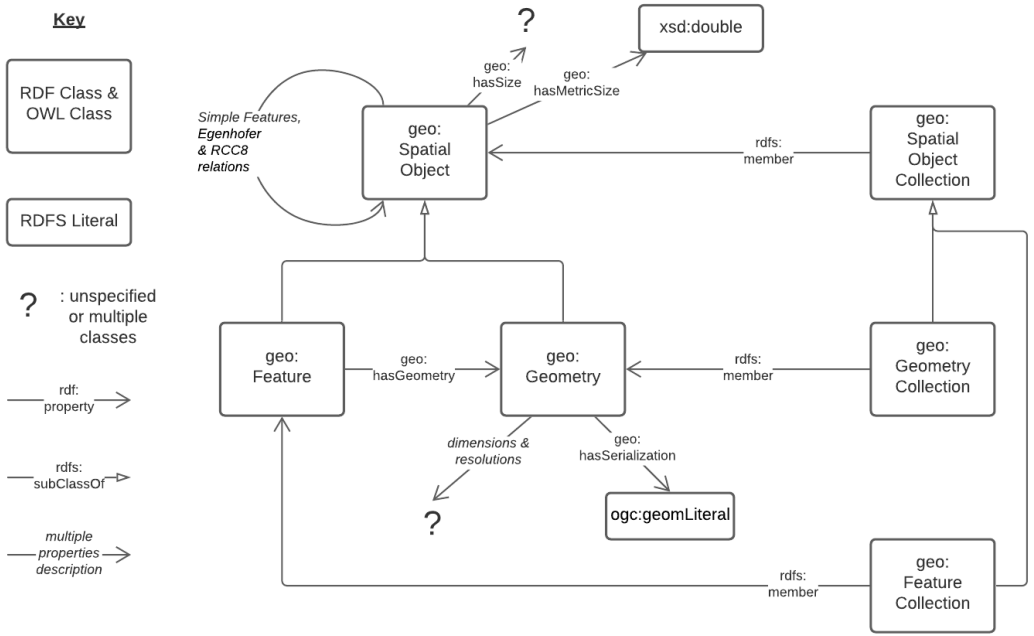
GeoSPARQL

Models:

- Geometry
- Spatial relationships
- Topological predicates

Examples:

- intersects
- within
- contains
- overlaps



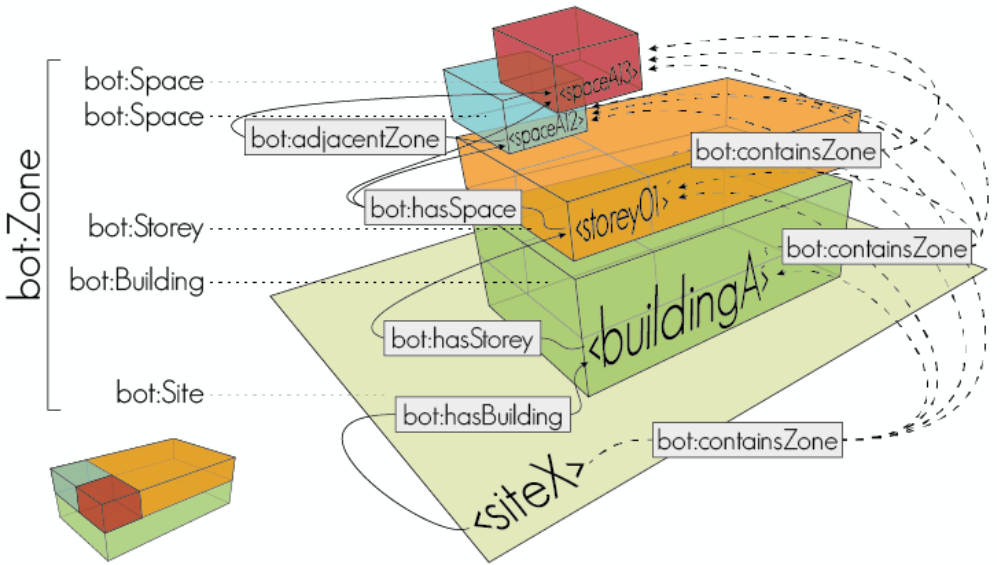
Credit: https://docs.ogc.org/is/22-047r1/22-047r1.html

FOUNDATIONAL ONTOLOGIES FOR URBAN DIGITAL TWINS

Building Topology Ontology (BOT)

Models:

- Zones
- Sites
- Buildings
- Structural elements



Credit: https://w3c-lbd-cg.github.io/bot/

KM4City

The km4city knowledge model enables the description of smart cities, leveraging interconnection, storage and interrogation of data from many different sources.

Developed to support:

- Smart Cities
- Urban Digital Twins
- Data interoperability
- Decision support systems

KM4City connects:

- Buildings
- Roads
- Public transport
- Sensors and IoT devices
- Energy systems
- Environmental data
- Administrative services
- Points of interest

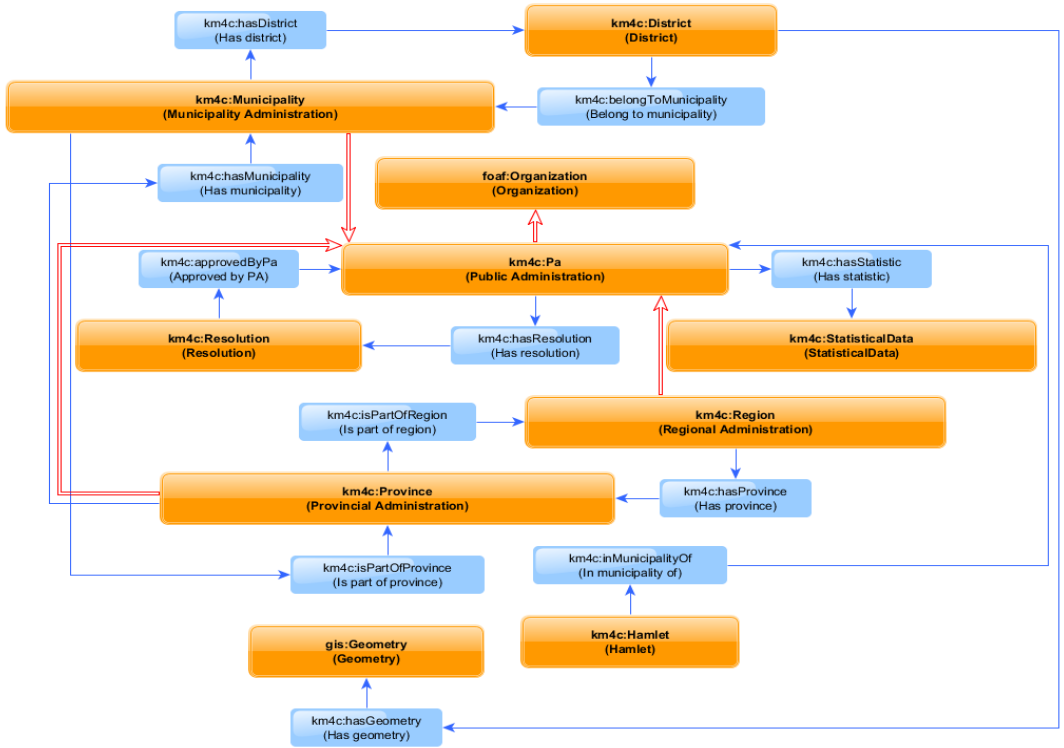
Organized in 7 macro-areas:

- Administration
- Street Guide
- Points of Interest
- Local Public Transport
- Sensors
- Temporal
- Metadata

http://wlode.disit.org/WLODE/extract?url=http://www.disit.org/km4city/schema

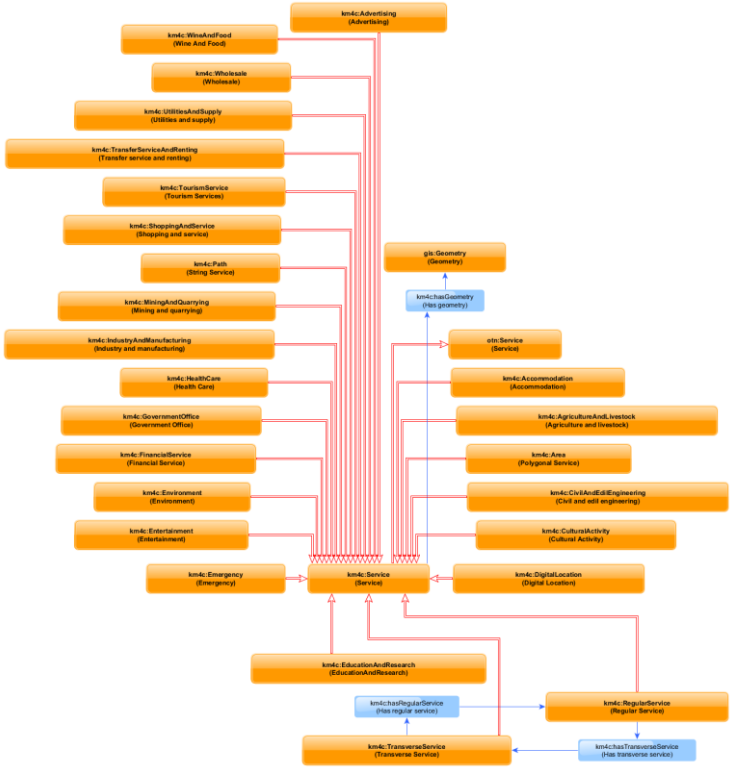
KM4CITY ONTOLOGY: PUBLIC ADMINISTRATION

- Organization
- PA
- Region
- Province
- Municipality
- District
- Resolution
- StatisticalData
- Geometry



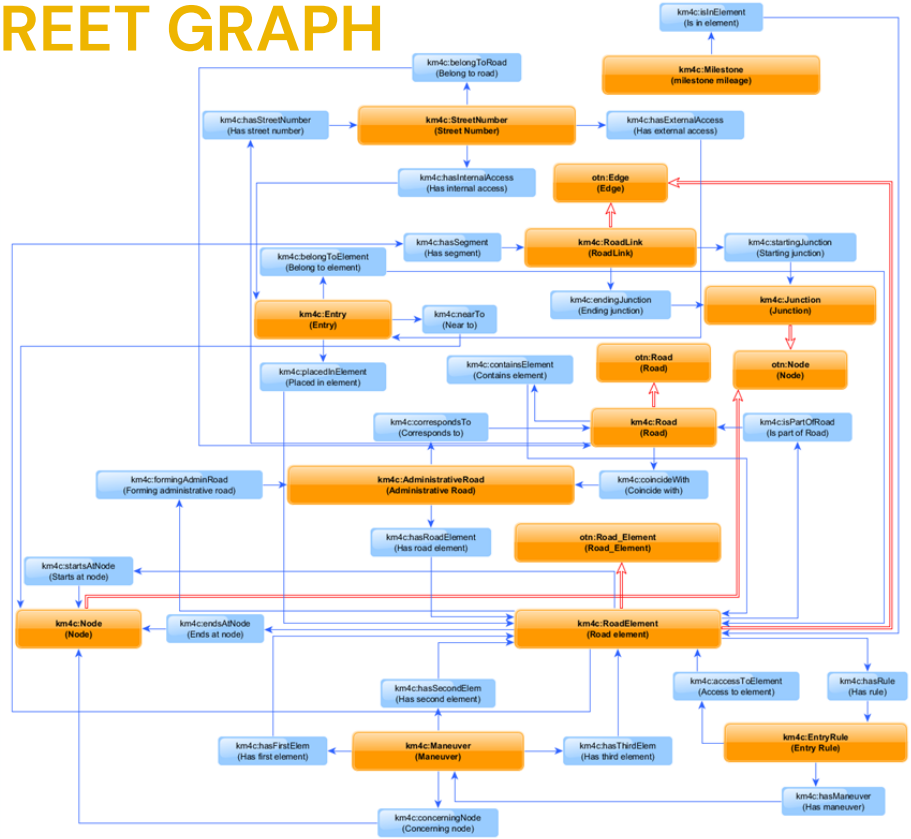
KM4CITY ONTOLOGY: POINTS OF INTEREST

- Service
  - RegularService
    - WineAndFood
    - Path
    - HealthCare
    - GovernmentOffice
    - Entertainment
    - Accommodation
    - …
  - TransverseService



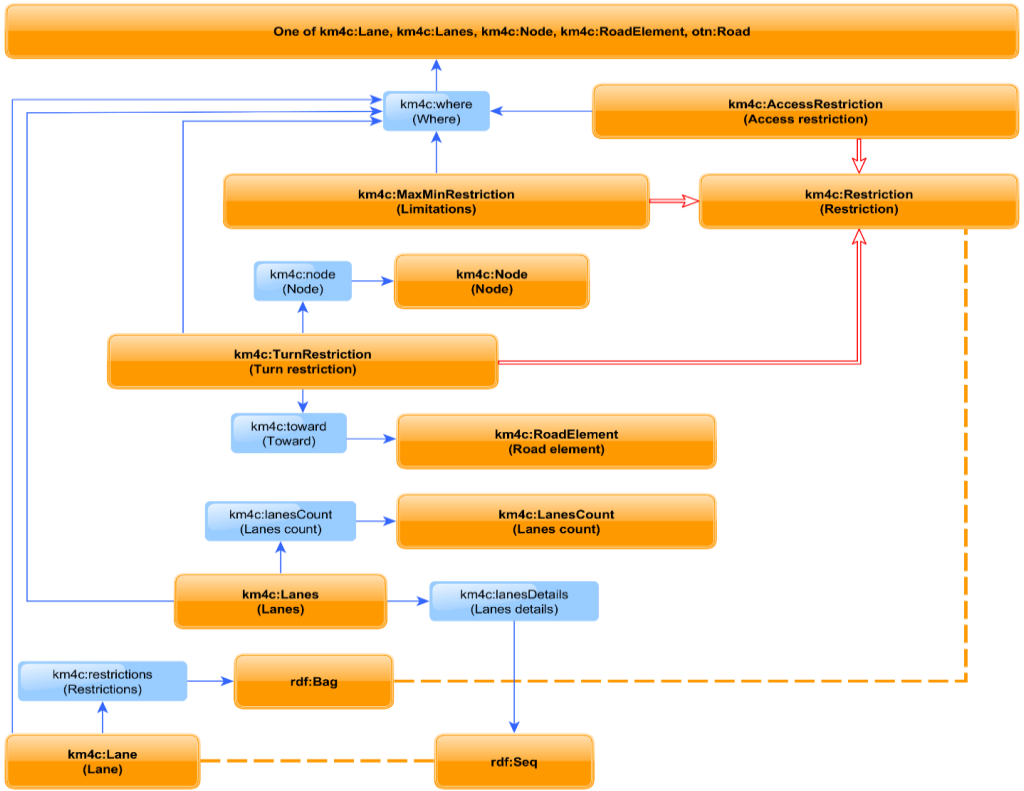
KM4CITY ONTOLOGY: STREET GRAPH

- Road
- AdministrativeRoad
- RoadElement
- Node
- StreetNumber
- Entry
- …



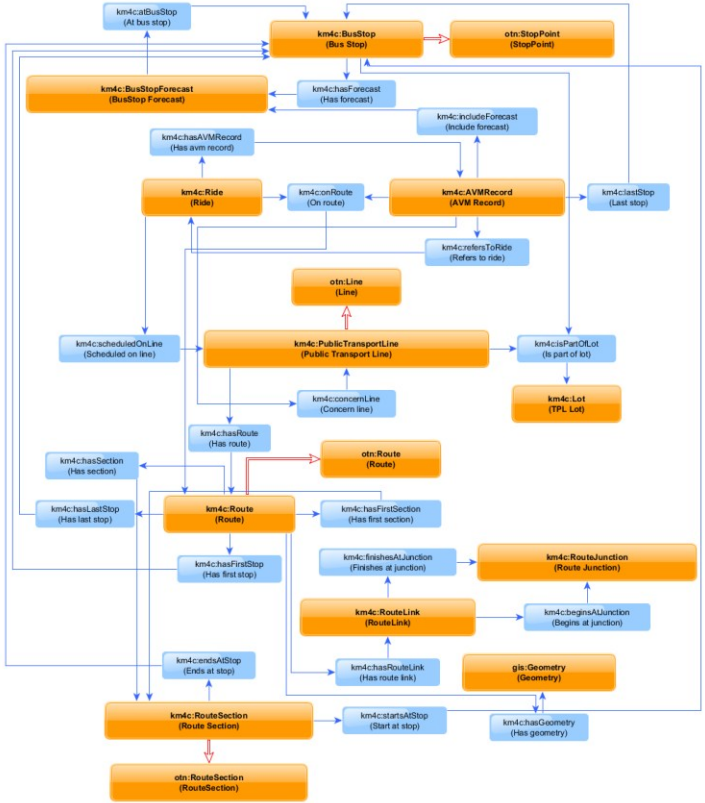
KM4CITY ONTOLOGY: STREET GRAPH

- Lanes
  - LanesCount
  - Lane
- Restriction
  - AccessRestrictions
  - TurnRestrictions
  - MaxMinRestrictions
- A Lane *can* have a ***Bag*** of Restriction
- A Lanes res. can have a **Seq** of Lane



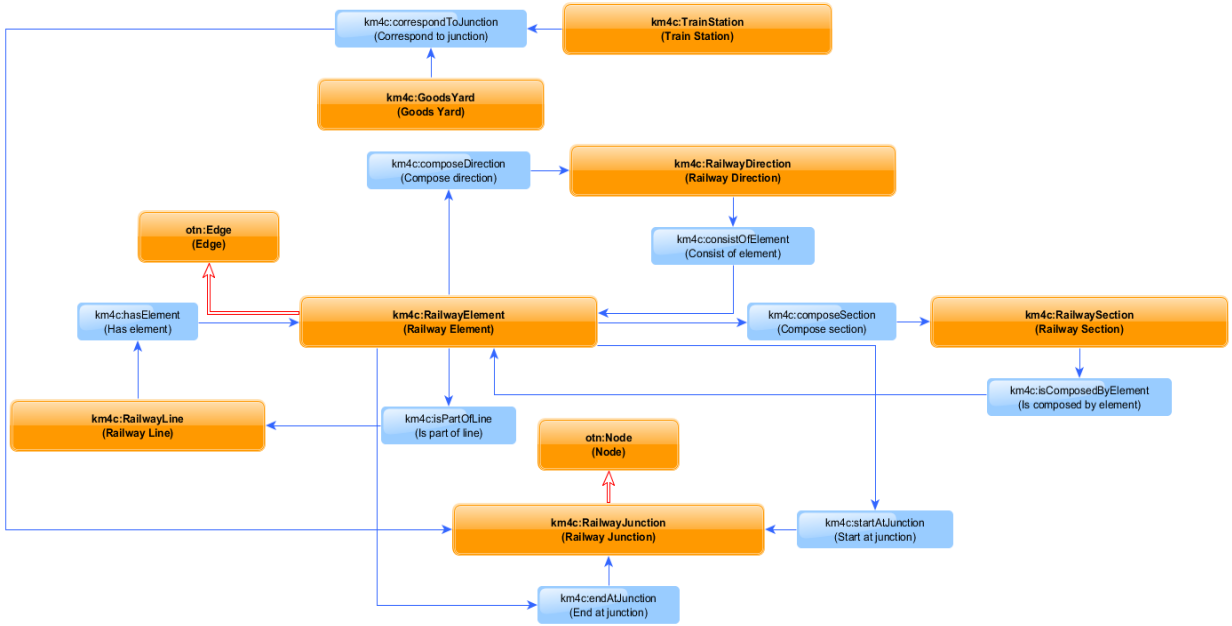
KM4CITY ONTOLOGY: PUBLIC TRANSPORT

- PublicTransportLine
- Route
- RouteSection
- RouteLink
- RouteJunction
- Ride
- BusStop
- AVMRecord
- BusStopForecast
- Geometry



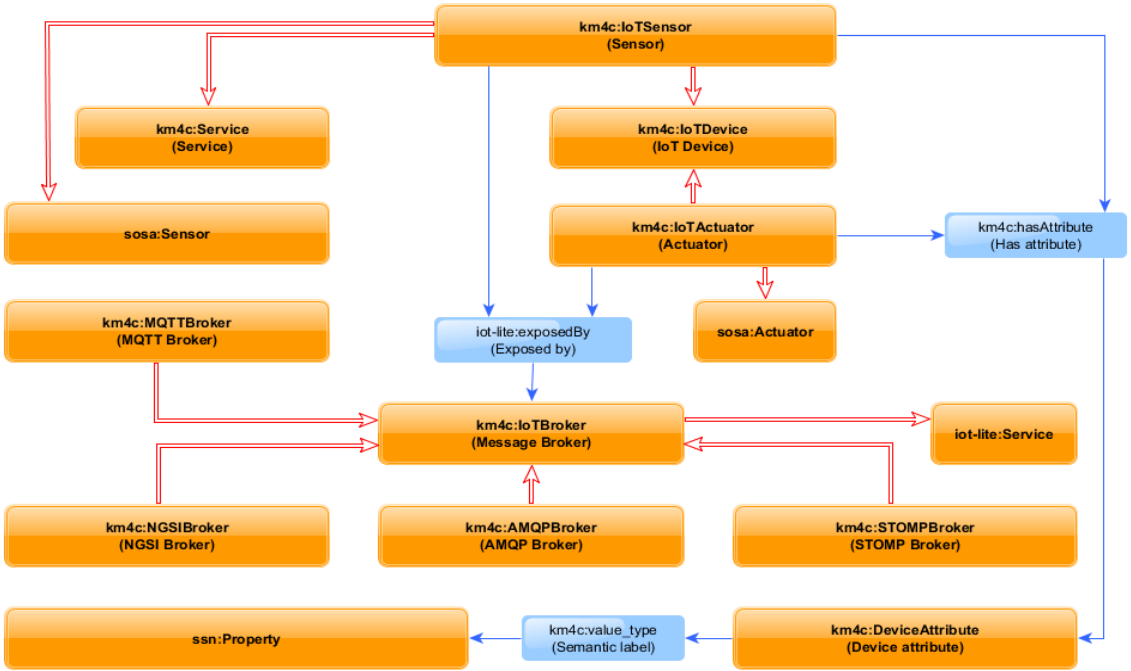
KM4CITY ONTOLOGY: RAILWAYS

- RailwayLine
- RailwaySection
- RailwayElement
- RailwayJunction
- TrainStation
- GoodsYard



KM4CITY ONTOLOGY: INTERNET OF THINGS

- IoTDevice
  - IoTSensor
  - IoTActuator
- DeviceAttribute
- IoTBroker
  - MQTTBroker
  - NGSIBroker
  - AMQPBroker
  - STOMPBroker



STORING KNOWLEDGE GRAPHS

Triple Stores as Knowledge Bases: Knowledge Graphs are typically stored in specialized RDF databases called **Triple Stores** that provide the semantic backbone of ontology-driven Digital Twins

Triple stores manage
- Buildings and infrastructures
- Transportation networks
- IoT devices and observations
- Public services
- Energy and environmental data

Example Technologies
- Virtuoso
- GraphDB
- Blazegraph
- RDF4J

Key capabilities
- Semantic indexing
- Spatial queries
- Temporal queries
- Relationship traversal

QUERYING THE DIGITAL TWIN

From Data Retrieval to Knowledge Retrieval

- Traditional databases answer: "What data are stored?"
- Knowledge Graphs answer: "How are urban entities related?"

Example:

Find public buildings monitored by temperature sensors with energy consumption above threshold

Graph traversal: [Building] --[monitoredBy]--> [TemperatureSensor] --[generates]--> [Observation]

Semantic Queries to search for complex multi-domain relations

Graph Databases navigation to retrieve urban knowledge

Complex Filtering across spatial, temporal, and semantic boundaries

SPARQL: SQL FOR KNOWLEDGE GRAPHS

SPARQL (SPARQL Protocol and RDF Query Language) serves as the standardized semantic query language for the Semantic Web, acting as a counterpart to SQL for relational databases

Its primary purpose is to enable users and software agents to **retrieve and manipulate data stored in RDF format** across various distributed Knowledge Graphs

```
PREFIX km4c: http://...

SELECT ?building WHERE {
  ?building km4c:monitoredBy ?sensor .
}
```

SPARQL: SQL FOR KNOWLEDGE GRAPHS

The core of a SPARQL query is the "Graph Pattern," i.e., a set of triple patterns

Resemble RDF triples but **incorporate variables** (e.g., ?f) to represent unknown elements

Typically includes a **SELECT** clause to specify which variables to return and a **WHERE** clause containing the triple patterns to be matched against the dataset

```
PREFIX km4c: http://...

SELECT ?building WHERE {
  ?building km4c:monitoredBy ?sensor .
}
```

SPARQL: SQL FOR KNOWLEDGE GRAPHS

Prefixes are used to indicate the reference ontology schemas.

Entities can be associated to types using rdf:type.

FILTER allows for the application of constraints on **variable values.**

```
PREFIX km4c: http://...
PREFIX rdf: http://...

SELECT ?building WHERE {
  ?building rdf:type km4c:Building .
  ?sensor rdf:type km4c:TemperatureSensor .
  ?building km4c:monitoredBy ?sensor .
  ?sensor km4c:generates ?observation .
  FILTER(?observation > 30)
}
```

REASONING IN KNOWLEDGE GRAPHS

Reasoning derives new knowledge from existing facts via **inference** without the need of explicitly storing new triples.

```
[Building_123] --[locatedIn]--> [District_A]

[District_A] --[underAlert]--> [Heatwave]
```



```
[Building_123] --[affectedBy]--> [Heatwave]
```

Inference requires **rules** or **ontological axioms**

KG REASONING TASKS

Triple Classification: (h,r,t) True or False?

Relation Prediction: Given h,t find r (among selected candidates)

Entity (Link) Prediction: $(h,r,?)$, i.e., find the tail entity given head and relation

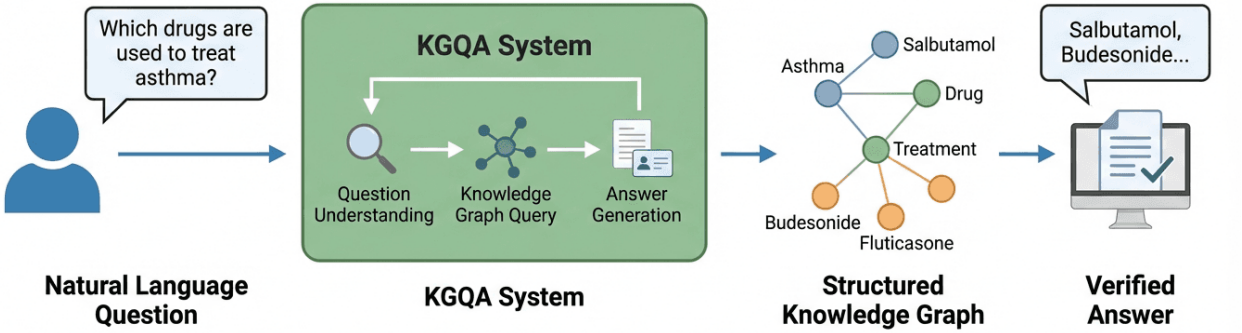
Addressed via

- Symbolic reasoning → Inference using rules and formal logic.
- KB embeddings (TransE, GNN, etc.) → inference through learned vector representations.
- NeuroSymbolic reasoning (LNN, LTN, etc.) → Integration of symbolic logic and neural networks.
- Large Language Models (BERT, LLaMA, GPT, etc.) → use of linguistic models to interpret, interrogate and reason on structured knowledge.

KNOWLEDGE GRAPH QUESTION ANSWERING

KGQA is defined as the task of answering **natural language questions** by grounding the reasoning process in the information stored within a **structured KG**

The paradigm aims to **bridge the gap** between human linguistic intent (the question) and the machine-readable, verified facts available in a KG, facilitating cooperation between **human users** and **software agents**



DATA MODELS

Corso di: Big Data Architectures

Dott. Marco Fanfani

DISIT Lab, Sistemi Distribuiti e Tecnologie Internet

http://www.disit.dinfo.unifi.it/

Dipartimento di Ingegneria dell'Informazione

Università degli Studi di Firenze

CC BY SA

Internet of Things

Internet of Things (IoT) refers to networks of physical devices that collect data, exchange it, and support services in digital environments.

IoT systems connect the physical and digital worlds by linking devices, applications, and communication infrastructure.

Internet of Things (IoT) is a **distributed cyber-physical system** where:

- physical devices observe the environment
- digital systems interpret the observations
- actions can be triggered back into the physical world

Smart City Example

A single *smart street* may include:

- air quality sensors (PM10, NO_2)
- traffic counters
- smart lamps
- weather stations
- parking sensors

Each device is independent, but all contribute to a **shared urban state**.

SENSORS AND ACTUATORS

Sensors observe the physical environment and generate data such as temperature, motion, presence, or air quality.

Actuators receive commands and change something in the physical world, such as opening a valve or switching a light.

IoT systems are therefore not only about monitoring; they also support **action** and **control.**

Closed loop:

Sensor → Data → System → Decision → Actuator → Physical change → Sensor

WHY BROKERS

When many devices and applications must interact, direct point-to-point communication becomes hard to manage

- every device must know every application
- tight coupling
- no scalability
- no reuse

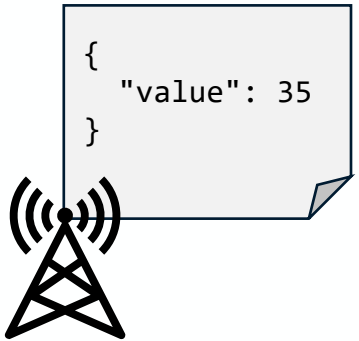
A **broker** acts as an intermediary that **decouples data producers** from **data consumers** and supports publish/subscribe communication.

In IoT, this makes integration more **scalable** and **easier to reuse** across applications.

FROM BROKERS TO CONTEXT BROKERS

A message broker only forwards messages. It does NOT know:

- what a sensor is
- where it is
- what it measures



Of limited use without:

- identity
- location
- time
- meaning

Context transforms data into something that can be interpreted and reused.

CONTEXT BROKERS

A **Context Broker** is a system that stores, updates, queries, and distributes **context entities**, not messages.

For example, an **entity** is:

- an object in the city
- with attributes
- with relationships

Entity: **Air Quality Station**

AirSensor_12
locatedIn: Milan_Center
PM10: 78
status: critical

A context broker manages the life cycle of context information, including **updates**, **queries**, **registrations**, and **subscriptions**.

It maintains and distributes the **updated status** of real entities (sensors, buildings, traffic, air, energy…).

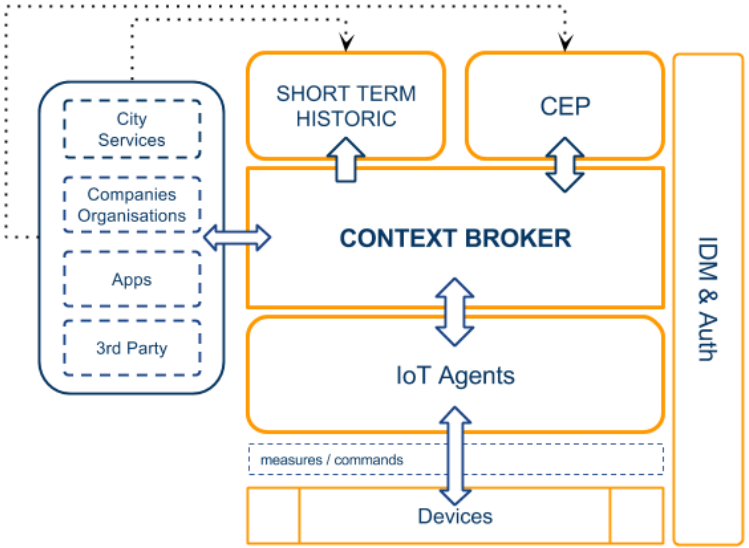
CONTEXT BROKER ARCHITECTURE

The **Context Broker** acts as the "Central Nervous System" of a digital twin ecosystem, providing a standard API for context management.

Decoupling: Separates data producers (IoT, crawlers) from consumers (Apps, Dashboards).

Pub/Sub: Notifies applications of changes in real-time.

State Preservation: Maintains the "Last Known State" (Context) of any entity.



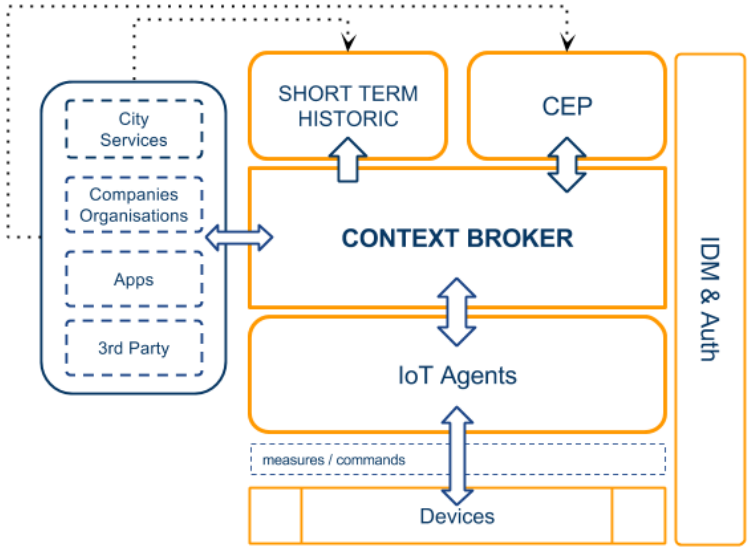
CONTEXT BROKER ARCHITECTURE

The **Context Broker** acts as the "Central Nervous System" of a digital twin ecosystem, providing a standard API for context management.

Decoupling: Separates data producers (IoT, crawlers) from consumers (Apps, Dashboards).

Pub/Sub: Notifies applications of changes in real-time.

State Preservation: Maintains the "Last Known State" (Context) of any entity.



SMART DATA MODELS

In a real-world environment, data is generated by a very large number of heterogeneous sources: IoT sensors, mobility platforms, building management systems, etc.

A standard formats (i.e., JSON) is not sufficient to guarantee interoperability: each system uses its own *interpretation* of the data.

A temperature value might be called "temp", "temperature", or simply "t", and may or may not include units or metadata.

Systems can technically exchange data but cannot reliably understand without custom integration logic: **semantic consensus is required**!

Smart Data Models address this problem by introducing a **shared semantic agreement on how data should be described**.

SMART DATA MODELS

Smart Data Models are **reusable** and **standardized information models** for distributed applications.

They define:

- entity types (e.g., Building, Parking, AirQualityObserved)
- attributes and units
- relationships
- examples and validation rules

The goal is to allow different platforms to **use the same semantic definitions**.

INFORMATION MODEL

- An **information model** defines how a specific domain should be structured and interpreted.
- It does not describe a single dataset, but a **template for all possible data instances**.

The model defines the **structure of knowledge**, not the data itself.

NGSI-v2

NGSIv2 is an API specification that defines a **standard information model** and a **set of REST operations** for context management.

It uses **RESTful APIs and JSON** for *create*, *read*, *update*, *delete*, *query*, and *subscription* operations.

The context data model based on:

Entity Model

The primary unit of information. Identified by an id and a type. Examples: Room1, Sensor_A.

Attributes

Properties of an entity. They have a type and a value. NGSI-v2 follows a strict Key-Value structure.

Metadata

Optional data *about* the attribute (e.g., accuracy, unitCode, or timestamp).

NGSI-v2: Operational Example

Operation	Endpoint & Payload	Description

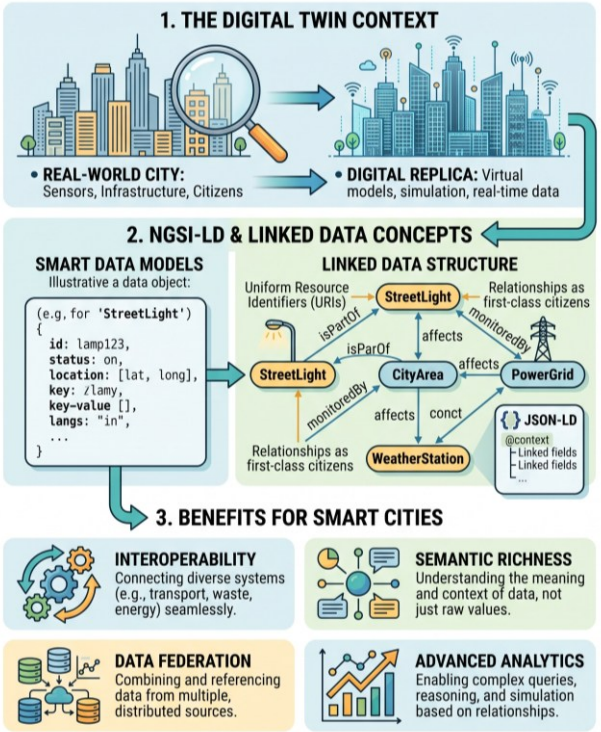
Create Entity	`POST /v2/entities`
	`{`
	`  "id": "urn:ngsi-v2:Room1",`
	`  "type": "Room",`
	`  "temperature": {`
	`  "value": 23.5,`
	`  "type": "Number"`
	`  }`
	`}`
		Instantiates a new context entity in the broker memory.

Get Value	`GET /v2/entities/urn:ngsi-v2:Room1/attrs/temperature/value`	Retrieves only the raw value (e.g., `23.5`) for fast processing.
Subscribe	`POST /v2/subscriptions`	Sets a trigger: notify subscriber when update `temperature`

EVOLUTION: THE MOVE TO NGSI-LD

NGSI-LD is an evolution of NGSI-v2, formalized by ETSI to **incorporate Linked Data** into real-time context management.

NGSI-LD evolves the flat model of v2 into a **Property Graph** using **JSON-LD**.



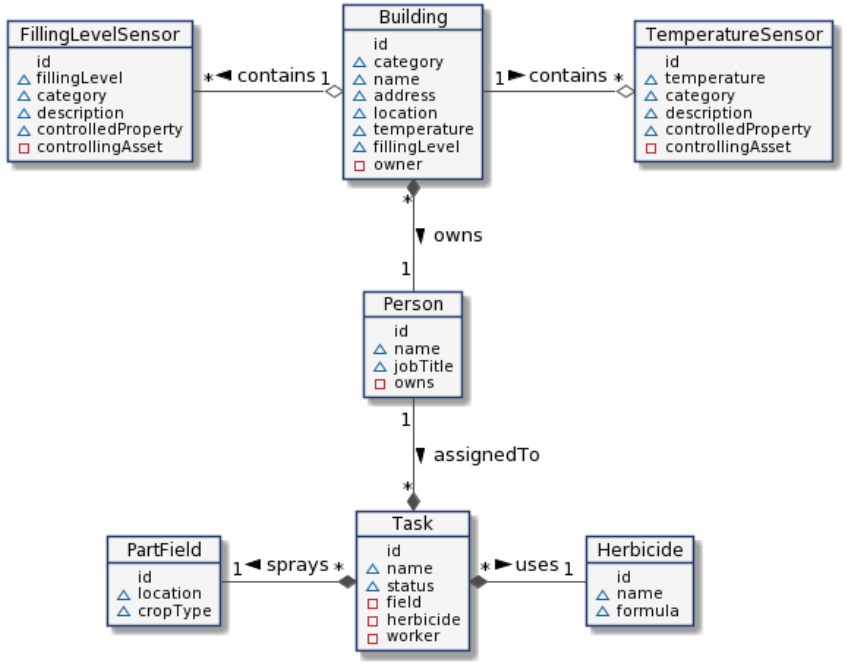
Linked Data

They represent the operational arm and set of rules of thumb for publishing and connecting structured data on the web. If the Semantic Web is the final goal, Linked Data is the **methodology and standards (such as RDF and URI)** used to concretely build that network.

NGSI-LD: Information Model

NGSI-LD transitions from flat Key-Value pairs to a formal **Property Graph** model:

- **Entities**: Represent real-world objects.
- **Properties**: Attributes representing state (values).
- **Relationships**: Directed links between entities (objects).
- **Property of Property**: Rich metadata as nested attributes.



JSON-LD

JSON-LD (JavaScript Object Notation for Linked Data) is a lightweight format used to transport linked data using the syntax of classical JSON

Attribute @context

```
{
  "@context": {
      "temperature": "http://schema.org/temperature",
      "Celsius": "http://vocab.nerc.ac.uk/collection/P06/current/UPAA/"
  },
  "temperature": 22.5
}
```

With `@context` the local key `"temperature"` is mapped to a global URI. Any system will know exactly what that data means without the need for prior agreements.

JSON-LD

The @context

A mapping file that translates short, human-readable terms into **Unique URIs** to prevent semantic ambiguity.

Unique IDs

Every entity must have an URN or URL as its ID, ensuring it is globally referenceable across systems.

Interoperability

By using standard ontologies (e.g., Schema.org), data becomes understandable by any machine without prior configuration.

NGSI-v2 vs. NGSI-LD

Feature	NGSI-v2 (JSON)	NGSI-LD (JSON-LD)
Primary Model	Entity-Attribute (Flat)	Property Graph (Relationships)
Semantics	Implicit / Custom	Explicit / Formal (URIs)
Entity IDs	Strings (Simple)	URIs (URN/URL) - Mandatory
Metadata	Metadata Object	Nested Properties / Relationships

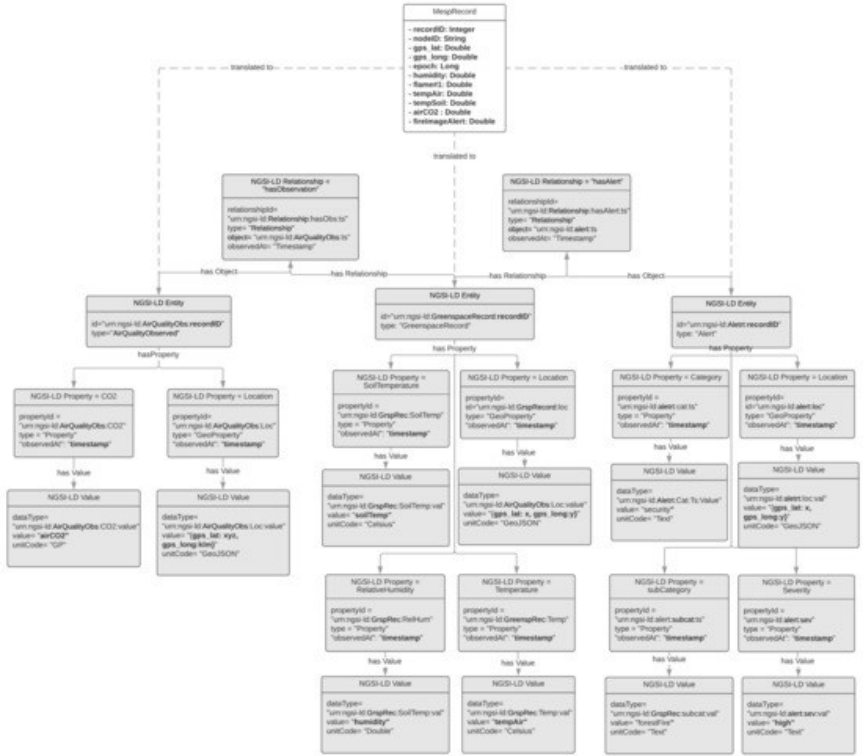
NGSI-LD: Built-in Technical Attributes

observedAt: Standardized timestamp for temporal context. Essential for historical trend analysis.

unitCode: Formalized UN/CEFACT codes for measurement units (e.g., CEL for Celsius).

datasetId: Allows multiple values for the same property if coming from different sources/sensors.

instanceId: Unique identifier for each specific observation or update event.



NGSI-LD: Built-in Technical Attributes

datasetId: Allows multiple values for the same property if coming from different sources/sensors.

Identify the instance of the value (its source).

E.g.: two speed values for the same vehicle

- speed = 50 → comes from GPS
- speed = 48 → comes from speedometer

```
{
  "id": "urn:ngsi-ld:Vehicle:123",
  "type": "Vehicle",
  "speed": [
    {
      "type": "Property",
      "value": 50,
      "datasetId": "urn:ngsi-ld:dataset:GPS"
    },
    {
      "type": "Property",
      "value": 48,
      "datasetId": "urn:ngsi-ld:dataset:Speedometer"
    }
  ]
}
```

NGSI-LD: Built-in Technical Attributes

instanceId: Unique identifier for each specific observation or update event.

Identifies that **specific occurrence** of the Property, especially in the temporal representation.

Useful to selectively modify/delete a specific time information via NGSI-LD API.

```
{
  "id": "urn:ngsi-ld:Sensor:123",
  "type": "Sensor",
  "temperature": [
    {
      "type": "Property",
      "value": 20.1,
      "observedAt": "2026-10-22T10:00:00Z",
      "instanceId": "urn:ngsi-ld:instance:001"
    },
    {
      "type": "Property",
      "value": 20.4,
      "observedAt": "2026-10-22T10:05:00Z",
      "instanceId": "urn:ngsi-ld:instance:002"
    }
  ]
}
```

RELATIONSHIPS

In NGSI-LD, the relationships between entities are made explicit.

Unlike NGSI-v2 (where relationships were often disguised as normal textual attributes or managed implicitly), NGSI-LD **introduces a formal and standardized construct** based on two key concepts:

- the **`Relationship`** type and
- the use of the **`object`** field

```
{
  "id": "urn:ngsi-ld:Sensor:Sensor_01",
  "type": "TemperatureSensor",
  "temperature": {
    "type": "Property",
    "value": 24.5
  },
  "isInstalledIn": {
    "type": "Relationship",
    "object": "urn:ngsi-ld:Room:Room_A"
  },
  "@context": [
    "https://uri.etsi.org/...",
    "https://example.org/context.jsonld"
  ]
}
```

TEMPORAL & GEOSPATIAL QUERIES

NGSI-LD offers native support for complex **spatial** and **temporal** context queries.

Each entity can have a GeoJSON-compliant `location` property.

```
"location": {
  "type": "GeoProperty",
  "value": {
    "type": "Point",
    "coordinates": [11.255, 43.769]
  }
}
```

APIs allow spatial queries such as:

- within
- contains
- intersects
- near
- equals
- disjoint



Each property can have time information such as `observedAt`, `createdAt`, `modifiedAt`



```
"temperature": {
  "type": "Property",
  "value": 28,
  "unitCode": "CEL",
  "observedAt": "2026-06-26T10:30:00Z"
}
```

Then you can make queries such as "latest observation", or "observations between two timestamps"

The problem:

- The Context Broker primarily maintains the current state of the entities. It was not born as a historical database.
- To manage time series, you normally need additional components (e.g., Cygnus / QuantumLeap / TimescaleDB).

CORE API OPERATIONS

Context Management

Standard RESTful operations:

- `POST /entities - Create`
- `PATCH /entities/{id}/attrs - Update`
- `GET /entities/{id} - Retrieve`
- `DELETE /entities/{id} - Remove`

```
curl -iX POST http://localhost:1026/ngsi-ld/v1/entities \
 -H 'Content-Type: application/ld+json' \
 -d '{
  "type": "Building", "id": "urn:ngsi-ld:Building:store001",
  "category": { "type": "Property", "value": ["commercial"]},
  "address": {  "type": "Property"," value": {
        "streetAddress": "Bornholmer Straße 65",
        "addressRegion": "Berlin",
        "addressLocality": "Prenzlauer Berg",
        "postalCode": "10439"
     },
     "verified": { "type": "Property", "value": true }
  },
  "location": { "type": "GeoProperty",
     "value": { "type": "Point",  "coordinates": [13.3986, 52.5547]}
  },
  "name": { "type": "Property", "value": "Bösebrücke Einkauf" },
  "@context": [
     "https://fiware.github.io/data-models/context.jsonld",
     "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
  ]
}'
```

IS NGSI-LD A "TRUE" KNOWLEDGE GRAPH?

The NGSI-LD graph is not merely a collection of data; it is **Ontologically Grounded**.

- **URI-based Identity:** Every node (Entity) and edge (Relationship) is mapped to a global URI using the @context.
- **Property Graph Model:** While the underlying storage is a Property Graph, the semantics are inherited from formal ontologies (SAREF, Schema.org).
- **Structural Population:** The graph is built dynamically as JSON-LD payloads are ingested by the Broker.

NGSI-LD vs KNOWLEDGE GRAPH

NGSI-LD is not a KG that can be queried like RDF/SPARQL.

NGSI-LD manages an **operational context graph**, but it is not an RDF triple store and does not expose a SPARQL endpoint. Queries are made via the REST APIs defined by the ETSI standard, with filters on properties, relationships, geometries and time intervals.

*For advanced semantic analysis or inference, it's **common to integrate the Context Broker with a Knowledge Graph** (like KM4City in Snap4City), which offers much richer reasoning and querying capabilities.*

This also clarifies the distinction of roles between the two components within an Urban Digital Twin.

NGSI-LD vs KNOWLEDGE GRAPH

Feature	Context Broker (NGSI-LD)	Semantic KG (RDF/OWL)
Primary Goal	Operational Real-time Performance	Deep Logical Reasoning
Mechanism	**Structural Validation:** Ensures data matches JSON Schemas.	**Semantic Inference:** Computes implicit facts via Reasoners.
Complexity	Predictable latency (O(1) or O(n) lookup)	High complexity (O(EXP) for OWL DL reasoning)
Use Case	High-frequency IoT streams, Digital Twin state	Complex Querying, Knowledge Discovery

A COMPLEMENTARY HYBRID ARCHITECTURE

1. Operational Layer (NGSI-LD)

Manages the **Last Known State**. Handles dynamic relationships and asynchronous notifications to external microservices.



2. Knowledge Layer (Triple Store)

The **Long-term Memory**. Context is "siphoned" into an RDF Database for SPARQL querying and OWL inference.

Seamless Synchronization

Since both layers share the same **Linked Data identifiers (URIs)**, synchronization is virtually cost-free.

- **Subscription-based Sync:** The Triple Store subscribes to the Broker to receive real-time updates.
- **Offline Reasoning:** Inference is performed on the Triple Store without affecting IoT ingestion latency.