Semantics & BI

1) Introduction

Features: Repository of semantically annotated data, information and knowledge.

Import RDFized data from heterogeneous data sources. Sync data sources with operations in the model.

Nodes (peers) deployment architecture loaded with the data from each sources, share an Unified URI space (for aligned resources and triples). Functional API for augmentation, discovery and analysis.

Type (classes) / relationship types, relationship instances / state flows (operations, rules, flows, events) inference over ‘raw’ RDF data from data sources.

Align & Merge. Equivalent resources / triples from different ontologies merged according their meaning. Model by example.

Order inference: inference over the order of events / operations occurred, occurring and that may occur according metadata aggregated from data sources. Enforce use case flows declaratively (model by example)

Uniform API Ports: CRUD bindings to common protocols as REST (HATEOAS / OData), SOAP, LDP (Solid) and others.

Metadata is aggregated decomposing a triple (quad) into their three resources (SPO) and its context. Then a set model is arranged where there is a set for each SPO part, the intersection of the three sets which accounts for the triples itselfs (contexts) and there is another intersection for each of the three SPO combinations.

This last three intersections are regarded as ‘Kinds’, one for each SPO. So, there is a SubjectKind, an ObjectKind and a PredicateKind. Kinds aggregate ‘attributes’ and ‘values’ (see diagrams below) which account for the description of ‘classes’ and ‘metaclasses’. Related attributes describe related classes and related values describe related metaclasses. So, for example, a SubjectKind aggregates Predicates and Objects as attributes and values respectively.

2) Datasources

Datasources are plugeable into the architecture. There must be a couple of interfaces implemented for each to build some kind of ‘driver’ which allows for syncing the changes made in the models into the original source while being able to retrieve their up-to-date contents as RDF.

Relational databases (JDBC compliant via D2RQ), EIS (CRMs, ERPs) and other common applications / protocols should have drivers for the system.

3) Inference

a) Set predicates

Knowledge aggregated in models should be capable of being abstracted in such a way that general knowledge may be obtained from specific knowledge. Richer query / browsing and inference capabilities should arise from such schema.

```
Data: [someNewsArticle] [subject] [climateChange]
Information: [someMedia] [names] [ecology]
Knowledge: [mention] [mentions] [mentionable]
```

Set oriented approach implementation:

In order to achieve the set oriented abstractions needed (set representation of triples, set models and metamodels) a following (pseudo) API should be implemented:

TripleLoader: Instantiates Resources (SPO, Kinds, Triples) from input RDF.

KindsAggregator: Aggregates and calculates Kind class / metaclass ID URIs. Assignates class / metaclass to SPO resources. Reifies Kinds.

TripleAggregator: Prepares triples from this metamodel layer level as input for an (eventual) next level aggregate.

Model: The model itself (an instance of a metamodel level). Contains set functional arrangements and dimensional arrangements (see below). Base entry point for services API.

Models of data, information, knowledge are relative to their positions respect to other models.

Set: Basic set class. Defined by one set Predicate. Basic set union, intersection, complement operations.

Resource: Superclass of all Set elements. SPO. Monadic wrapper for functional APIs (see below)

Predicate: (Set) Predicate. Holds for a Resource belonging to one Set.

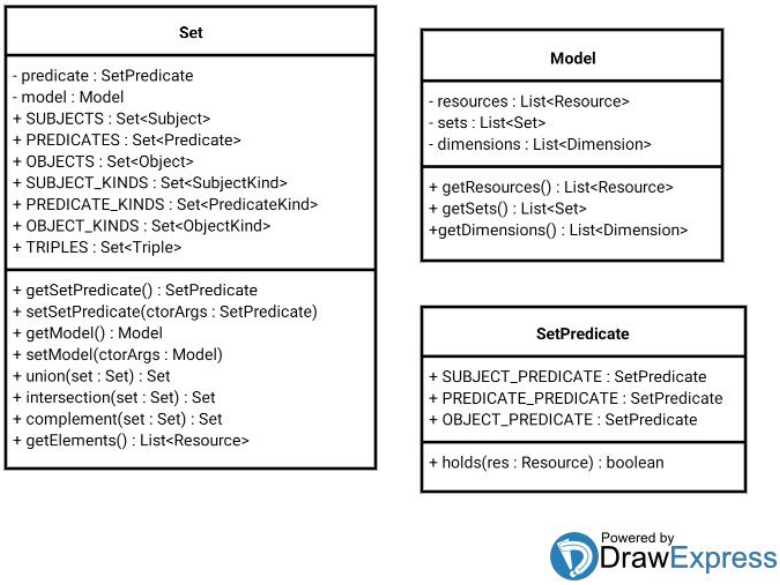
Subject Predicate def: Subject, as Resource appears as Subject in Resources. Is S (& is not P & is not O)?.

SubjectKind Predicate def: attr/val exists at same time in Resources. PO preds holds for subj kind. Is P & is O & Is not S.

Triple Predicate def: SPOs of Triple Resources holds for all SPO preds. Is S & is P & is O.

Hierarchies: when Kinds classes / metaclasses represents some hierarchy relationship the class / metaclass URI ID of them somehow renders this relationship. Then the Kind itself is reified (as an S, P or O) representing this hierarchy's top and aggregating its instances.

Triple Resource Predicate: Context occurs. Context in each metamodel has a meaning and triples sharing the same quad context share meaning being this a temporal, order or causal relationship, among others.



b) Metamodels

i) Kinds aggregation semantics

By the virtue of some resources sharing related ‘attributes’ and ‘values’ according their occurrences in multiple triples (for example, given Subjects having related Predicates and Objects) a ‘Kind’ relationship could be stated of those resources and their types.

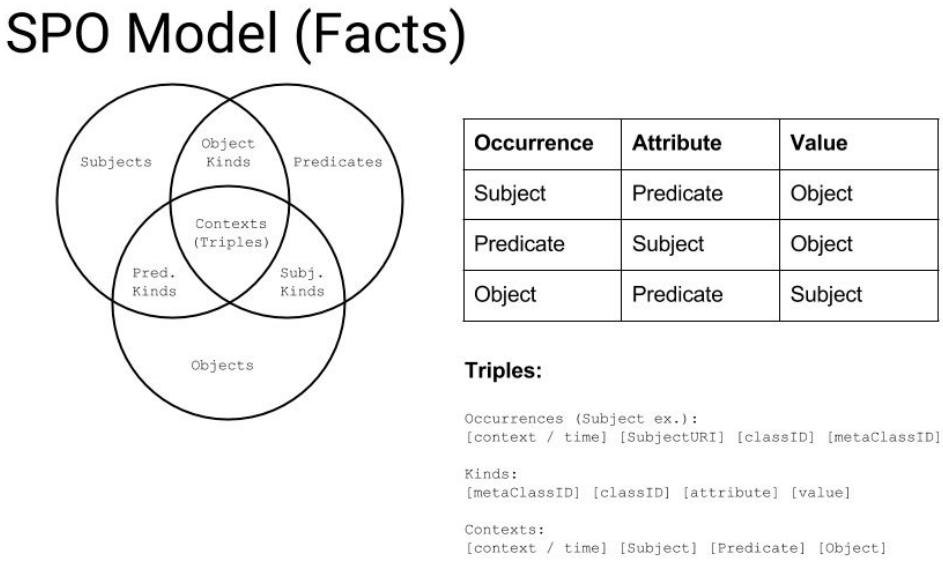
Kinds aggregate classes / metaclasses hierarchies given these attributes and values encoded in class / metaclass URI IDs resources. One example could be all the Subjects that ‘worksAt’ and all resources that ‘workAt’ ‘XYZ Corp.’.

ii) Layers

Models aggregate knowledge in such a way abstractions can be made from source statements and allows for richer query capabilities as in the example triples:

```
Data: :someNewsArticle :subject :climateChange
Info.: :someMedia :names :ecology
Knowl.: :mention :mentions :mentionable
```

iii) Data



Occurrence	Attribute	Value
Subject	Predicate	Object
Predicate	Subject	Object
Object	Predicate	Subject

Triples:

```
Occurrences (Subject ex.):
[context / time] [SubjectURI] [classID] [metaClassID]

Kinds:
[metaClassID] [classID] [attribute] [value]

Contexts:
[context / time] [Subject] [Predicate] [Object]
```

Data model level: this model layer is composed of raw data from which information and knowledge models will be built. Data for this layer comes from the raw RDF from the data sources component.

This set arrangement from triples into SPO and Kinds is the same of the remaining models. Triples feed to the forthcoming levels are aggregated into SPO structure aggregating SPOs, Kinds and triples into new statements. An SPO resource in the next layer triples occur with its corresponding Kind in an occurring triple.

Data layer example:

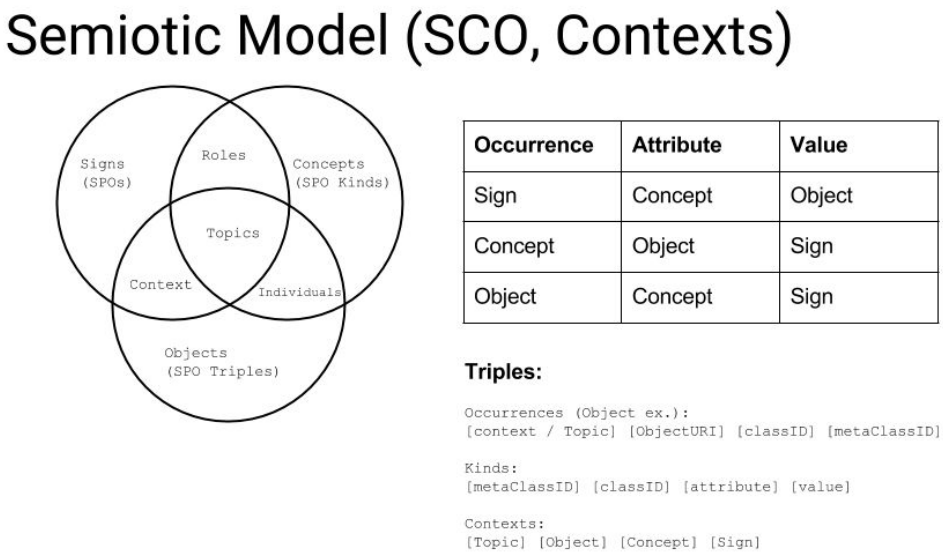
Triple: Peter worksAt XYZ Corp.

Subject / Subject Kind: Peter / Employee

Predicate / Predicate Kind: worksAt / Employment

Object / Object Kind: XYZ Corp / Employeer

iv) Information



Occurrence	Attribute	Value
Sign	Concept	Object
Concept	Object	Sign
Object	Concept	Sign

Triples:

```
Occurrences (Object ex.):
[context / Topic] [ObjectURI] [classID] [metaClassID]

Kinds:
[metaClassID] [classID] [attribute] [value]

Contexts:
[Topic] [Object] [Concept] [Sign]
```

This model layer will aggregate data into information which can be then converted into a knowledge model. This layer is called ‘Semiotic’ because it is concerned with Signs (SPO resources from the previous model), Concepts (Kinds from the previous model) and Objects (Context Triples from the previous model).

Semiotic layer example:

Object / Role: [Peter worksAt XYZ Corp] (SPO Triple) / Role defs. All Roles that apply.

Concept / Context: Employee, Employeer, Employment / Context defs.

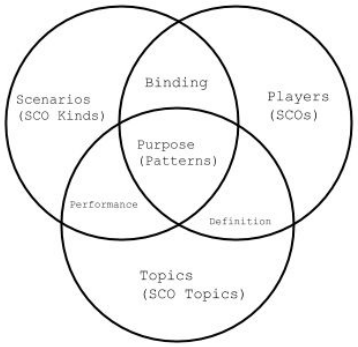
Sign / Individual: Peter, XYZ Corp, worksAt.

Topic: Topic1: hiring. (above SCO triples)

Topics aggregated by same kinds, ordered by SCO contexts (SPO contexts). Order relation examle: hiring, promotion.

v) Knowledge

Behavior Model (TSP)



Occurrence	Attribute	Value
Scenario	Topic	Player
Player	Scenario	Topic
Topic	Scenario	Player

Triples:

```
Occurrences (Topic ex.)
[context / Purpose] [TopicURI] [classID] [metaClassID]

Kinds:
[metaClassID] [classID] [attribute] [value]

Contexts:
[Purpose] [Topic] [Scenario] [Player]
```

This model layer will render knowledge extracted from aggregated triples of the previous data and information layers.

Knowledge layer metamodel example:

Topic / Binding: Topic1

Scenario / Definition: newProject (SCO Context)

Player / Performance: aProject (SCO Concept)

vi) Metamodel example

Metamodel features example:

A of B is C. (S of Triple is Kind)

Peter worksAt XYZ Corp. : EmpTriple

SPO Metamodel:
SubjectKind: Employee(worksAt, XYZ Corp.)
PredicateKind: Employment(Peter, XYZ Corp.)
ObjectKind: Employeer(Peter, WorksAt)

SCO Metamodel:
SignKind: Instance(Employent, EmpTriple)
ConceptKind: Context(Peter, EmpTriple)
ObjectKind: Role(Peter, Employee)

The ‘functional’ notation used above is not casual. It will be used later in APIs designed to build Template(s) or ‘patterns’ which will be the basis for provide a Services interface for interacting with the models.

c) Sets, Monads, Higher order functions

Resources: Higher order functions mappings. Resource monads.

Kinds. Kind monads.

A of B is C. (S of Triple is Kind)

Resources, Kinds: Container, Container Profiles (LDP / Solid).

Bound functions.

Templates are the basic IO messaging method of the service interface which allows for CRUD and state flow (rules, flows, events) manipulation.

Functional Template:
Template : (Kind, Template lhs, Template rhs)

Template example:
Employment(Person((Age, 'young') {&/|} (Sex, 'M')), Business((Size, 'Small')))

Template Promotion:
Person -> Employee (at Employment Kind, due to predicate/object statements addition. Add inst. rel/attrs: salary, position, dept, etc. for class via callbacks / prompts of their values or value Kinds, ie.: high salary).

Functional assertions are bound to order relationships encoded in metamodel triples / quads contexts so, for example, one could query about available templates and possible values regarding this state.

Monads:

interface M<T> : public(T) : M<T>

function unit<T>(val: T) : M<T>

function val<T>(m: M<T>) : T

function bind<T, U>(inst: M<T>, transform: (value: T) => M<U>) : M<U>

ResourceMonad<R extends Resource>(resource : R)
ResourceMonad.Triple
ResourceMonad.Context
ResourceMonad.Subject
ResourceMonad.Predicate
ResourceMonad.Object
ResourceMonad.SubjectKind
ResourceMonad.PredicateKind
ResourceMonad.ObjectKind

Bound functions:
Inference, Triple joins: (S -> Object: Kinds, S -> Concept: Triples, Kind -> Sign: Triples)

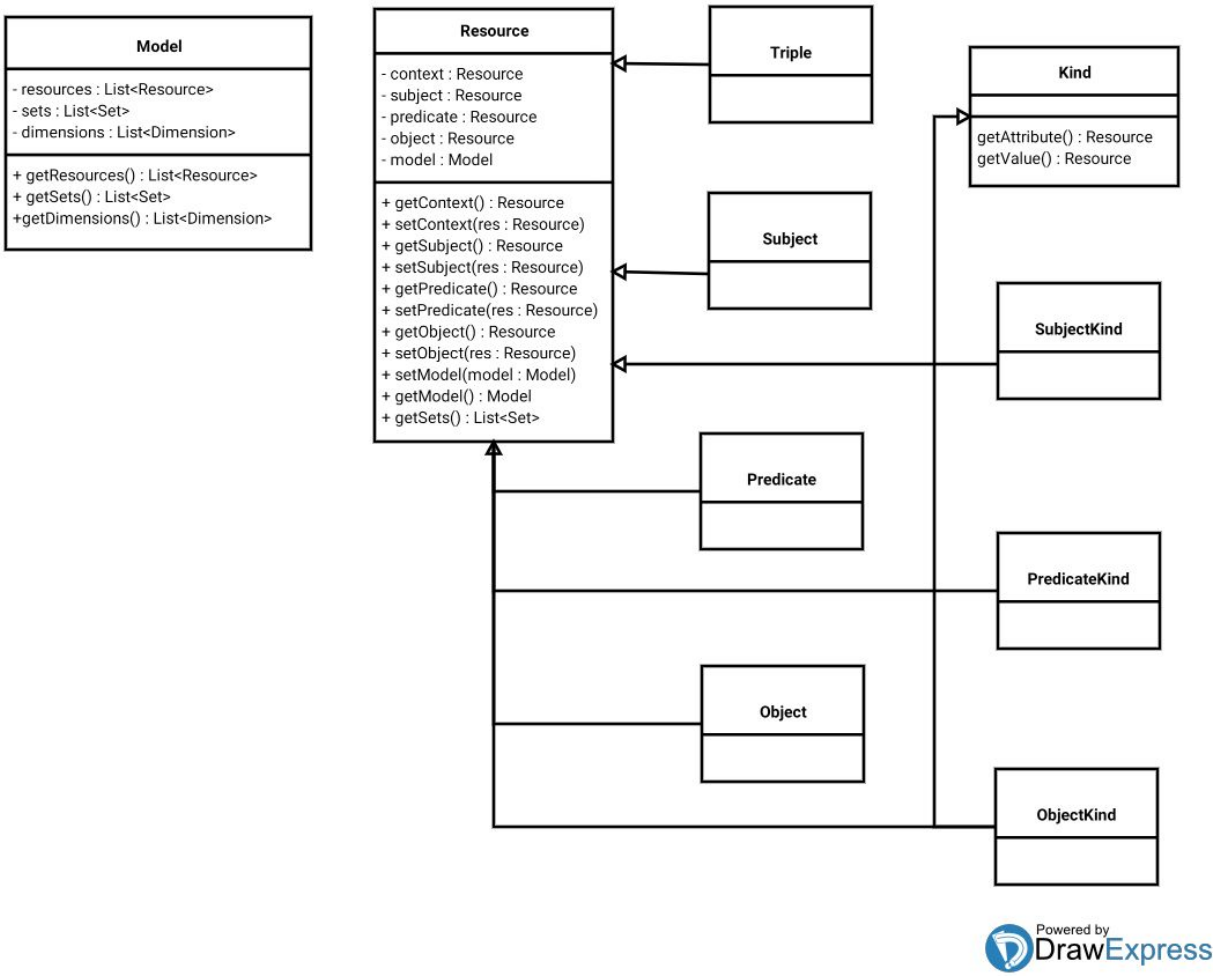
Monadic type ctor.:
ResourceMonad<T extends Resource>

Subject example:
SubjectMonad extends ResourceMonad<Subject>

Unit function:

if(subjectPredicate.holds(val)
return new SubjectMonad(val);

Bind function:
Specific to each type. Retrieve argument callback.
Returns type monad



d) Dimensional arrangement of Resources

For ease of development in what to ontology alignment and merge is concerned a ‘Dimensional arrangement’ of the underlying data is attached to each model thus providing enough metadata for equivalence resolution and for augmenting functional APIs behavior.

Object example: below is an example of such an arrangement for a given Object. It consists of a ‘dimension’ of the object types being considered, a ‘unit’ of measure in such dimension and the instances of their ‘values’

Dimension: Object Kind.
class: hiers (attributes) / ranges: meta (values)

Unit: Predicate Kind.
class: hiers (attributes) / instances: meta (values)

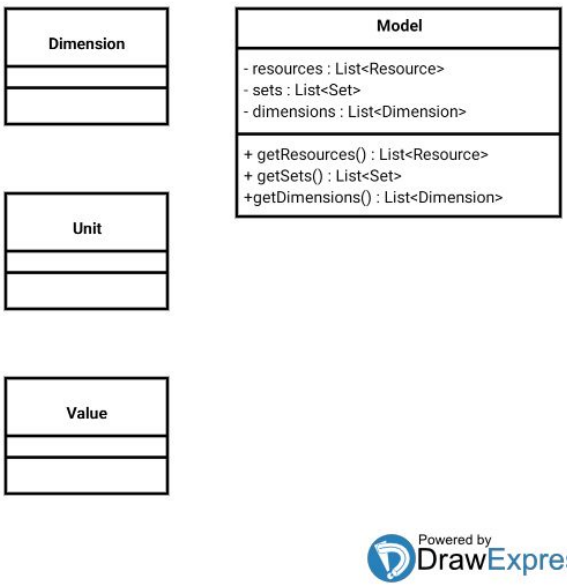
Value: Subject Kind
class: hiers (attrs) / domain: meta (values)

Model representation:
Map<Dimension, Map<Unit, Set<Value>>

Dimension, unit, value: hierarchies, primitives. Primitives and composition should be used for ontology translation / merge.

SCO Primitives, mappings, merge.
Knowledge level similarity. Primitive Signs, Concepts, Objects.

Function<Domain, Range> : instances.
Monadic bound functions. Augment functional API.



i) Equivalence inference functions

Model representation:
Aggregate Model Kinds (Subject example):

Map<Subject<Map<Predicate<Map<Object, ClassMeta>>>>

ClassMeta: Kind Resource. URI ID. Reifiable. Resource resolution. Predicates (attribute / value)

Align & Merge: mappings between equivalent Subjects, Predicates, Objects (merge dimensions, units & values):

Eqivalence sets:

Set<Set<Subject>> : ranges.
Set<Set<Predicate>> : instances
Set<Set<Object>> : domains

Equivalence functions SPOs:

Identify keys (attributes/values) that doesn't repeats for the same occurrence class.

Equivalent Predicates: equiv. domains / ranges, instances mappings.

Predicate equivalence predicates having equivalent SO in their statements.

e) Type inference (Subjects, Predicates and Objects)

Type inference is performed via Model Kinds and dimensional metadata.

For example a Subject of Kind ‘Person’ is a Subject with name, surname and birth date / age attributes of its Kind class (Predicates of SubjectKind) and their respective values which determines the class ‘metaclass’. For example an Octogenarian Person is a person which its age attribute has 80 as its value (SubjectKind’s Object).

Whenever a Person is hired for some Employment (PredicateKind) relationship, let’s say a ‘worksAt’ predicate statement is made with some Employeer (ObjectKind) value as its Object, there could be stated for listeners this as a flow event and try by some means to obtain the rest of the attributes that also should occur for the Employee (promoted Person) Kind: department,

salary, position, manager for example firing some rule event mechanism or having inference done with the relations of values occurring for that attributes (state flow event).

Predicates and Objects have their types (Kinds) inferred in a similar way, aggregating their attributes and values into classes and metaclasses (see the above diagrams). Maybe the only difference is that PredicateKind should use SubjectKind and ObjectKind as its attributes / values instead of plain Objects and Subjects.

f) Equivalence inference (Resources and Triples)

Equivalence of resources and triples is meant to be the basis for ontology alignment and merge of, for example, diverse vocabularies perhaps talking about the same subjects.

Type inference and dimensional arrangements are chances to provide the necessary means for performing such a task.

One equivalence resolution task may be to try find the ‘keys’ (like PKs on a database) on the attributes and values of a set of Kind(s). So, for example, in one source ontology one can find that the attribute ‘ns:ssn’ has values which doesn’t repeat one can be confident at certain point that this attribute values are keys for its Kind.

Then, for merging, there must be a mapping function between one ontology attributes and values into other ontology attributes and values. This is a very naive approach as values representing the same entities may be encoded differently in different sources (ie.: dashes separating digits) So, for now, the only attempt is to leave the API open for eventual and more sophisticated equivalence resolution mechanisms via the use of events and callbacks.

g) Relationship type and instances inference. Graph navigation

Due to, for example, a Subject having a class / metaclass of a given Kind there could be reasoning done about the relationships and the values of them the Subject of this given SubjectKind could have.

For example a Person being promoted into an Employee Kind due to it participating into an Employment relation (some statement asserts a ‘worksAt Predicate) turns to have defined for its Kind deptartment, manager, position and salary relations.

This relation’s values could be inferred from other Kinds instances such as same department Employees having the same manager, same position Employees having similar salaries and all this being backed by as much metadata corroborating this facts as possible. There is also the event listeners mechanism which can propagate this flow (hiring) event into rule firing events and state flow events for gathering / prompting for the missing data.

h) State transitions (Rules, Operations, Flows) inference

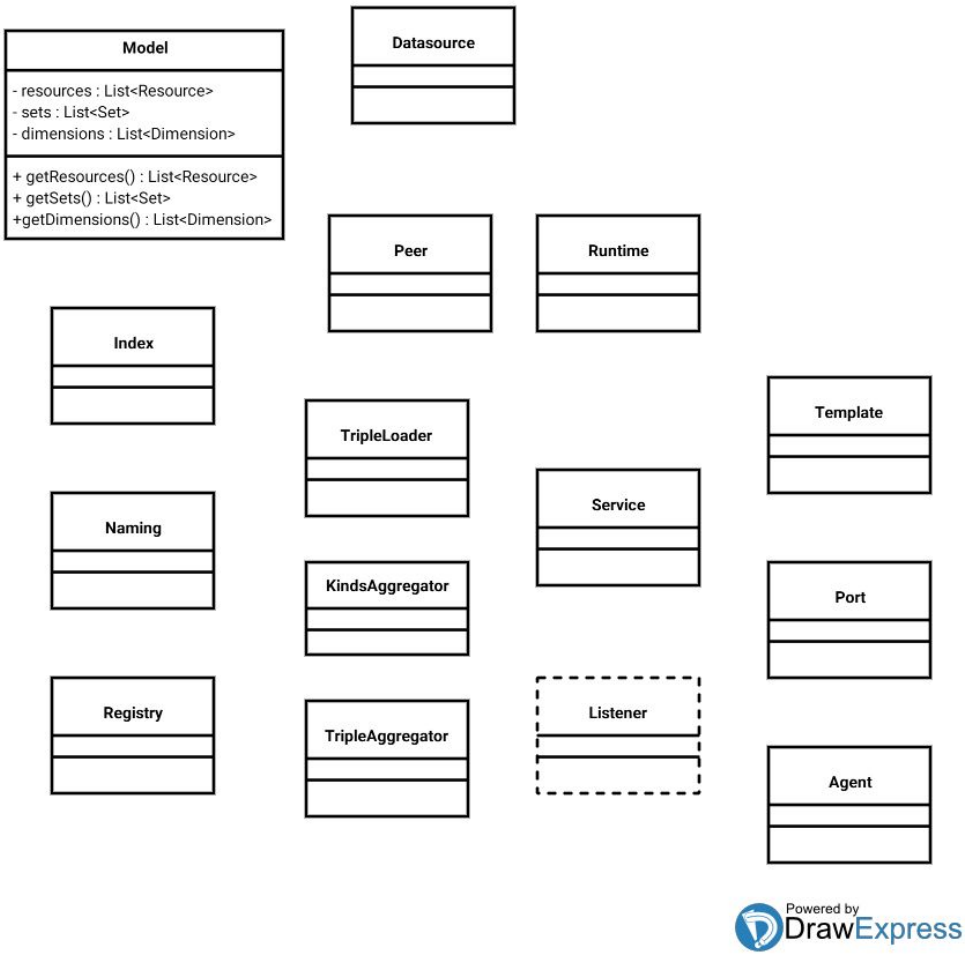
Contextual order relations lead to a tree structure in which when given some facts (statements) occurring in determinate position in a given sequence in the tree they fires flow, rule and state flow listener events. Contextual order (because it's stored in the context of the quads) is the means to arrange, first with a timestamp, order in triples and kinds related with those triples, first temporally and later logically (logical order of events)

Interactions with the datastore are done by means of a Service API. This Service API is where listeners registers themselves for the previous events resolution and also for ontology merging equivalence callbacks thus allowing distributed nodes to collaborate and learn merging ontologies.

The basic IO mechanism for interactions with the datastore at Service layer level are Templates (see below). Templates provide the means of message passing along CRUD and navigation invocations as well as the message payload format for the events and resolution listeners.

Functional Template invocation may make, for example, a PredicateKind look like a function (see below). Is important to note that this notation is intended to: 1) Declare things, ie.: instantiating something and 2) Pattern match things, ie.: as arguments for creating some new instance and that it maybe it is doing 1) in some contexts and 2) in anothers. So query and modification / data are embedded in a same sintax.

4) Architecture



a) Loaders (Data sources, sync)

Datasources are plugeable into the architecture. There must be a couple of interfaces implemented for each to build some kind of ‘driver’ which allows for syncing the changes made in the models into the original source while being able to retrieve their up-to-date contents as RDF.

Relational databases (JDBC compliant via D2RQ), EIS (CRMs, ERPs) and other common applications / protocols should have drivers for the system.

b) Runtime (Peer)

i) Input Jena Model

Raw loaders data. Used to populate initial models and to keep provenance metadata for syncing / updating models.

ii) Models (Sets, Dimensions, Tree)

Above set models and dimensional models comprise the core of the framework. There is also an order related (temporal and logical) model held for time related query / operations. It is represented as a tree ordering things from previous (root) to following (branches, leaves).

Dimensional models contains the basis for the callbacks / events mechanisms used to disambiguate equivalence relationships while merging.

The sets metamodels are implemented using functional programming techniques such as ‘Monads’ for uniforming the type space and deploy a set of useful ‘bound functions’ which are intended to build a richer API over Resources / Triples as jQuery (implemented with ‘Monads’) is for HTML DOM / JS.

iii) Index

Index: Any Triple to Name(s) Graph fragment. Graph API (Models). Registry bindings (topic / queue) dataflow.

iv) Naming

Naming: Parse / normalize names / URIs (Name entity: domain, NS, parts. Uniform Names abstraction layer. NLP. Dictionary. Definitions. Synsets (equivalence).

v) Registry

Registry: Hierarchical endpoint, dataflow placeholder (possible individuals) in dialogs over purpose protocol. Naming references resolution. Feed.

Listeners: dataflow.

vi) Output Jena Model

Model synchronized with sets metamodels to provide RDF(S), OWL and SPARQL endpoints.

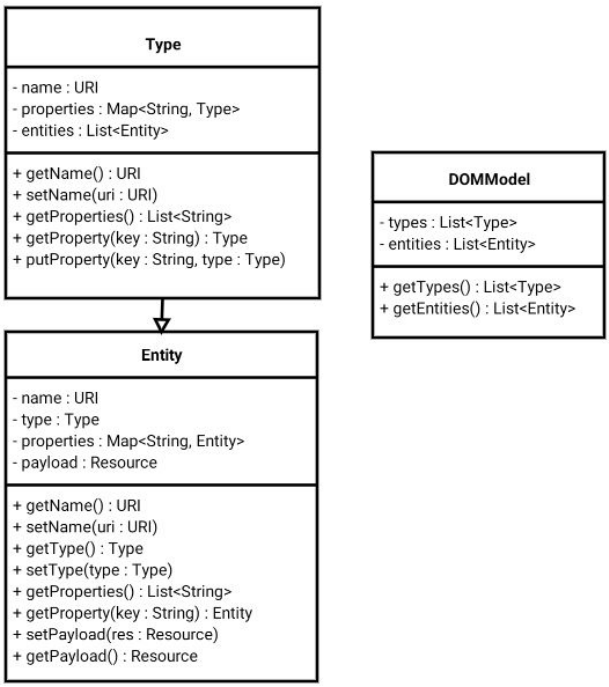
vii) Output DOM Model

DOM Model: Model for ORM like bindings, synchronized with sets metamodels. Example: JAXB binding to generated Java classes.

Model
types : Type[]
entities : Entity[]

Type : Entity
name : string (URI)
properties : Type[] (Map<string, Type>)
entities : Entity[]

Entity
name : string (URI)
type : Type
properties : Entity[] (Map<Type, Entity>)
payload : object



c) Services API

i) Templates

Templates are the mean for Service layer interaction IO (query, assertions). They are a recursive structure representing classes / instances of Kinds and their declaration or pattern matching.

Template definition:
Template : (Kind, Template lhs, Template rhs)

getTemplate (Template state) : Template temp

putTemplate (Template temp) : Template[] next

Contextual ordering: Templates returned from putTemplate are the ones temporal / logical contextual ordering allows for a given interaction.

Service layer operation fire flow, rule and state flow events as callback event triggers for equivalence resolvers registered.

ii) Listeners

State flow listeners: An instance is being ‘promoted’ to a new Kind because of a new assertion.

Rule listeners: Used for example in relationship resolutions, rule holds: employee dept A, employee manager Z.

Event listeners: State change event: Single, Married.

d) Functional query API

Due to the functional programming ‘Monadic’ design pattern and the Template(s) abstraction and notation it should be easy to compose the two concepts and, by the means of an uniform function space, to be able to build rich interactions (browsing, CRUD) solely from sending and receiving Template instances.

This ‘low level’ interaction protocol will be later leveraged by ‘Ports’ which will talk application specific and domain specific higher level protocols.

‘Nodes’ augmentation, analysis and discovery. DCI (Data, Context, Interactions) design pattern: behavior model. Model by example.

Agent client interfaces ‘activated’ w./ domain behavior. JAF (Javabeans Activation Framework) Components (AngularJS). Template state.

e) Ports (Representations)

i) RDF(S) / OWL

This port is not implemented by means of Service Templates but by exposing the ‘Output Jena Model’ by some endpoint.

RDF(S):
Kinds: Classes. Props: classId, metaClassId, attrs, values.
SPOs: Instances of Kinds. Props: ctx, resourceUri, classId, metaClassId.
Triple class. Triples: reified statement.

OWL:
Kinds: Classes. Restrictions.
RDFS Props.
SPOs: Individuals.
Triples: Individuals.

Add Inference layer.

ii) SPARQL

This port is not implemented by means of Service Templates but by exposing the ‘Output Jena Model’ by some endpoint.

Fuseki (embedded) server implementation based (Apache Jena).

iii) RESTFul HATEOAS (OData)

Leveraging Service Templates and interacting with the DOM output model. Representations build from metadata schema and instances. OData implementation based on Apache Olingo project.

iv) SOAP

Expose each Kind type as an operation. Nested template payloads (encoded in RDF) defines invocation semantics (query, CRUD)

v) Solid / LDP

Solid:

Implement LDP on top of RESTFul features. Containers should have WebID Profile describing what they contains and WebID(s) for instances of their contents.

WebIDs / WebID Profiles: Concise Bounded Description.

Container WebIDs: Data (SPOs)
Container Profile WebID: Schema (Kinds)

(Persons, Organizations, Groups, Devices, Requesting 'Agent', Server, Service: WebIDs Profiles?)

CBD Definition:
Given a particular node (the starting node) in a particular RDF graph (the source graph), a subgraph of that particular graph, taken to comprise a concise bounded description of the resource denoted by the starting node, can be identified as follows:
. Include in the subgraph all statements in the source graph where the subject of the statement is the starting node;
. Recursively, for all statements identified in the subgraph thus far having a blank node object, include in the subgraph all statements in the source graph where the subject of the statement is the blank node in question and which are not already included in the subgraph.
. Recursively, for all statements included in the subgraph thus far, for all reifications of each statement in the source graph, include the concise bounded description beginning from the rdf:Statement node of each reification.
This results in a subgraph where the object nodes are either URI references, literals, or blank nodes not serving as the subject of any statement in the graph.

Representations: Content negotiation / Activation.

Identity / Discovery (WebIDs, Profiles)

Authentication (WebID-TLS) / Login (HTML5 keygen cert. pub.)

Contacts Management

Messaging / Notifications

Feed aggregation / Subscription

Comments / Discussions

Friends / Followers / Following lists (topics, profiles). Users / Agents (event flows). WebIDs.

LDP: Linked Data Platform: RESTFul applications, shared storage space.
Resources, CRUD (Containers), Drive (Gestures, Augmentation)

Servers: LDP Implementations (ldnode). Decentralized RDF Data Model.

Basic Container -> Direct Containers (Multiple Facets)

Solid SPARQL: Each Resource is its own endpoint (default graph) INSERT, SELECT, DELETE

JSON-LD: (Metamodel loaders, services: OData). Contexts: Declarative schema.

WebSockets: Pub / Sub. Listeners changes.

Notifications (LDN Linked Data Notifications w3c.org): Inbox, discovery.

ActivityStreams (w3c.org). Messages / Streams.

POD: Personal Online Datastores:
. Server: Impl Default Containers (Kinds, Purposes). Workspaces, Preferences.
. Clients: Activation, Dashboards (workspaces, AngularJS DOM Activation).
. Applications: Configuration (Purpose instances). Runat Server / Clients.

Metamodel / Augmentations (Enhancements: Apache Sling, Apache Stanbol)

. Container WebID Profile: Kinds / Schema.
. Container WebID: Schema instances.

. Container schema / instance browseable, linked (HATEOAS).

Resource: Address, Type, Representation. Dereferencing.

Naming, Index storage, Registry: Persistence, Models.

f) Agents

i) Activation (over Representations)

Discovery of operations over resources regarding content type (JAF: Javabeans Activation Framework). Expose functionality to containers.

ii) Client API configurations.

JAF aware clients / agents adopt and render available operations over content resource. Declarative interface building.

5) Lab

a) Encoding and addressing

b) Octal order relation encoding

c) Lab: Higher order like predicates for SPOs, Kinds, Triples aggregates. Monadic constructors / wrappers. Logic, sets, filters, selection. Algebra (Monadic functors)

d) Lab: NodeJS + node-java or messaging protocol (JSON + Jersey / JMS). Browserify, local peer's nodes.

6) Dashboard example

Sample dashboard table aggregating vertically topics (ie. FriendFinder: by Location, RelationshipType, ContactName, 'BA', 'coworkers', 'Joe') and horizontally temporal / order relationships (time, events, duration and causal relations) grouped from less to more specific.

a) Dashboard Model

This model is intended for interaction with the models and Service APIs available from the core framework. It has its own Port implementation and 'protocol' that, although being RESTFul, doesn't complains with any specific standard, regardless other standard complaining Ports may be used while interacting with this one.

The Port and the protocol is presentation agnostic. Only renders Xs, Ys and Points. This concepts are defined below. Services API / Functional query frontend, REST Port Agents interfaces may be used / built for end user / services front ends of this component. Although the example given is a Tree Table like structure with a calendar like content rows pane there should be many other means to express (linked parts) of the knowledge exposed like as a dashboard with linked embedded 'portlets'.

Model:

X Axis: Topics / Actors (DCI Data).
Grouping (Model serialization):
(Kind / Instance) (Kind / Instance) (Kind / Instance)

Y Axis: Temporal / Events, (DCI Contexts). Steps.
Grouping (Model serialization):
(Kind / Instance) (Kind / Instance) (Kind / Instance)

Points: Performances (DCI Interactions).
Grouping (Model serialization):
X(Kind / Instance) Y(Kind / Instance) Point(Kind / Instance)

Temporal (logical) relations groups time lanes. Before event, after event, process steps by date, etc. allows for drilling, expanding, collapsing and facets filtering of shown data. Waterflow models: stages, stakeholders rendered as 1) Temporal events. 2) Topics (with context semantics).

Purpose layouts:
Scenarios, roles / actors, events, tasks, states regarding one specific context or situation (work, entertainment, holidays planning, etc.) may be statef as a set of filters / facets / groupings of axis to represent the best manner to work over some set of data.

Topics which aggregate data (vertically) are meant to be re-arranged having, for example, a sub-topic becoming the main node for a tree or having an instance view of all its topics. Drill, grouping, aggregation and (assisted) queries should narrow facets of the displayed knowledge.

Goals: Goals are the instantiation of a behavior kind (Purpose) by a topic (X Instance) into a place between orderings given temporal relations which, in turn, may be composed of the ordering given from the relation between other Goals (Y instance). Goals are represented by Points in the Model.

Point expanded view (of Point Model). Tiles or Gantt chart (w. context semantics) like embedded / rendered as / into dashboard. Chart would nest goals in contexts. Tiles / chart would render expanded ordering of goals / sub-goals due to their nesting and their ordering. The Goal / Point view should be expanded into the Dashboard using the same topic / ordering semantics of the summarized view thus leveraging X/Y drill, aggregation and filtering.

Parent (tile): Goal.
Child (tile): Subgoals (ordered).
Context injection: Parameterize sub goals (Prepare Apple cake, buy ingredients, buy apples).

Point Model Interactions:

CRUD Service Layer.
Subgoals ‘posted’, ‘notifications’.

Notifications: Topics, feeds: any item is ‘post-able’ to. Feeds (criteria sorted) of postings / events from any feed. Messaging: posts are replyable (post to a topic, user, subject. Threads. Ratings (relevance respect to a subject) comments (metadata, structured polls). Tags.

Point Model Activation (Javabeans JAF):
First glance of topic table lanes shows only title and metadata of outermost goals for a given temporal frame and a given topic. Activation allows to ‘interact’ with goals / events and retrieve allowed operations over items according its state. For example, in the goal in state ‘preparation’ of the cake example, one operation would be ‘helpWithRecipeTips’ while in ‘complete’ state an operation could be ‘askForSomeCake’.

Dashboard Sample:
Left side topics are intended to work as a tree table rendering / behavior.