**Date:** March 2023

### Common Datatypes

The APIs use a limited number of data structures across input and/or outputs, which can be divided into basic value types, reusable Data Structures and (ReST-enabled) Resources.

#### Primitive Datatypes

API4KP uses a limited number of primitive datatypes. In addition to the basic UML String, Integer and Boolean. additional datatypes are defined as follows:

* **DateTime –** The representation of an atomic point in time, with arbitrary granularity and optional time zone, as defined in the standard ISO 8601, and mapped to the W3C XML Schema ‘dateTime’ datatype.
* **URI / URL** **–** The unique identifier of a resource on the web, or address thereof, as specified in https://www.ietf.org/rfc/rfc3305 and predecessors

* **UUID** −A universal ID, conforming to the structure specified in <https://tools.ietf.org/html/rfc4122>

* **Bindings** −A Collection of Entries. An Entry is a key/value pair, where the key is a locally unique identifier, associated to a value of Any type. Bindings are primarily intended to capture variable assignments, such as may result from the execution of a Query. In this context, values can either be NULL (free/unassigned variable), or immutable (bound variable).
* **Any** − Marker datatype used to describe values that have an undefined, unstructured and/or unconstrained form. This datatype is mapped to the W3C XML Schema ‘anyType’.

#### Signs, Identifiers, Terms and Descriptors

An Identifier is a symbol used to *identify* one and one entity[[1]](#footnote-1) within some context. Depending on the context – *e.g*. universal vs local, web vs internal – it may convenient, or even necessary, for the identifier to have a specific form.   
.

In API4KP, URIs and UUIDs are considered the primary, general purpose form of identifiers. URI should be preferred when the identifier is intended to be persistent and/or dereferenceable. UUIDs are primarily used within the scope of API-mediated interactions, because they do not require a central authority to be minted and/or assigned. UUIDs and other forms of identifiers with internal structure such as OIDs and DIDs can also be used for more permanent entities, as long as they are mapped to their canonical URI form.

API4KP also defines a number of Structured Identifiers, as a way to uniquely denote Resources of interest, while providing a minimal descriptive context. The Resources of Interests include formal Knowledge Resources as managed by a Knowledge Platform, Semantic entities from a Business Domain of reference (“Concepts”), and versions and representations thereof.

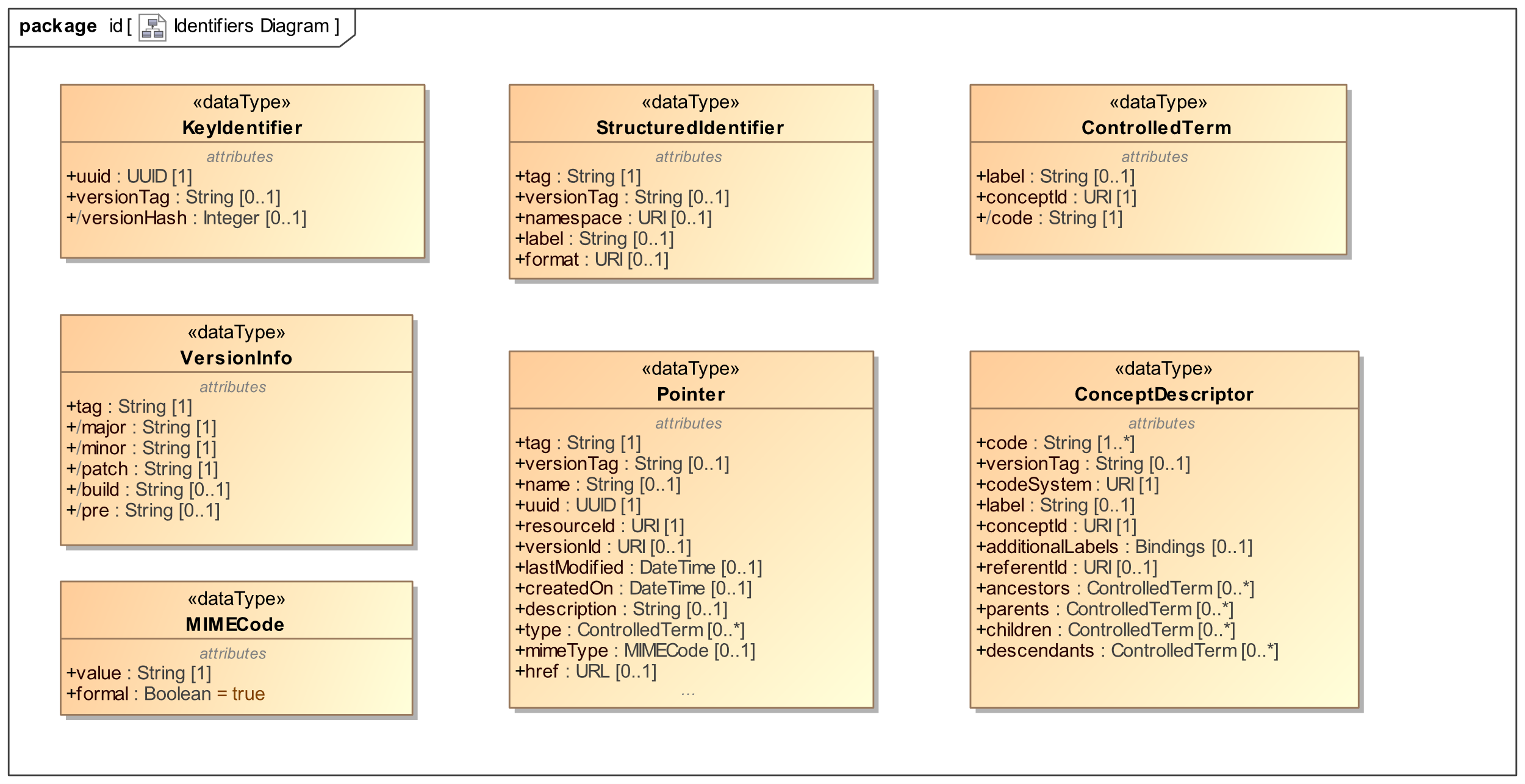


Figure 4. Identifier and Resource Identifier Descriptions

##### ConceptDescriptor

A Structured Designator that supports the bridging of Concepts, as atomic fragments of Semantic Knowledge Resources (e.g, Ontologies, Concept Schemes, Vocabularies) and Terms, syntactic representations used in other forms of Knowledge Resources and/or the APIs used to process them.

ConceptDescriptor is influenced by the W3C Ontolex model, and cover all three dimensions of the semiotic triangle. In particular, Concepts are universal individuals (in the SKOS sense) which are evoked by Terms, and are the intensional counterpart of their extensional Referent Entities – classes, relationships and/or known individuals. More specifically, Concepts can be organized in Concept Schemes, which usually provide a codification system and a ‘broader/narrower’ relationship, and/or (formally) defined in one or more Ontologies.  
As a datatype, ConceptDescriptor allows to carry references to other Concepts in the neighborhood of the given Concept, which can be used to support various reasoning tasks.  
  
Remark: since Concepts are considered fragments of a Semantic Knowledge Resource, they should not have a version that is independent from the version of the scoping Resource.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| code | The primary, local identifier of the Concept, within the scope of the defining Semantic Knowledge Resource. May or may not coincide with the local part of the conceptId. |
| conceptId | A universal identifier associated to the Concept which, by definition, is not scoped by the defining Semantic Knowledge Resource. Concept Ids are usually version agnostic. |
| codeSystem | The universal identifier of a Concept Scheme, where this Concept has been scoped. If present, the referenced Scheme should be the context which assigned the code and/or the conceptId. When possible, this URI should be version-specific. |
| versionTag | An optional version identifier associated to the specific Concept. If present, should be consistent with the version of the scoping Resource, which should be inferable from the respective URI. |
| referentId | The URI of the denoted Referent entity, possibly dereferenceable to, and resolvable within a formal Ontology |
| label | The (primary) term used to evoke this Concept, in the context of use of this ConceptDescriptor |
| additionalLabels | Additional terms associated to this Concept, as key/value pairs where the the key denotes the type/role of the label. The key should be derived from an annotation property (e.g. skos:altLabel), or a language code (e.g. ‘us-en’). |
| parents | References to other Concepts Pj=0..N in the same Scheme, such that this Concept ‘has broader’ Pj |
| children | References to other Concepts Cj=0..N in the same Scheme, such Cj ‘has broader’ this Concept |
| ancestors | The closure of the ‘has broader’ relationships in the context of the defining Scheme |
| descendants | The closure of the inverse ‘has broader’ relationship in the context of the defining Scheme |

##### ControlledTerm

A simple Structured Designator that can be used to denote an entity, and/or evoke a Concept. Combines a formal identifier with a (contextual) human readable label.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| conceptId | The universal identifier of either an evoked Concept, or a Referent entity.  Given that either choice has different formal properties, the choice should be consistent with the context of use of the ControlledTerm |
| code | A local identifier associated to the denoted/evoked entity. |
| label | The (primary) term used to evoke this Concept in its context of use |

##### KeyIdentifier

Structured Identifier that identifies a specific version a Knowledge Resource. A KeyIdentifier is designed for internal use by API4KP services, to index and retrieve efficiently resources, including KnowledgeBases and components thereof.

KeyIdentifer can be considered the minimal counterpart of a Pointer, from which it can be derived.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| uuid | The universal identifier associated with the resource, across its versions |
| versionTag | The component of the identifier associated to the specific version of the resource |
| /versionHash | A compact binary encoding of the versionTag, based on a hash function |

##### MIMECode

A designator of the characteristics of the manifestation of a Knowledge Resource. In API4KP, (generalized) MIME codes are terms that denote Media Types (<https://www.iana.org/assignments/media-types/media-types.xhtml>). More specifically, MIME “types” are associated to the categories of Knowledge Artifacts such as text or software, while the "subtypes” are correlated with the characteristics of the Knowledge Expression carried by a Knowledge Artifact.   
Moreover, A MIME Code is considered *formal* if its sub-type code can be parsed, and used to denote unambiguously the syntactic components of an Expression (language, profile, serialization, format, lexicon, alphabet and encoding). Informal MIME codes are considered pre-coordinated, and their interpretation is predetermined.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| value | The string-based representation of the MIME type code |
| formal | If true, denotes a post-coordinated (parsable) MIME code |

##### Pointer

Structured Designator that identifies a specific version a Knowledge Resource, while providing a minimal description of the Resource itself.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| tag | A version-agnostic Identifier associated to the Resource. Tags are not required to be universal identifiers. |
| versionTag | An identifier of the specific version of the denoted Resource, to be used in combination with the tag and/or uuid. |
| uuid | A version-agnostic, universal Identifier associated to the Resource, in the form of a UUID |
| name | A human readable, informative name that designates the Resource |
| resourceId | A version-agnostic URI that identifies the Resource – also known as “series” Identifier |
| versionId | A version-specific URI that identifies the Resource |
| description | A human readable, informative, contextual description of the Resource. |
| lastModified | The date/time of the creation of this version of the denoted Resource |
| createdOn | The date/time associated to the creation of the first, original version of the denoted Resource |
| type | One or more ControlledTerms that denote classifiers (e.g. OWL Classes) that apply to this Resource (e.g. such that the denoted Resource can be considered an instance of) |
| mimeType | A pre-coordinated basic descriptor of the representational characteristics of the denoted Resource.  When the Pointer denotes a Knowledge Asset, the mimeType can be omitted, or be used to denote a canonical representation of the Asset |
| href | A URL where the Resource can be accessed |

##### StructuredIdentifier

Structured Identifier that identifies an Entity as it is known to the Business *Clients* of a Knowledge Platform. Identifiers of this kind should be generally considered “metadata” by the API4KP services.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| tag | The textual form of the business identifier associated to the denoted Resource |
| versionTag | The identifier of the specific version of the denoted Resource |
| namespace | The URI of the namespace that scopes this identifier, as a proxy for the identification authority that assigns and/or allows to dereference the identifiers |
| label | A human readable name associated to the denoted Resource |
| format | A URI that denotes the grammatical rules that the tags should conform to, e.g. to distinguish OIDs from DIDs |

##### VersionInfo

VersionInfo are post-coordinated structures used to parse and process the version-specific part of the Identifier of a particular version of a Knowledge Resource, usually referenced as a ‘versionTag’.

Since API4KP recommends the use of semantic versioning, the structure is borrowed directly from the SemVer 2.0 specification, and its use should be consistent with that specification.   
Other strategies such as Calendar-based versioning should be aligned, for example using Year/Month/Day as Major/Minor/Patch components.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| tag | The pre-coordinated version tag |
| major | The major component of a structured version tag |
| minor | The minor component of a structured version tag |
| patch | The patch component of a structured version tag |
| build | The build component of a structured version tag |
| pre | The pre-release component of a structured version tag |

#### Correlation Between Identifiers

Implementations should, where possible, correlate the elements of structured designators such as Pointer and ConceptDescriptor. An entity should have a primary, universal URI, and optionally a version URI. These URIs should be decomposable into their namespace, tag and versionTag components. When appropriate, UUIDs should be derived functionally from the primary and/or version URIs, according to the UUIDs v5 methodology.  
  
Example:  
A Pointer with a decomposable URI and a versionTag:

**resourceId**: https://ckm.m.e/a/{tag}  
**versionTag**: {versionTag}  
 *// derived*  
**versionId**: https://ckm.m.e/a/{tag}/versions/{versionTag}  
**tag**: {tag}  
**uuid**: isUUID(tag) ? { tag } : UUID.v5from( {resourceId} )

Example:  
A Pointer to a Resource natively identified by a UUID (with an optional versionTag):

**uuid** : {uuid}   
**versionTag**: {versionTag}

// derived  
**resourceId**: urn:uuid:{uuid}  
**versionId**: urn:uuid:{uuid}{versionTag}  
**tag**: {uuid}

Example:  
A ConceptDescriptor referencing a coding system that differentiates between the system namespace and the entity namespace:

**resourceId**: {entityNs}/{tag}  
**codeSystem**: {systemNs}  
**tag**: {tag}  
**versionTag (system scope):** {systemVersionTag}

// derived  
  
**uuid**: isUUID(tag) ? { tag } : UUID.v5from( {resourceId} )

### (Canonical) Knowledge Surrogates

A Knowledge Surrogate is a ‘metadata’ Knowledge Artifact about other Knowledge Resource(s) that carries a relevant subset of syntactic, structural and semantic information that is relevant to some API4KP operation. There are numerous approaches to ‘Metadata’ models in and for Knowledge Management. The APIs formalize the notion of metadata record as ‘Knowledge Surrogate’, which is, and thus is processed as, a kind of Knowledge Artifact. The ‘canonical’ API4KP Surrogate model is a Schema (thus a kind of Knowledge Representation Language) which is mappable to other metadata models, and transrepresentable to/from those models by means of API4KP operations. The canonical model ensures that, whatever metadata model is natively adopted by a particular organization, the metadata that supports different API4KP operations is isolated and can be exchanged/exposed in a predictable way.

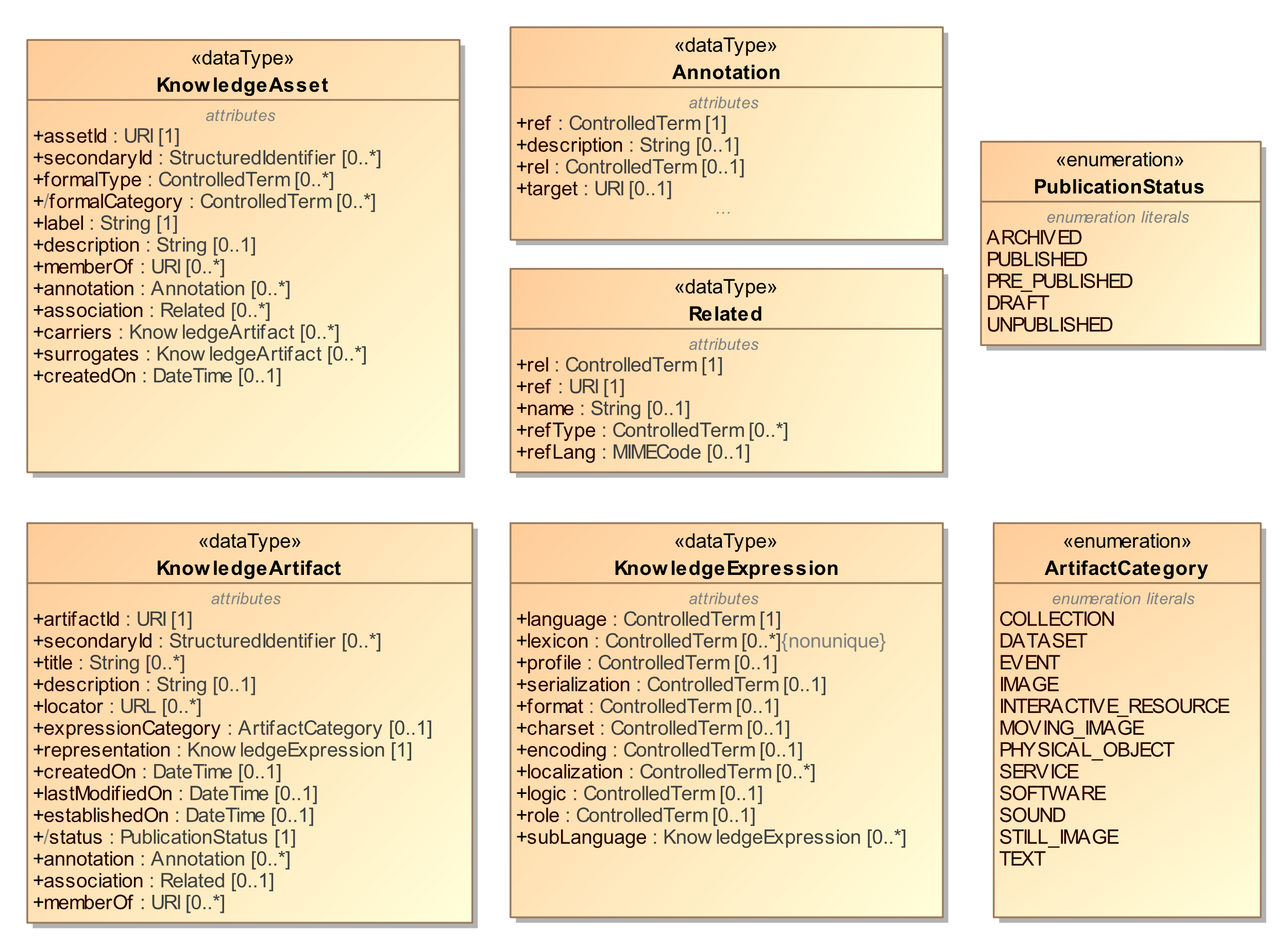


Figure 5. Knowledge Assets, Artifacts and Expressions

Figure 56 provides a view of knowledge assets and associated artifacts as used in API4KP. It is important to consider that, strictly speaking, the Surrogates are Representations of, but distinct from, the actual Knowledge Resources. Knowledge Assets are immaterial until expressed in some Language: a Surrogate is the only way to acquire some information without access to a representation of that knowledge itself. Knowledge Artifacts are documents, files and (in a sense beyond the scope of API4KP) even physical objects such as books and diagrams that the Knowledge Artifact Surrogate is a proxy for.

### Knowledge Asset (Surrogate)

A Surrogate that focuses on the Knowledge Asset, *i.e*., the knowledge content of a Knowledge Resource, regardless of the availability of any representation of that Knowledge.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| assetId | The version-specific URI that identifies the version of the Knowledge Asset described by this Surrogate. Should be decomposable into a KeyIdentifier |
| secondaryId | Zero or more business identifiers associated to the Knowledge Asset |
| formalType | Controlled Term that denotes a subclass of api4kp:KnowledgeAsset, according to a classification that is based on, or implies, the formal semantics of the work of Knowledge. Example: dol:Ontology, as in a logic-based axiomatic theory |
| formalCategory | Controlled Term that denotes a generalized classifier that classifies the Knowledge Asset |
| label | A human readable name, possibly contextual, associated to the Knowledge Asset |
| description | A human readable description of the Knowledge Asset. Descriptions are informative and for human consumptions: they should not be used for Knowledge Processing, and should not be confused with Knowledge Expressions that use some Natural Language representation |
| memberOf | A reference to one or more collections that this Knowledge Asset if member of.  Note that aggregates of Knowledge Assets should not be confused with Set-oriented Composite Knowledge Assets |
| annotation | Structured Annotations used to describe Asset/Concept relationships (see section 7.2.5.1) |
| association | Structured Links used to describe Asset/Asset relationships (see section 7.2.5.2) |
| carriers | The association between a Knowledge Asset (Surrogate) and its Knowledge Artifact (Surrogates), to reflect the association between the described entities |
| surrogates | The association between a Knowledge Asset (Surrogate) and other Surrogates of the same Asset, possibly including this Surrogate (“self”) |
| createdOn | A dateTime that reflects the moment when the Knowledge Asset was first created, to a sufficient degree of precision. |

### Knowledge Artifact (Surrogate)

A Surrogate that focuses on a Knowledge Artifact, *i.e*., any one of the concrete manifestations of a Knowledge Asset, including Assets that are Descriptions of other Assets.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| artifactId | The version-specific URI that identifies the version of the Knowledge Artifact described by this Surrogate. Should be decomposable into a KeyIdentifier |
| secondaryId | Zero or more business identifiers associated to the Knowledge Artifact |
| title | Human readable, often official, designations associated to the Knowledge Artifact |
| description | A human readable description of the Knowledge Artifact. Could be used as an informative summary of the Artifact content for human consumption, but should not be used for processing. |
| memberOf | A reference to one or more collections that the denoted Knowledge Artifact is member of. |
| locator | Any URL where (copies of) the Knowledge Artifact can be acquired |
| association | Structured Links used to describe Asset/Asset relationships (see section 7.2.5.2) |
| annotation | Structured Annotations used to describe Asset/Concept relationships (see section 7.2.5.1) |
| status | Publication Status of the described Knowledge Artifact |
| createdOn | A dateTime that reflects the moment when the Knowledge Artifact was first created, up to a sufficient degree of precision. |
| lastModifiedOn | A dateTime that reflects the moment when the given version of the Knowledge Artifact was created, up to a sufficient degree of precision. |
| establishedOn | A dateTime that reflects the moment when the given version of the Knowledge Artifact was published to a Knowledge Platform, up to a sufficient degree of precision. |
| expressionCategory | A classification of the material/digital form of this Knowledge Artifact, based on the DCMI (Artifact) Types taxonomy |
| representation | The description of the syntactic characteristics of the Knowledge Expression carried by the described Artifact |

It is important to remark that publication statuses are *derived.* Instances of an API4KP KnowledgeArtifact describe specific and immutable versions of an actual Knowledge Artifact Resource. As the Artifact evolves in terms of quality and maturity, different versions should be established, and the version tag should reflect the publication status.

In particular:

* Unpublished Artifacts do not “exist” as Resources, and thus do not have a stable version nor a status.
* Draft Artifacts are likely to have “SNAPSHOT” versions, not all of which may be tracked explicitly, which would be associated to a specific timestamp (“lastModifiedOn”).
* Pre-Published Artifacts versions (aka “Final Draft” or “Release Candidate”) would be tagged with a version tag that denotes the candidacy status.
* Published Artifact versions would have a stable version tag.

Also note that the attribute “establishedOn” allows to differentiate the time an Artifact becomes available to/through a Knowledge Platform, even if the publication process has not been managed through the Knowledge Platform itself.

### Knowledge Expression (Representation)

A Surrogates that focuses on the aspects of the formal language(s) used to construct a Knowledge Expression, as the representation of a Knowledge Asset and carried by a Knowledge Artifact.

As a whole, the Representation metadata correlates to the minimal required capabilities of a Knowledge Platform that is expected to parse the Artifact content, before it is processed.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| language | Term that denotes the abstract syntax of the (primary) representation language used in the Expression. Example: OWL2 |
| lexicon | Term that denotes the vocabularies, and implies the Concept Schemes / Ontologies, from which the Terms used in the Expression have been sources Example: SNOMED-CT, FIBO as medical and financial ontologies, respectively, providing terms (denoting concepts) that can be used to construct sentences (axioms) in OWL2 |
| profile | Term that denotes a well-known restriction of the primary language, usually trading expressivity for complexity. Example. OWL2-RL – a simpler but less computationally intensive subset of OWL2 |
| serialization | Term that denotes the concrete syntax used in conjunction with the primary language Example: RDF/XML, Turtle for OWL2 ontologies |
| format | Term that denotes the meta-format used to define the language’s serialization Example: The RDF/XML serialization of OWL2 is based on XML |
| charset | Term that denotes the Character Set used to construct the representation of the Terms, assumed to be compatible with the serialization Example: UTF-16, Windows-1252 |
| encoding | Term that denotes any additional re/encoding of the serialized expression Example: “Default” for the given charset; Base64 |
| localization | Term that denotes the natural language(s) used in the non-computational aspects of the Expression |
| logic | Term that denotes the formalism sufficient and necessary to capture the Expression, possibly associated to the computational complexity of the Expression. |
| subLanguage | Association between a primary and one or more secondary representations. A secondary representation denotes the language used to construct fragments, which are woven into a primary expression, usually to decorate, complement or supplement the primary expression. Example: fragments of MathML injected into a set of OWL2 axioms |
| role | Term that classifies a secondary representation, with respect to a scoping primary representation. Example: MathML as a (mathematical) functional expression language, extending OWL2 |

### Knowledge Resource Relationships

Knowledge Resources are related in different ways other than the ‘vertical’ Artifact – Asset stack.. Surrogates can carry those relationships, and linked Surrogates form a special kind of Knowledge Base that is at the core of the Semantic Knowledge Asset Repository APIs.

Relationships can be partitioned in two main categories:

* Knowledge Resource to (Domain) Concept relationships
* Knowledge Resource to Knowledge Resource relationships

### Resource to Concept Relationship

Knowledge Resource to Domain Concept relationships, also called informally “Annotations”, can be used to highlight focal Concepts that are part of that Resource’s Knowledge Asset for purposes such as searching and querying.  
Semantically, Annotations should be considered as reified RDF statements (“triples”) where the subject is the annotated Knowledge Resource. Annotations are expected not only to be consistent with the formal semantics of the target ontology, but also with the business domain semantics.

API4KP Annotations are generally aligned with the Web Annotation Data Model, which should be adopted for more complex use cases.

##### Annotation

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| rel | A ControlledTerm that denotes the ‘property’ that relates the subject Knowledge Resource and the object Concept. If omitted, the Concept should be considered a “semantic tag” |
| ref | A ControlledTerm that evokes the Concept related to the Knowledge Resource. |
| description | An optional, human readable representation of the referenced concept. Enables the preview of the annotation without having to dereference the object Concept |
| target | An optional identifier of the Component or Fragment in the source Knowledge Resource that this annotation more specifically applies to. If not specified, should be assumed to coincide with the source Resource itself. |

.

### Resource to Resource Relationship

Related(Resource) is a Link-like data structure that follows the principles of HATEOAS, and is used to establish associations between Knowledge Resources that are managed using the API4KP. A Related instance should be considered a reified Statement that points to a target Resource by means of its Identifier.

**Related**

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| rel | A ControlledTerm that denotes the specific semantic association between the subject Resource (denoted by the identifier in KnowledgeAsset or KnowledgeArtifact) and the target (“ref”) Knowledge Resource. The denoted relationship should be, or be consistent of, a subproperty of api4kp:associatedTo |
| ref | The Identifier of the (version of the) target Knowledge Resource |
| name | A human readable designation of the target resource |
| refType | An optional ControlledTerm that classifies the target Resource |
| refLang | An optional code that describes the syntactic form of the target Artifact, when the target Resource is a Knowledge Artifact |

Semantically, associations between Resources can be partitioned in 5 sub-categories, and further refined using hierarchies of properties defined in the API4KP-rel ontology.

* **Version (Series):**  
  Relationship between two individual Knowledge Resources that are versions of the same Mutable entity, and are partially ordered in a Series. As such, version-related Resources share essential characteristics such as subject or asset type/category.  
  Newer versions are usually meant to replace the older ones. Succession between Assets often implies derivation from the predecessor version; succession between Expressions often, but not necessarily, implies derivation.
* **Derivation**  
  Relationship between two Knowledge Resources A and B, which implies not only that B was used in the intellectual effort of creating A, but also that A exhibits some of the key concepts of B.  
  A derivative Resource may derive from multiple source Resources at the same time, but derivatives are generally independent: as a consequence, derivatives are usually not used together with their sources.  
  Derivation implies that the two Resources are distinct, so they do not need to share key characteristics.
* **Dependency**  
  The Dependency of a Resource A on a Resource B implies that the use of A is impacted by the ability to acquire and use B at the same time. Resources may or may not be distributed together, allowing for late resolution and binding.   
  The strength of the dependency, implied by the specific dependency relationship, determines if B is optional (A can still be used if B is not available), recommended (A can be used   
  without B, although less effectively) or mandatory (A cannot be used without B).
* **Component**Structural relationships are used in the definition of Composite Assets and Artifacts. Resources that are structurally related cannot be used without each other, and are retrieved and used together. In fact, removing or even rearranging the components results in a different Resource.   
  **Role**Component Links allow to define the role that the target Resource plays with respect to the subject Resource. Role is used primarily in the relationships between a Composite and its Components.
* **Variant**A is a variant of B if and only if A and B are alternative, distinct representations of the same Knowledge Asset.  
  In particular, Variance is based on characteristics of the Expression, not the carrier Artifact.  
  Variance MAY be associated to derivation, especially when a Resource B is obtained from a Resource A by means of a ‘horizontal’ transrepresentation operation.  
  Note: If asserted between Knowledge Assets, variance implies equivalence.

More details on the usage of relationships in combination with API4KP operations can be found in Annex A.

### Knowledge Carrier

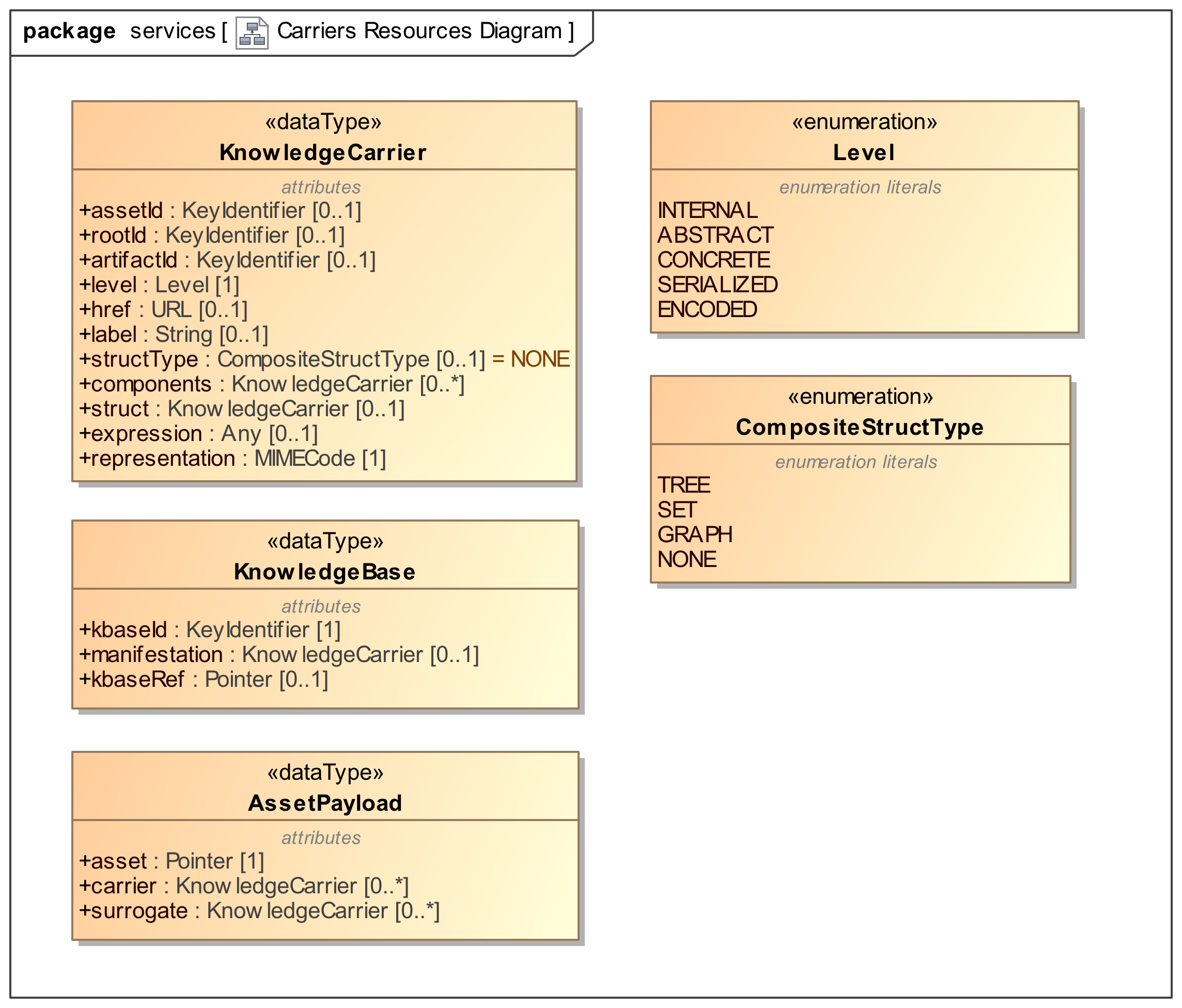


Figure 6. Structure of a Knowledge Carrier

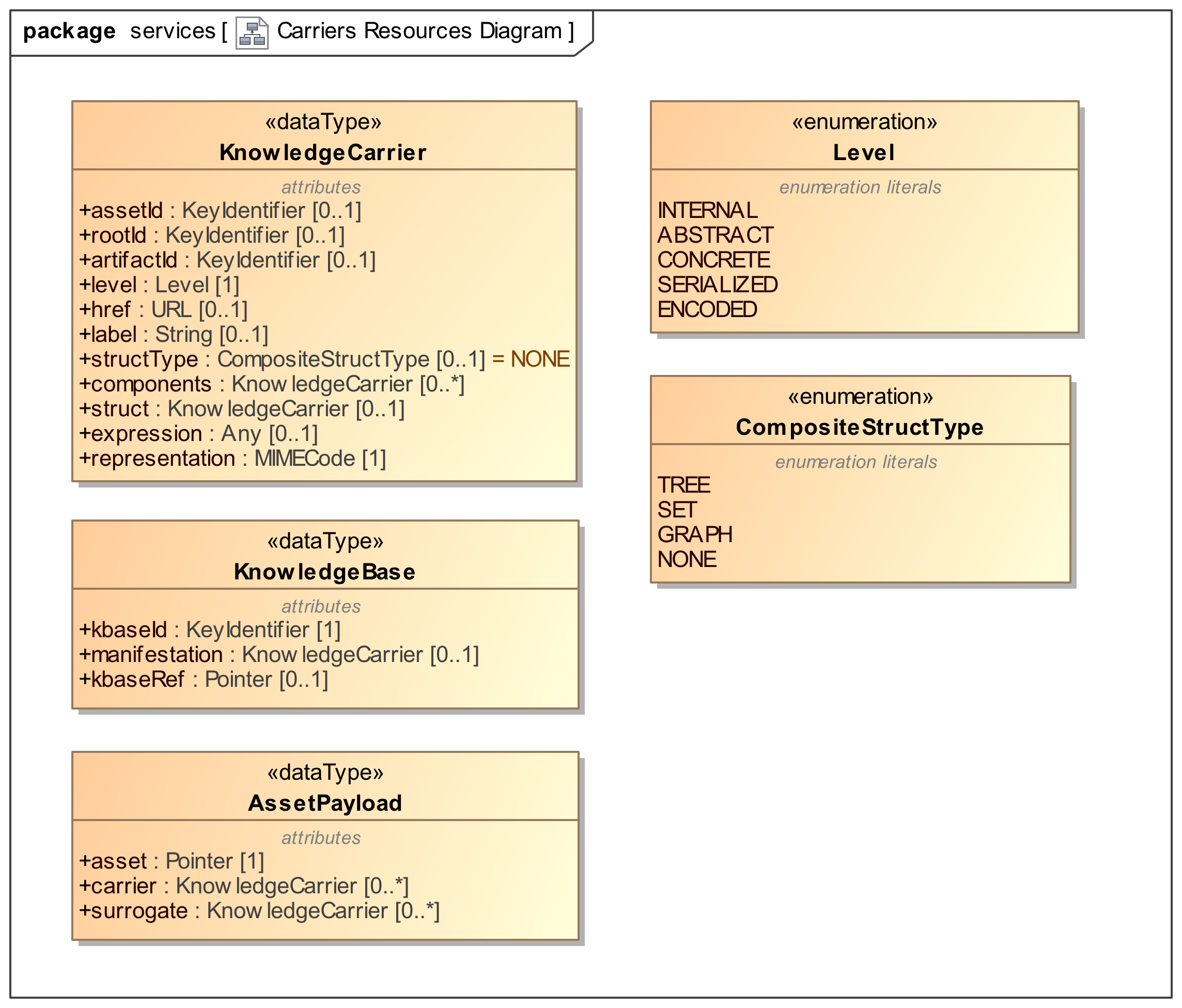
The KnowledgeCarrier structure, given in 

Figure 68, is the runtime counterpart of KnowledgeAsset. As a Surrogate, KnowledgeAsset provides information on Knowledge Resources ‘at rest’. KnowledgeCarrier, conversely, provides runtime metadata for Knowledge Resources ‘in motion’, as they are processed using the Operations exposed by the APIs. KnowledgeCarriers are initialized with information extracted from a KnowledgeAsset/Artifact surrogate, and updated by the same operations that manipulate the carried Knowledge Resource.

### Composite Knowledge Carrier

A Composite Knowledge Carrier is a Knowledge Carrier, and contains Knowledge Carriers, to support computation with Composite Knowledge Resources. While the components are stored in a flat list, a Composite KnowledgeCarrier delegates to a dedicated component, a “Struct”, the responsibility of tracking the actual internal structural relationships. Structs are Expressions, and can be implemented using, e.g., extensional RDF graphs or intensional sequence of DOL structuring operations. Composite Knowledge Carriers enable the distribution (map/reduce) of operations from the composite to the components. To this end, Composite KnowledgeCarrier categorizes the Struct in terms of its topology, and maintains a reference to the ID of the root component.

**(Composite) KnowledgeCarrier**

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| assetId | The identifier of the version of the carried Knowledge Asset |
| artifactid | The identifier of the version of the wrapped Knowledge Artifact |
| level | The level of abstraction of the wrapped Artifact (see 7.2.6.2) |
| label | A human readable designation of the wrapped Artifact |
| href | The URL where the Artifact can be retrieved, if not embedded as “expression” |
| expression | The wrapped Knowledge Artifact, at the given parsing Level |
| representation | A (formal) MIME Code that summarizes the representation aspects of the wrapped Artifact |
| rootId | Only applies to TREE-oriented Composite Resources, identifying the “root” component |
| structType | A summary descriptor of the topology of the Composite Artifact, or NONE for atomic Artifacts |
| struct | The representation of the Structure of a Composite Artifact, itself wrapped in a KnowledgeCarrier. Atomic Artifacts should not have a Struct |
| components | The Components of a Composite Artifact, each wrapped in a KnowledgeCarrier. In the case of Atomic Artifacts, any attempt to access the Components should return an empty collection, or a singleton that only comprises the atomic Artifact itself |

**KnowledgeBase**

A structured datatype that acts as a proxy for the manifestation of a named Knowledge Base. Note that Knowledge Bases are defined as Composite Knowledge Assets, which are manifested as (Composite) Knowledge Artifacts for computational purposes. A KnowledgeBase structure can either embed the Artifacts, or reference their location.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| kBaseId | The identifier of the version of a Composite Knowledge Asset, as a Knowledge Base |
| manifestation | An embedded (Composite) Knowledge Carrier, which realizes the Knowledge Base |
| kBaseRef | A Reference to the (Composite) Knowledge Base, when the Knowledge Base is not embedded. The Pointer should include a dereferenceable URI, or a href URL, to support the resolution of the KnowledgeBase content |

**AssetPayload**

A specialized KnowledgeCarrier designed to hold any number of Artifacts and Surrogates which co-reference the same Knowledge Asset, thus being mutual variants..

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| assetId | The identifier of a specific version of a Knowledge Asset |
| carrier | The variant of Artifact(s) that embody the Knowledge Asset version |
| surrogate | The variant of Surrogate(s) that embody a description (“metadata”) of the Knowledge Asset version |

### Parsing Levels

One important consideration is that, while some operations may conceptually apply to Knowledge Resources at the Knowledge Asset level, computation can only happen if some kind of Knowledge Expression is involved. Knowledge Carriers use the notion of *Parsing Level* to reflect the highest *Lifting* that the Expression has been subject to. The parsing level determines the nature of the ‘expression object’ carried by the Knowledge Carrier.

* **(Internal Semantic Graph)**   
  An internal, private representation that corresponds to an Agent’s internalization of a Knowledge Asset.   
  Not used by Knowledge Carriers, whose primary role is to facilitate the flow of information between Agents, including servers that implement the APIs
* **Abstract Knowledge Expression**An expression that conforms to the Abstract Syntax of the language used for representation.  
  The Knowledge Carrier wraps an Abstract Syntax Tree / Abstract Syntax Graph representation of the Expression
* **Concrete Knowledge Expression**An expression that conforms to the Concrete Syntax of the language used for representation  
  The Knowledge Carrier wraps a Parse Tree representation of the Expression
* **Serialized Knowledge Expression**A sequence of characters/symbols that has been generated according to the Concrete Syntax of the language used for representation.  
  The Knowledge Carrier wraps a String representation of the Expression, based on a Character Set.
* **Encoded Knowledge Expression**A sequence/array/stream of bytes which results from a mapping of the characters/symbols to a binary encoding.  
  The Knowledge Carrier wraps a binary-encoded representation of the Expression. The Encoding can be the DEFAULT Character Set / Binary encoding provided by a platform, but also re-encodings (e.g. Base64), compressions (e.g. ZIP) and/or cryptographic encodings,

### Monads

API4KP monads provide context around (Atomic) Knowledge Resource Objects, ensuring that operations, and chains thereof, can be applied consistently. Knowledge Representation languages and the tools that process them do not always  
support the context information natively because they are not generally designed for use in a hybrid environment. This gluing information is then provided as part of the API4KP infrastructure. Monad constructors ensure that necessary information such as identifiers and/or structure is available even when the languages used to express the knowledge do not support that natively; bindings then ensure that the context is maintained and propagated correctly as operations are performed on the Resources.

The role of Monads in API4KP can be summarized as follows:

* Monads are used as arguments by the public APIs.
* Monads wrap Resources expressed in a variety of notations, normalizing their use
  + Monads encapsulate the part of the API4KP specification that does not vary across logics, languages and serializations
* Monads bind Resources to API4KP atomic actions
  + Operations, exposed as APIs, are defined in terms of chained, atomic, functional actions

Conceptually, several Monads can be defined, each one highlighting a different aspect that concerns every API4KP operation, regardless of its specific nature and purpose.

#### Identifiable

Identifiable carries a Knowledge Resource’s ID in context, and ensures that IDs are propagated correctly as functions are applied to derive new Resources. Most operations transform a Resource and require assigning a new ID to the result. Some operations, however, preserve the identity of its operand(s)

For example, a *translation* action applied to Knowledge Resource preserves the ID of the Asset, but not the ID of the Expression, and thus impacts the ID of the Artifact the result will be engraved on.

Constructor:

Identifiable<R,I a Identifier>

= Dub R I | Mint R

Bind:

Identifiable<R> >>= f

= *preserveIdentity*( f ) ?

Dub f( R ) getId( R ) | Mint f( R )

Map:

( f : R → S ) → ( g: Identifiable<R> → Identifiable<S> )   
 = *sameAs*( R, f( R ) ) ?

Dub f( R ) getId( R ) | Mint f( R )

Functions:

getId : R → I

hasId : R, I → bool

newId : void → I

See also: Identity monad

#### Series

Series tags a Resource with version information, regardless of the actual implementation (timestamp-based, semantic versioning, incremental, etc.). Additionally, it ensures that version tags are updated properly as functions are applied to a Resource, depending on the nature of the function. An operation that does not alter its argument should not modify its version. Conversely, an operation that does not preserve Identity should also always generate a new version for the product. The new version may be set to an initial value (e.g. “0.0.1”), based on the time of execution, or derived (functionally) from a combination of the input’s version and the operation.

Constructor:

Versionable<R,V a VersionTag>

= Tag R V | Init R

Bind:

Versionable<R> >>= f

= *case:*

*preserveIdentity*( f ) -> Tag f( R ) getVersionTag( R )

*revise*( f ) -> Tag f( R ) next( getVersionTag( R ) )

*otherwise* -> Init f( R )

Functions:

getVersionTag : R → V

hasVersion : R, V → bool

next : V → V

newTag : void → V

See also: Identity monad

#### Trace

Trace maintains the list of (versions of) a Resource, involved in a chain of computations, according to the Memento pattern. Trace is a specialization of List, which assumes that the elements are ordered.

When applied to Resources, Trace is used to retrieve particular versions of a Resource, as well as to apply Functions to either a specific version of a Resource, or to the entire chain.

See also: List monad

#### Carrier

Carrier wraps an Expression with context that contains information about the representation of the Expression itself.   
As a monad, it ensures that the metadata is updated consistently, according to the semantics of the action itself. The constituents of the Carrier are as follows:

Constructor:

Carrier<R>

= Tag R representationInfo

Bind:

Carrier<R> >>= f

= Carrier

f( R )   
 f’( f( R ), representationInfo ( R ) )

Functions:

representationInfo : R → SyntacticRepresentation

Every action f that applies to Resources must also be implemented in a way that returns (infers, asserts or retrieves) the representation metadata about the returned Resource. In general, this information is function of the specific f( R ) and, secondarily of the metadata about R

See also: Maybe monad

#### Structure

Structure allows composition of atomic Resources into complex ones, by means of parent/child (tree) and sibling (set) relationships. As described in the seminal paper[[2]](#footnote-2). Structure has two components: the structure itself, a composite tree/set organization of one of more (specific versions of specific, carrier-wrapped) Resources, and a ‘manifest’ structure descriptor, which is itself a Resource

Constructor:

Structure<T,S,C>

= Empty | Construct (TreeSet T) (Manifest S)

TreeSet<R>

= Atomic R | TreeSet R

Bind:

Structure<T,S,C> >>= f

= case:

Empty → Empty

otherwise → Construct TreeSet f( T ) Manifest f’( S )

TreeSet<R> >>= f

= Atomic f( R ) | TreeSet f( R )

Functions:

flatten : ( T, S ) → C

See also: Either, Tree monad

#### Explain

Explain wraps a Resource and keeps an ‘explanation’ in context - an additional (Structured) Resource that carries additional information - e.g. provenance, or proofs - about the main Resource. The explanation is built incrementally, as operations are chained together, specializing the behavior of the classic Writer monad. Additionally, in case the Resource (or the Explanation) include variables/parameters, a Bindings structure is used to convey the associated values.

Constructor:

Explain<S,E>

= Explain S (With b)? (Explanation E)?

Bind:

Explain<S> >>= f =  
 X ← f( bind( S, b) )

Explain X   
 (With subst( b, f’(X) ))?

(Explanation add( explain(X), E) )

In particular, when Explains are chained through operations: i) the bindings are applied to the main structure S, ii) the associated action(s) are applied to the bound structure to derive the next Answer; iii) as needed, the bindings are propagated by substitution; iv) the Explanation for the latest action, if any, is incrementally added (structured) into the current Explanation.

See also: Writer monad

#### Integration

An API4KP Monad integrates *Identifiable, Series, Carrier and Explain,* and can be extended with *Series and Structure.*

In the API specification, Knowledge Carriers are used in combination with a Monadic wrapper, Answer, that combines the behavior of all the API4KP monads. In particular, (Composite) KnowledgeCarrier provides the structural component, while Answer defines the monadic operations – return/of, bind/map, join/flatmap.   
Answer is the same object has been introduced in Section 7.2.7, with the motivation of generalizing web-oriented operations to other integration patterns.

To support the development of the common API substrate, the following data structures harmonize some of the more popular, modern API development frameworks, while remaining compatible and aligned with the API4KP functional approach. Note that the structures are not specific to Knowledge APIs, and thus support but do not assume that the wrapped / references data is an actual Knowledge Resource.

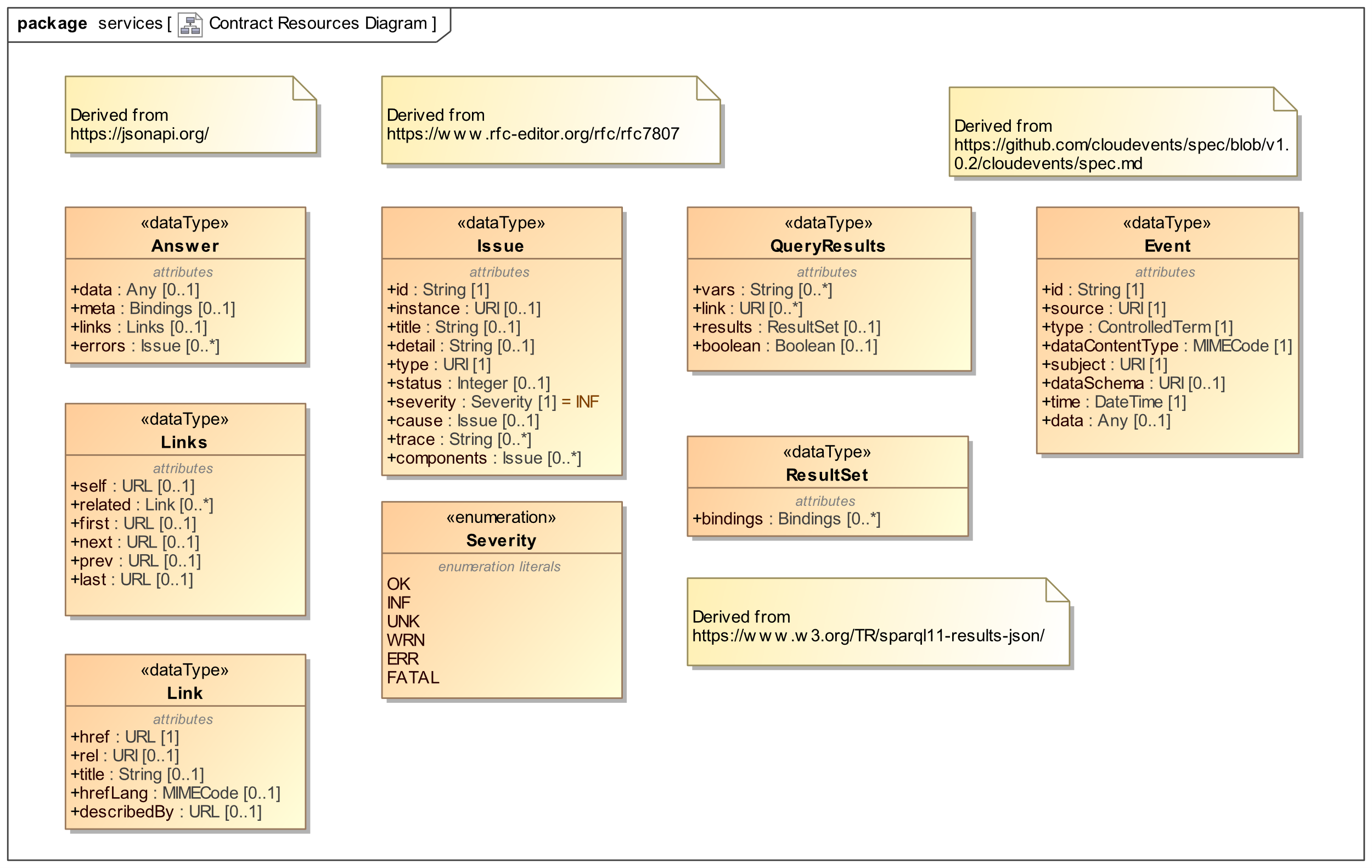


Figure 7. General Operation Patterns

**Answer**

The Answer data structure provides the substrate for the Answer monad.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| data | The actual payload – in API4KP, this usually consists of a KnowledgeCarrier |
| meta | Metadata about the execution of an operation. Mostly used for “technical” metadata, supports platform-specific execution contexts. In web-based implementations, this capability is implemented using HTTP headers |
| links | Server-driven links to drive further interaction |
| errors | Semantic metadata about the execution of an operation. Most often used to report errors and issues. |

**Links**

Links is a wrapper object that arranges any server-driven suggestion for further interaction with additional resources, available at the linked endpoints. Links can be paginated

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| self | A link to the original resource, which generated the Answer containing this supplemental Links |
| related | The collection of Link objects |
| first / next / prev / last | Pagination links, used when the ‘related’ list is considered too long |

**Link**

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| href | The URL where the linked target Resource is expected to be available |
| describedBy | A reference to the formal description of the behavior of the linked resource. Can range from a classifier Concept URI (e.g. type, role) to the location of a Knowledge Resource that specifies the nature and/or behavior of the resource. |
| title | A human readable designation of the target Resource |
| hrefLang | A MIME type that describes the syntactic representation of the information retrievable from the target link |
| rel | A term that denotes the relationship between the Resource just acquired and the linked one. |

**Issue**

Issue is a general-purpose data structure, based on <https://www.rfc-editor.org/rfc/rfc7807>, which can be used to describe a variety of operation outcomes, including errors of varying severity. When using Issue, servers should distinguish between the outcomes of an operation request as opposed to the outcomes of an operation execution*.*  
For example, a request may be successful, but return no information because there is no information to return, which could be an issue from a client perspective. Likewise, a request to perform a consistency check on an Ontology may succeed, yet discover that the ontology is inconsistent – a different scenario than one where either client or server was unable to provide/access the ontology to validate in the first place.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| id | An identifier of the Issue instance |
| instance | A reference to the primary individual entity that this Issue is about |
| title | A human readable summary of the Issue |
| detail | A full representation of the issue. May be human readable, or be handled as an embedded Knowledge Expression |
| type | The identifier of a classifier used to categorize this issue |
| status | The operation request outcome, as a HTTP status code, reflecting the status of the server from the client’s perspective. |
| severity | The operation execution outcome, as a severity level reflecting the server’s perspective on the client.  FATAL outcomes are expected block the client’s execution; ERROR outcomes require the client’s intervention; WARNING outcomes expect a client’s eventual intervention; OK and INF outcomes do not need nor expect the client’s intervention, respectively. UNK(nown) outcomes are undetermined |
| cause | An upstream Issue, which is considered to be the cause of this Issue |
| trace | An explanation of the issue |
| components | Sub-issues, used as components/fragments to describe this Issue in finer details |

**QueryResults**

A wrapper structure that provides context for a ResultSet, returned in response to a Query

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| vars | The list of variables for which bindings to result values are provided |
| link | Generic reference to additional information |
| results | The query response, as a matrix of variable bindings. Specifically, a ResultSet is a collection of Bindings, where each element in the collection describes a different entity, while each entity is described by a variable Bindings. |
| boolean | The query response, for queries that have a boolean response |

**Event**

Event is a data structure that can wrap Event payloads, in line with the CloudEvents specification

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| id | A unique identifier of this Event instance |
| source | The identifier of the context where the event was originated |
| type | A classifier of the Event |
| dataContentType | A MIME type that describes the format of the event payload. Must complement, or be consistent, with the dataSchema |
| subject | The identifier of the primary entity that the event is about |
| dataSchema | The identifier of the grammar/schema used to represent the event payload |
| time | The time at which the event occurred |
| data | The payload that describes the event occurrence |

### Operations - General Patterns

In line with the general API4KP principles, most operations are designed to follow a common pattern, as shown below in Figure 89.

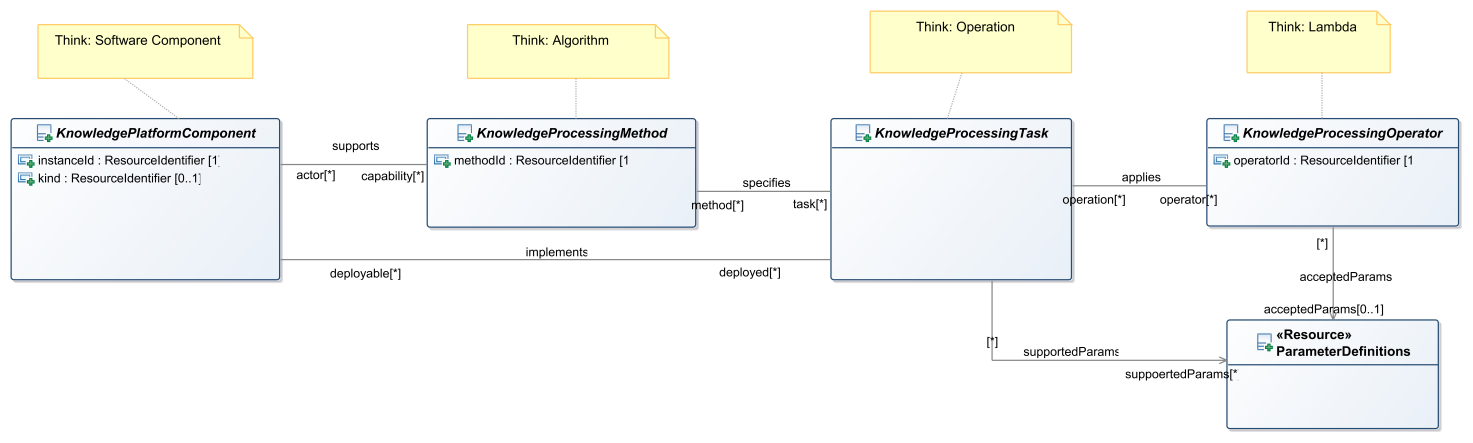
**

Figure 8. General Operation Patterns

API4KP Components are named Software components that implement at least one API4KP Knowledge Processing Task. The role of API4KP Component can be played by existing knowledge-oriented software, wrapped using API4KP interfaces, but also dedicated software that implements a variety of different algorithms.

An API4KP Component must provide behaviors that are compliant with the operation semantics defined in the API4KP ontology of Knowledge Processing Operations (api4kp-ops).

Service endpoints that conform to the API4KP signatures expose the operations to clients as functions. Operators are the modules that bind the signature to the underlying implementation, and are usually realized with strategies that range from sub-components to “lambdas”.

The APIs allow for some degree of insight into the Knowledge Platform Implementation.

#### Discovery

Servers as a whole, as well as individual Operators, can return self-describing resources with metadata and other descriptive information. Knowledge Platform components – including servers - can use “manifest” data structures to describe their own capabilities. The manifests include Operator descriptors, which can be used to advertise the specific operation types provided by the platform component.

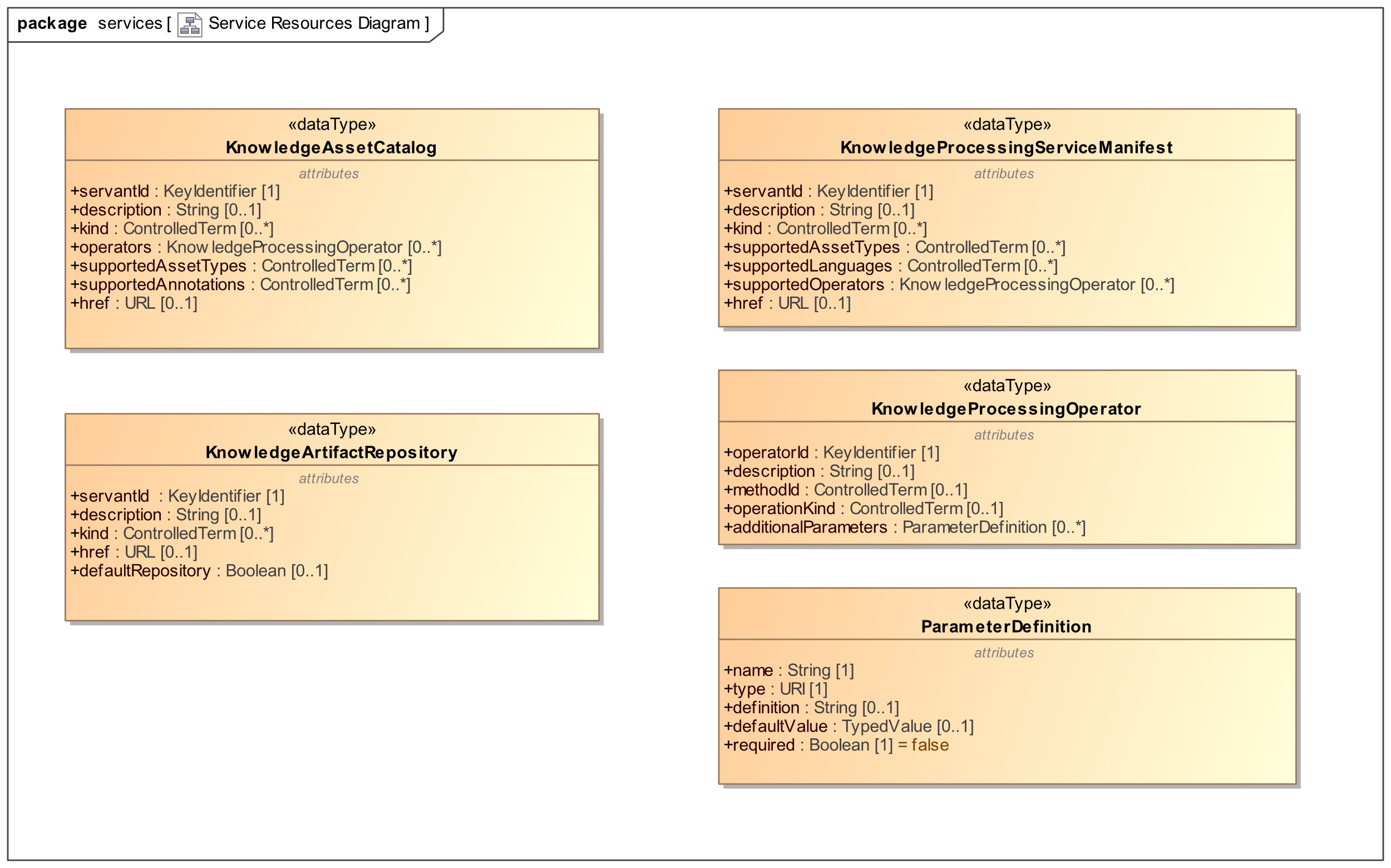


Figure 9. Service Capability Manifests

**KnowledgeAssetCatalog**

The manifest (summary descriptor) of a Semantic Knowledge Asset Repository Service

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| servantId | A unique identifier of the server, as a specific implementation of the API4KP specification |
| description | A human readable name and/or description of the server |
| kind | ControlledTerm that denotes a classifier that applies to this server, according to some classification scheme implemented in a given Knowledge Platform |
| href | The base URL where the server is deployed |
| operators | The additional KnowledgeProcessingOperator that are embedded in the server, augmenting its capabilities |
| supportedAssetTypes | The list of Types (Classifiers) of Knowledge Assets that the server has the capability to process |
| supportedAnnotations | The list of Property types that this server is able to support as Annotations (Resource / Concept association) |

**KnowledgeArtifactRepositoryManifest**

The manifest (summary descriptor) of a Knowledge Artifact Repository Service

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| servantId | A unique identifier of the server, as a specific implementation of the API4KP specification |
| description | A human readable name and/or description of the server |
| kind | ControlledTerm that denotes a classifier that applies to this server, according to some classification scheme implemented in a given Knowledge Platform |
| href | The base URL where the server is deployed |
| default | Flag that denotes a ‘default’ repository, where requests for specific Artifacts should be routed to, unless otherwise specified |

**KnowledgeProcessingServiceManifest**

The manifest (summary descriptor) of a Knowledge Transrepresentation, Construction and/or Reasoning Service.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| servantId | A unique identifier of the server, as a specific implementation of the API4KP specification |
| description | A human readable name and/or description of the server |
| kind | ControlledTerm that denotes a classifier that applies to this server, according to some classification scheme implemented in a given Knowledge Platform |
| href | The base URL where the server is deployed |
| supportedOperators | The KnowledgeProcessingOperators instantiated by the server |
| supportedAssetTypes | The list of Types (Classifiers) of Knowledge Assets that the server has the capability to process |
| supportedLanguages | The list of Representations of Knowledge Artifacts that she server has the capability to process |

**KnowledgeProcessingOperator**

The descriptor of any specific Knowledge Processing Operation implemented by a Knowledge Platform component. While API4KP endpoints differentiate the various Knowledge Processing Tasks structurally, the endpoints are not able to provide the semantic details of how an operation has been implemented by a specific server. The Operator descriptor, or extensions thereof, is designed to provide the additional information.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| operatorId | A unique identifier of the operator, specific to the implementation (version), but common across its deployments and instatiations |
| description | A human readable name and/or description of the operator |
| methodId | A Term that classifies the implementation technique(s) that the operator is based on (e.g. logic-based reasoning, NLP), up to denoting the specific algorithm, if well-known. |
| operationKind | A ControlledTerm that classifies the specific type of Knowledge processing operation, consistent with (any extension of) the API4KP Knowledge Operations ontology |
| additionalParameters | Operator-specific parameters that allow clients to further refine the behavior of the server |

#### Parameters

Several operations allow the client to provide component-specific Parameters.  
API4KP parameters are key/value pairs of simple Strings, which are (de)serialized according to the following grammar.

<Parameters> := <Parameter> (‘,’ <Parameter>)\*  
<Parameter> := <Key> ‘=’ <Value>

<Key> := <STR>

<Value> := <STR>

<STR> := \w\*

Component descriptors should include ParameterDefinitions – simple metadata objects that enumerate the supported parameters, and map each parameter to a definition. The definition COULD consist in a Knowledge Asset URI, to link to formal, machine readable and/or computable definitions.

**ParameterDefinition**

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| name | The unique name of the Parameter |
| type | The datatype of the Parameter value |
| definition | A human readable definition of the parameter’s purpose, admissible values and general usage |
| required | If true, the parameter will be considered mandatory. Clients are expected to provide a value, or a default value will be used, or the operation request will fail. |
| defaultValue | A representation of the value that will be assigned to the parameter, when no value is provided by the client |

Parameters can be used to drive the behavior of an operator, and could be used to refine, but must not extend nor alter the execution semantics of an operation. In particular, parameters MUST not be used to drive an API4KP endpoint to provide a function that should be exposed using a different API4KP endpoint.

#### Content Negotiation

APIs that return Knowledge Artifacts COULD support content negotiation to return variant formats of the resulting Artifacts, in order to meet client’s preferences.

Given that APIKP APIs use (Composite)KnowledgeCarrier wrappers, content negotiation, when supported, should distinguish between the format and/or encoding of the wrapper from the language, serialization, format, and/or binary encoding used in the Artifact itself. The former is usually controlled by the implementation frameworks: over web transactions, for example, Accept and Content-Type headers are used by user agents such as browsers and REST clients. The latter should be controlled by the API4KP components. Operations that support content negotiation expose an optional “extended Accept” parameter. Implementations should distinguish between their (in)ability to support content negotiation in general (*Unsupported*), from their inability to handle individual requests (*NotAcceptable*).

#### Pagination and Filtering

Operations that enumerate collections of resources COULD support pagination and filtering.

Pagination is supported using optional parameters *offset* and *limit*, following the usual semantics of indexing a Collection, returning resources in the range [offset .. offset + limit]. Default values of 0 and -1, respectively, allow to access the entire collection.

Sorting and filtering is resource-specific. Unless otherwise specified, sorting is performed according to the timestamps associated to the resources’ identifiers. The default filter is the null filter, which returns the entire collection.

Operations are performed in the order: filtering, sorting, pagination.

# 

1. Conversely, an entity can have multiple identifiers [↑](#footnote-ref-1)
2. <https://github.com/API4KBs/api4kbs/blob/master/publications/Monad_Trees.pdf> [↑](#footnote-ref-2)