# KTBS Documentation

***Release 1.0***

**P.A Champin, F.Conil**

**July 19, 2022**

# Contents

kTBS is a specific DBMS dedicated to traces.

This documentation first describes the *general concepts* of trace-based systems, and how they are implemented in kTBS. It then describes the *RESTful API* exposed by kTBS. The last chapter is a *developer's documentation* for using kTBS directly from Python code, or for modifying it.

Contents:

CHAPTER 1

# General concepts

## 1.1 Overview of the General Concepts

While in classical (relation) DBMSs a database contains tables, in kTBS a base contains traces (as well as other kinds of objects that we will describe later).

A trace aims at representing an activity as a set of **obsels** (observed element). Each obsel has, at least, a *type* and two *timestamps* (begin and end). It can also have an arbitrary number of **attributes** and **relations** with other obsels. Basically, this is all there is to know about obsels.

A trace is also linked to a **trace model**, which can be stored in the same base in the kTBS, or anywhere on the web. The trace model describes the obsel types that the trace can contain, their attributes and their relations. A trace model is to a trace roughly what a schema is to a table in an RDBMS —except that a trace model has an identity of its own, and can be shared by several traces.

Finally, traces can either be **stored** or **computed**. While stored traces contain data that is explicitly put there by external applications, obsels in computed traces are automatically generated based on a computation. That computation is specified by a **method**, either built-in or stored in the base together with traces and trace models.

## 1.2 Obsel

Obsels (short for 'Observed elements') are the atomic elements of traces. An obsel is described by the following elements:

- an obsel type,
- a begin timestamp,
- an end timestamp (that can equal the begin timestamp),
- an optional subject (the agent being traced).

It can also have multiple attributes, and be linked to other obsels of the same trace through binary relations.

The obsel type, the attributes and the relations are described by a *trace model*.

### 1.2.1 Obsel total ordering

When obsels of a trace need to be ordered, kTBS uses a total ordering considering

- their end timestamp, then
- their begin timestamp, then
- their identifier.

So obsels with different timestamps will be ordered according to their end timestamps; obsels with the same end timestamp but different begin timestamps will be ordered according to the latter; obsels with the exact same timestamps will be ordered according to their identifiers.

## 1.3 Trace Model

A trace model defines the following elements:

- a hierarchy of obsel types,
- attributes that obsels of each type can have,
- relations that can exist between obsels of each type.

In the future, the trace model will also define the time-unit to be used in the corresponding traces. For the moment, the only supported unit is the millisecond.

The obsel types are organised in a specialisation hierarchy: each obsel of a subtype also belongs to the supertype. As a consequence, attributes and relations are inherited from a supertype by its subtypes.

Note also that relations are also structured in a specialisation hierarchy.

In the future, it will be possible for a model to import another model; this will allow a model to refine another one, or modular models to be combined into complex ones. For the moment, this is possible by copy-pasting models into each others.

### 1.3.1 Model validation

A trace complies with its model if the following conditions are met:

- every obsel has an obsel type belonging to the model;
- every attribute belongs to an obsel whose type is (a subtype of) the domain of that attribute;
- every relation links two obsels whose type are (a subtype of) the domain and range (respectively) of the relation.

Note that there is no way for the moment to restrict the cardinality of attributes or relations: any attribute or relation may be omitted or repeated.

Note that traces are allowed not to comply with their model; kTBS will not prevent the creation of an obsel or the amendment of a trace causing it to be non-compliant.

## 1.4 Trace

A trace is a container of obsels, and is described by the following attributes:

- a trace model with which it should complies,
- an origin,

- optionally, one or more RDF graph containing contextual data about that trace.

### 1.4.1 Model compliance

A trace complies with its model if all its obsels have a type defined by the model, all their attributes are legal for their type, and all the obsel pairs in relation have types consistent with the domain and range of those relations.

Note that it is not required for a trace to comply with its model at any time. This is intended to let the user decide whether the trace or the model itself should be fixed.

### 1.4.2 Origin

Typically, the origin of a trace is a timestamp. However, it can also be an opaque string, meaning that the precise time when the trace was collected is not known.

This opaque string can nevertheless be reused across several traces (especially *computed traces*) deriving from another one), to indicate that timestamps are *comparable* across those traces.

### 1.4.3 Contextual data

The contextual data of a trace is any information that can not be captured by the trace model nor the obsels of that trace, but is useful to interpret the trace. For example, the contextual data may contain information about the subjects referred to by the obsels (their name, address, occupation. . . ).

Obviously, the contextual data of a trace should be valid for the whole duration of that trace. This limits the kind of information that it can contain: in the example above, if the subjects' occupation was expected to change in the time-span of the trace, they should rather be represented as obsels in the trace (with explicit timestamps).

### 1.4.4 Stored traces

The obsels contained in a stored trace are updated from outside the kTBS. There are two ways a stored trace can be updated:

- collection: this is the addition of a new obsel, with all its attributes and relations,
- amendment: this is an arbitrary modification of a trace (addition, modification or deletion of obsels, attributes or relations).

While it is expected that collection is the primary mode of updating stored traces, users may sometime have full control on their traces and require to amend them. This has some technical implication, as discussed in a *further section*.

### 1.4.5 Computed traces

The obsels contained in a computed trace are generated by the kTBS according to a computation *Method*. More precisely, a computed trace is described by:

- a method,
- optionnaly, a number of parameters,
- optionnaly, a number of source traces.

Which parameters and source traces are possible depend on the method used by the computed trace. In general, the model and origin of a computed trace are also determined by the method.

Note that the kTBS automatically updates the content of a computed trace each time it is accessed, so nothing is required to run the computation.

## 1.5 Method

A method is used by a *computed trace* to determine its model and origin, and to generate its obsels. The kTBS provides a number of *built-in methods*. It is also possible to create user-defined methods, that are stored in a base besides trace models and traces.

### 1.5.1 User-defined methods

A user defined method is described by:

- an inherited method (either built-in or user-defined),
- a number of parameters.

For simple methods such as filter, this is merely a way to define a reusable set of parameters. However, for more generic method such as Sparql or External, it provides a mean to encapsulate a complex transformation, possibly requiring its own parameters (via extensibility).

## 1.6 Monotonicity

Monotonicity is, loosely, the property of evolving always in the same "direction".

Traces have two ways of evolving: by collecting obsels, or by being amended. While amendment allows any kind of evolution of the content of the trace, collecting is more constrained.

By definition, collecting is restricted to adding new obsels, with their attributes and relations to previously created obsels. In a sense, those constrained can be considered as a kind of monotonicity, that we call **logical monotonicity**.

A stronger version of monotonicity is **strict monotonicity**: it is verified if every newly added obsel has its *end* timestamp greater or equal than the *end* timestamp of any obsel already present in the trace. In other words, a collecting is strictly monotonic if obsels are added in an order consistent with the internal chronology of the trace.

### 1.6.1 Why does it matter?

The more constrained the evolution of a trace, the more hypothesis transformations can make, hence the more optimised they can be.

For example, consider a transformation filtering obsels between two timestamp $s$ and $f$. If the source trace changes in a strictly monotonic way, once an obsel after $f$ is encountered, the transformation can safely ignore all subsequent obsels without even checking their timestamps.

On the other hand, if the source trace changes in a (non-strict) logically monotonic way, for every new obsel, the transformation has to check its timestamp, to decide whether to included it or not in the computed trace.

### 1.6.2 Monotonicity of computed traces

Monotonicity does not only apply to stored traces, but to computed traces as well. In that case, the monotonicity depends on two factors: the monotonicity of the source trace(s) (if any), and the applied method.

In the example above (temporal filtering), the method perfectly preserves the monotonicity of its source trace. The computed trace will evolve in a strict (resp. logical, non) monotonic way if the source trace evolves in a strict (resp. logical, non) monotonic way.

On the other hand, consider a transformation method that would keep only the last obsel of the source trace. This kind of transformation does not preserve monotonicity: even if the source trace is strictly monotonic, the transformed trace will evolve non monotonically. Indeed, each time an obsel is added to the source trace, any existing obsel in the transformed trace will be *removed*, and replaced by a copy of the latest obsel.

### 1.6.3 Current handling in kTBS

Every trace has a number of etags that change at various rates:

| | standard etag | strict mon. etag | logical mon. etag |
|---|---|---|---|
| obsels are deleted or modified | yes | yes | yes |
| an obsel is added anywhere | yes | yes | no |
| an obsel is added at the end of the trace | yes | no | no |

Internally, those etags are used by computed trace to determine how their sources have changed, and hence decide on which optimisation they can apply.

Externally, those etags are attached to different representations of the trace, to help clients efficiently cache those representations. For example, considering a trace of 100 obsels, the representation of "the first 10 obsels of that trace" will not change as long as the trace is modified in a strictly monotonic way. On the other hand, the representation of "the last 10 obsels of that trace" may be impacted by any kind of change on the trace.

Actually, kTBS uses a fourth etag, related to so-called **pseudo monotonicity**. This etag changes unless an obsel is added *near* the end of a trace (*i.e.* inside a time window at the end of the trace, called the *pseudo-monotonicity range*). The rationale is that in some situations, strict monotonicity can not be completely guaranteed (*e.g.* when obsels are collected by several agents with different latencies), but still obsels will not be added at arbitrary times. Hence pseudo monotonicity is a weaker property than strict monotonicity, but stronger than logical monotonicity.

## 1.7 Trace time management

As a trace aims at representing an activity, time management is an important concept in the kTBS.

There are several areas where you manipulate timestamps, and we will focus at first on time management for the *Stored trace*.

### 1.7.1 Use ISO 8601 format for datetimes

When you want to specify a real datetime, you MUST use the ISO-8601 format and you should **specify the timezone**.

If you don't specify anything the datetime string (1) will be considered as an **UTC datetime** as when the datetime string ends with "Z" character (2) because "Z" character is the zone designator for the **zero UTC offset**.

```
(1) "2016-01-06T08:15:00"
(2) "2016-01-06T08:15:00Z"
```

The timezone is specified as an UTC-time-offset, showing the difference in hours and minutes from Coordinated Universal Time (UTC), from the westernmost (12:00) to the easternmost (+14:00).

Suppose that it is "09:15 am" in French local time on January 6th 2016, the UTC time is then "08:15 am". To specify that your datetime string as a French datetime, you must use the UTC datetime and add the French UTC offset (+ 1 hour) at the end of the UTC datetime string (3).

```
(3) "2016-01-06T08:15:00+01:00"
    "2016-01-06T08:15:00+0100"
    "2016-01-06T08:15:00+01"
```

See ISO-8601 time zones representation

### 1.7.2 Stored trace origin

Each stored trace must have an origin which should be either:

- a datetime in ISO-8601 format, as specified above;
- the special string `now`, which will be replaced by the current datetime;
- any other string that can not be interpreted as a datetime, called an **opaque** origin.

If you do not configure the trace origin explicitly, an random opaque origin will be generated and associated to the stored trace.

### 1.7.3 Trace Model time-unit

The Trace time-unit is specified in the *Trace Model*.

The kTBS supports 3 time-units:

- `:millisecond` which is the **default unit**
- `:second`
- `:sequence`

### 1.7.4 Obsels timestamps

For the sake of simplicity, we will only consider the "begin timestamp".

When you create an Obsel in the kTBS, you may :

- omit the "begin timestamp"
- specify an integer "begin timestamp"
- specify a datetime "begin timestamp"

#### No timestamp specified

If no timestamp is specified, the kTBS will compute the "begin timestamp".

If the trace model unit is `:second` or `:millisecond`, the "begin timestamp" is the difference between the current datetime and the trace origin.

**Warning:** If the trace origin is opaque, an error will occur.

If the trace model unit is `:sequence`, an automatic integer numbering could be generated.

**Warning:** This is not yet implemented.

#### Integer begin timestamp

The integer value must be passed in the `:hasBegin` rdf parameter or in the `"begin":` parameter if passed in json format.

The kTBS keeps the integer as `:hasBegin` value.

#### Datetime begin timestamp

The datetime value must be passed in the `:hasBeginDT` rdf parameter or in the `"beginDT":` parameter if passed in json format.

The kTBS keeps the datetime as `:hasBeginDT` value and computes `:hasBegin` as the difference between the `:hasBeginDT` value and the trace origin using the trace model unit. Note that this only happens when the obsel is created. If after that the obsel is modified, and one of the timestamp (`begin` or `beginDT`), the other one will *not* be automatically updated.

## 1.8 Abstract KTBS API

Below is a language independant API that has been designed to document the functionalities of KTBS in a programmer-friendly way, and guide the implementers of client APIs in other languages.

**Warning:** TODO

- traceBegin and traceEnd on storedTraces

### 1.8.1 Resource

**get_id**()
Return the URI of this resource relative to its "containing" resource; basically, this is short 'id' that could have been used to create this resource in the corresponding 'create_X' method

**Return type** str

**get_uri**()
Return the absolute URI of this resource.

**Return type** uri

**force_state_refresh**()
Ensure this resource is up-to-date. While remote resources are expected to perform best-effort to keep in sync with the server, it may sometimes be required to strongly ensure they are up-to-date.

For local resources, this is has obviously no effect.

**get_readonly**()
Return true if this resource is not modifiable.

**Return type** bool

**remove**()
Remove this resource from the KTBS. If the resource can not be removed, an exception must be raised.

**get_label**()
Returns a user-friendly label

**Return type** str

**set_label**(*str*)
Set a user-friendly label.

**reset_label**()
Reset the user-friendly label to its default value.

### 1.8.2 Ktbs (Resource)

**list_builtin_methods**()
List the builtin methods supported by the kTBS.

**Return type** [Method]

**get_builtin_method**(*uri:str*)
Return the builtin method identified by the given URI if supported, or null.

**Return type** Method

**list_bases**()

**Return type** [Base]

**get_base**(*id:uri*)
Return the trace base identified by the given URI, or null.

**Return type** Base

**create_base**(*id:uri?*, *label:str?*)

**Return type** Base

### 1.8.3 Base (Base)

**get**(*id:uri*)
Return the element of this base identified by the given URI, or null.

**Return type** Trace|Model|Method|Base|DataGraph

**list_traces**()

**Return type** [Trace]

**list_models**()
List the models stored in that base.

**Return type** [Model]

**list_methods**()
List the methods stored in that base.

**Return type** [Method]

**list_bases**()
List the bases stored in that base.

**Return type** [Base]

**list_data_graphs**()
List the data graphs stored in that base.

**Return type** [DataGraph]

**create_stored_trace**(*id:uri?*, *model:Model*, *origin:str?*, *default_subject:str?*, *label:str?*)
Creates a stored trace in that base If origin is not specified, a fresh opaque string is generated

**Return type** StoredTrace

**create_computed_trace**(*id:uri?*, *method:Method*, *parameters:[str=>any]?*, *sources:[Trace]?*, *label:str?*)
Creates a computed trace in that base.

**Return type** ComputedTrace

**create_model**(*id:uri?*, *parents:[Model]?*, *label:str?*)

**Return type** Model

**create_method**(*id:uri*, *parent:Method*, *parameters:[str=>any]?*, *label:str?*)

**Return type** Method

**create_base**(*id:uri?*, *label:str?*)

**Return type** Base

**create_data_graph**(*id:uri?*, *label:str?*)

**Return type** DataGraph

### 1.8.4 Trace (Resource)

**get_base**()

**Return type** Base

**get_model**()

**Return type** Model

**get_origin**()
An opaque string representing the temporal origin of the trace: two traces with the same origin can be temporally compared.

**Return type** str

**get_trace_begin**()
The timestamp from which this trace was being collected, relative to the origin. This may be omitted (and then return null).

**Return type** int

**get_trace_begin_dt**()
The datetime from which this trace was being collected, relative to the origin. This may be omitted (and then return null).

**Return type** str

**get_trace_end**()
The timestamp until which this trace was being collected, relative to the origin. This may be omitted (and then return null).

**Return type** int

**get_trace_end_dt**()
The datetime until which this trace was being collected, relative to the origin. This may be omitted (and then return null).

**Return type** str

**list_source_traces**()

**Return type** [Trace]

**list_transformed_traces**()
Return the list of the traces of which this trace is a source.

**Return type** [Trace]

**list_contexts**()
Return the data graphs providing contextual information for this trace.

**Return type** [DataGraph]

**list_obsels**(*begin:int?*, *end:int?*, *reverse:bool?*)
Return a list of the obsel of this trace matching the parameters.

**Return type** [Obsel]

**get_obsel**(*id:uri*)
Return the obsel of this trace identified by the URI, or null.

**Return type** Obsel

### 1.8.5 StoredTrace (Trace)

**set_model**(*model:Model*)

**set_origin**(*origin:str*)

**set_trace_begin**(*begin:int*)

**set_trace_begin_dt**(*begin_dt:str*)

**set_trace_end**(*end:int*)

**set_trace_end_dt**(*end_dt:str*)

**get_default_subject**()
The default subject is associated to new obsels if they do not specify a subject at creation time.

**Return type** str

**set_default_subject**(*subject:str*)

**create_obsel**(*id:uri?*, *type:ObselType*, *begin:int*, *end:int?*, *subject:str?*, *attributes:[AttributeType=>any]?*, *relations:[(RelationType, Obsel)]?*, *inverse_relations:[(Obsel, RelationType)]?*, *source_obsels:[Obsel]?*, *label:str?*)

**Return type** Obsel

### 1.8.6 ComputedTrace(Trace)

**get_method**()

**Return type** Method

**set_method**(*method:Method*)

**list_parameters**(*include_inherited:bool?*)
List the names of all the parameters of this trace.

**Parameters** **include_inherited** – defaults to true and means that parameters inherited from the method should be included

**Return type** [str]

**get_parameter**(*key:str*)
Get the value of a parameter (own or inherited from the method).

**Return type** str

**set_parameter**(*key:str*, *value:any*)
Set the value of a parameter. An exception must be raised if the parameter is inherited.

**del_parameter**(*key:str*)
Unset a parameter. An exception must be raised if the parameter is inherited.

### 1.8.7 Model (Resource)

**get_base**()

**Return type** Base

**get_unit**()
TODO find stable reference to unit names

**Return type** str

**set_unit**(*unit:str*)

**get**(*id:uri*)
Return the element of this model identified by the URI, or null.

**Return type** ObselType | AttributeType | RelationType

**list_parents**(*include_indirect:bool?*)
List parent models. Note that some of these models may not belong to the same KTBS, and may be readonly —see get_readonly.

**Parameters** **include_indirect** – defaults to false and means that parent's parents should be returned as well.

**Return type** [Model]

**list_attribute_types**(*include_inherited:bool?*)

**Parameters** **include_inherited** – defaults to true and means that attributes types from inherited models should be included

**Return type** [AttributeType]

**list_relation_types**(*include_inherited:bool?*)

**Parameters** **include_inherited** – defaults to true and means that relation types from inherited models should be included

**Return type** [RelationType]

**list_obsel_types**(*include_inherited:bool?*)

**Parameters** **include_inherited** – defaults to true and means that obsel types from inherited models should be included

**Return type** [ObselType]

**add_parent**(*m:Model*)

**remove_parent**(*m:Model*)

**create_obsel_type**(*id:uri?*, *supertypes:[ObselType]?*, *label:str*)
NB: if id is not provided, label is used to mint a human-friendly URI

**Return type** ObselType

**create_attribute_type**(*id:uri?*, *obsel_types:[ObselType]?*, *data_types:[uri]?*, *value_is_list:bool?*, *label:str*)
NB: if data_type represent a "list datatype", value_is_list must not be true NB: if id is not provided, label is used to mint a human-friendly URI TODO specify a minimum list of datatypes that must be supported TODO define a URI for representing "list of X" for each supported datatype?

**Parameters**

- **data_type** – uri is an XML-Schema datatype URI.
- **value_is_list** – indicates whether the attributes accepts a single value (false, default) or a list of values (true).

**Return type** AttributeType

**create_relation_type**(*id:uri?*, *origins:[ObselType]?*, *destinations:[ObselType]?*, *supertypes:[RelationType]?*, *label:str*)
NB: if id is not provided, label is used to mint a human-friendly URI

**Return type** RelationType

### 1.8.8 Method (Resource)

**get_base**()

**Return type** Base

**get_parent**()
Return the parent method, or null. Note that returned method may not be stored on this KTBS, or can even be a built-in method.

**Return type** Method

**set_parent**(*method:Method*)

**list_parameters**(*include_inherited:bool?*)
List the names of all the parameters set by this method or its parent.

**Parameters** **include_inherited** – defaults to true and means that parameters from the parent method should be included

**Return type** [str]

**get_parameter**(*key:str*)
Get the value of a parameter (own or inherited from the parent method).

**Return type** str

**set_parameter**(*key:str*, *value:any*)
set the value of a parameter. An exception must be raised if the parameter is inherited.

**del_parameter**(*key:str*)
Unset a parameter. An exception must be raised if the parameter is inherited.

### 1.8.9 DataGraph (Resource)

This class has no additional method.

### 1.8.10 ObselType (Resource)

**get_model**()

**Return type** Model

**list_supertypes**(*include_indirect:bool?*)
List the supertypes of this obsel type.

**Parameters** **include_indirect** – defaults to false; if true, all supertypes are listed, including indirect supertypes and this obsel type itself

**Return type** [ObselType]

**add_supertype**(*ot:ObselType*)

**remove_supertype**(*ot:ObselType*)

**list_subtypes**(*include_indirect:bool?*)
List the subtypes of this obsel type from the same model.

**Parameters** **include_indirect** – defaults to false; if true, all subtypes from the same model are listed, including indirect supertypes and this obsel type itself

**Return type** [ObselType]

**list_attribute_types**(*include_inherited:bool?*)
List the attribute types of this obsel type (direct or inherited).

**Parameters** **include_inherited** – defaults to true and means that attributes types inherited from supertypes should be included

**Return type** [AttributeType]

**list_relation_types**(*include_inherited:bool?*)
List the outgoing relation types of this obsel type (direct or inherited).

**Parameters** **include_inherited** – defaults to true and means that relation types inherited from supertypes should be included

**Return type** [RelationType]

**list_inverse_relation_types**(*include_inherited:bool?*)
List the inverse relation types of this obsel type (direct or inherited).

**Parameters** **include_inherited** – defaults to true and means that inverse relation types inherited from supertypes should be included

**Return type** [RelationType]

**create_attribute_type**(*id:uri?*, *data_types:[uri]?*, *value_is_list:book?*, *label:str*)
Shortcut to get_model().create_attribute_type where this ObselType is the obsel type.

**Return type** AttributeType

**create_relation_type**(*id:uri?*, *destinations:[ObselType]?*, *supertypes:[RelationType]?*, *label:str*)
Shortcut to get_model().create_relation_type where this ObselType is the origin.

**Return type** RelationType

### 1.8.11 AttributeType (Resource)

**get_model**()

**Return type** Model

**list_obsel_types**()

**Return type** [ObselType]

**add_obsel_type**(*ot:ObselType*)

**remove_obsel_type**(*ot:ObselType*)

**list_data_types**()

**Return type** [uri]

**add_data_type**(*data_type:uri*, *is_list:bool?*)
NB: if data_type represent a "list datatype", value_is_list must not be true

**Parameters** **is_list** – indicates whether the attribute accepts a single value (false, default) or a list of values (true)

**remove_data_type**(*data_type:uri*)

### 1.8.12 RelationType (Resource)

**get_model**()

**Return type** Model

**list_supertypes**(*include_indirect:bool?*)
List the supertypes of this relation type.

**Parameters** **include_indirect** – defaults to false; if true, all supertypes are listed, including indirect supertypes and this relation type itself

**Return type** [RelationType]

**add_supertype**(*rt:RelationType*)

**remove_supertype**(*rt:RelationType*)

**list_subtypes**(*include_indirect:bool?*)
List the subtypes of this relation type from the same model.

**Parameters** **include_indirect** – defaults to false; if true, all subtypes from the same model are listed, including indirect supertypes and this relation type itself

**Return type** [RelationType]

**list_origins**()

**Return type** [ObselType]

**add_origin**(*ot:ObselType*)

**remove_origin**(*ot:ObselType*)

**list_destinations**()

**Return type** [ObselType]

**add_destination**(*ot:ObselType*)

**remove_destination**(*ot:ObselType*)

### 1.8.13 Obsel (Resource)

**get_trace**()

**Return type** Trace

**get_obsel_type**()

**Return type** ObselType

**get_begin**()

**Return type** int

**get_end**()

**Return type** int

**list_source_obsels**()

**Return type** [Obsel]

**list_attribute_types**()

**Return type** [AttributeType]

**list_relation_types**()

**Return type** [RelationType]

**list_related_obsels**(*rt:RelationType*)

**Return type** [Obsel]

**list_inverse_relation_types**()

**Return type** [RelationTtype]

**get_attribute_value**(*at:AttributeType*)
Return the value of the given attribute type for this obsel.

**Return type** any

Obsel modification (trace amendment)

**set_attribute_value**(*at:AttributeType*, *value:any*)

**del_attribute_value**(*at:AttributeType*)

**add_related_obsel**(*rt:RelationType*, *value:Obsel*)

**del_related_obsel**(*rt:RelationType*, *value:Obsel*)

### 1.8.14 General Rules

- Whenever parameter is named 'id:uri', it must be possible to provide a relative URI, which will be resolved against the URI of the target object.
- The order of the parameter is important. Whenever an optional parameter is to be omitted, it can be set to NULL or named parameters (language permitting) can be used for the following parameters.
- For all get_X methods accepting a parameter, the result should be null if no object matches the parameter.
- For all create_X methods, an exception must be raised if the given URI is invalid or already in use.
- All modification operations (set_*, remove) on model elements (ObselType, AttributeType, RelationType) actually modify the model from which they were accessed. If the model is readonly (see the get_readonly method), those methods must raise an exception.

### 1.8.15 Design Rationale

- As method-controlled attributes are not possible or easy to implement in some/ languages, this abstract API only defines *methods*, in order to provide the least common denominator.
- For the same reason, whenever mutiple values are to be returned, it prescribes the use of a list (or the closest match in the target language, e.g. Array in javascript).
- However, adaptations are also recommended, depending on the features of the target language. All those adaptations should be documented with the given API. Below is a list of recommended adaptations:
  - for languages supporting read-only attributes, it is recommended to provide a read-only attribute 'x' for every method get_x(); if get_x has optional parameters, 'x' should be equivalent to calling it with 0 parameters.

    It is also recommended to provide a read-only attribute 'xs' for every method list_xs(); if list_xs has optional parameters, 'xs' should be equivalent to calling it with 0 parameters.
  - for languages supporting method-controlled attributes, it is recommended to make attribute 'x' settable whenever there is a method set_x(val); if set_x has additional optional parameters, 'x' should be equivalent to calling it with only the first parameter.
  - for language supporting a notion of iterator (which may be more efficient than lists), it is recommended to provide a method iter_xs(...) for every method list_xs(...), acceptin the same parameters.

    NB: implementing list_xs(...) on top of iter_xs(...) should be trivial, and would probably be the way to do.
  - for language having a tradition of using CamelCase instead of underscore, all method may be renamed by replacing _[a-z] with the corresponding capital letter.

CHAPTER 2

# Tutorials

This chapter contains an number of tutorials to help you familiarize yourself with different aspects of kTBS.

## 2.1 Installing and running kTBS

These tutorials aim at helping you install kTBS and running it, either as a standalone service or behind an HTTP server such as Apache or Nginx.

It has been written using Debian-like systems : Debian strech and Ubuntu server, but should be applicable with only minor changes (if any) to other flavours of Linux, and a few adaptation on MacOS or MS Windows.

### 2.1.1 Installing a local kTBS

Make sure you have read and executed the *Common Prerequisites* instructions, i.e installed all dependencies, and a Python virtual environment.

#### Installing the development version (recommended)

Ensure that you have *activated your virtual environment*. Get the source code and install it in your virtual environment with *pip -e*.

```
(ktbs-env) $ git clone https://github.com/ktbs/ktbs.git
Cloning into 'ktbs'...
remote: Enumerating objects: 53, done.
remote: Counting objects: 100% (53/53), done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 6842 (delta 17), reused 37 (delta 15), pack-reused 6789
Receiving objects: 100% (6842/6842), 2.72 MiB | 1.11 MiB/s, done.
Resolving deltas: 100% (4400/4400), done.

(ktbs-env) $ pip install -e ktbs/
```

**Note:** The `-e` option makes pip install the current project in editable mode. It means that whenever you update the repository with `git pull`, of if you edit the code, the changes will be taken into account automatically.

If you intend to contribute, you might also want to install the developer's dependencies:

```
(ktbs-env) $ pip install -r ktbs/requirements.d/dev.txt
```

### Installing the stable version

**Instead** of installing kTBS from the source code, you can install it and its dependencies from PyPI. Ensure that you have *activated your virtual environment*, and simply type:

```
(ktbs-env) $ pip install ktbs
```

Note however that the stable version is not updated very often, and so might very quickly be outdated compared to the development version. Hence, this option is **not recommended**.

### Testing the installed kTBS

Once installed, just run the **ktbs** command, it launches an internal HTTP server on the 8001 port (by default).

```
(ktbs-env) $ ktbs
INFO        2019-09-15 14:28:18 CEST        ktbs.server     PID: 26566
INFO        2019-09-15 14:28:18 CEST        ktbs.server     listening on http://
↪localhost:8001/
```

You stop kTBS with `Ctrl-C`.

### Make the trace bases persistent

By default, kTBS stores the trace bases in memory, so they will not be retained after you stop kTBS. To make the trace bases persistent, you need to **configure a repository**.

This can be done with the `-r` option.

```
(ktbs-env) $ ktbs -r <dirname>
```

A directory named `<dirname>` will be used to store the trace bases; if it does not exist, it will be automatically created and initialized.

**Note:** You must *not* create the directory for the store; if the directory already exists, kTBS will assume that it is correctly initialized, and fail if it is not the case (*e.g.* if it is empty).

### Advanced configuration

There are a lot more configuration options that you can set on the command lines (type `ktbs --help` for a list). But a safer way to configure your kTBS instance is to store those options in a configuration file. An example is provided in the example/conf/ directory of the source code. Then, pass the configuration file as an argument to kTBS:

```
(ktbs-env) $ ktbs my.conf
INFO        2019-09-15 14:28:18 CEST        ktbs.server     PID: 26567
INFO        2019-09-15 14:28:18 CEST        ktbs.server     listening on http://
↪localhost:1234/
```

### 2.1.2 Installing kTBS behing an Apache HTTP server

Apache can be used as a proxy in front of a running instance of kTBS. This has a number of advantages:

- adds HTTPS support;
- allows kTBS to co-exist with other services under the same domain name and port number;
- allows to add access control.

#### kTBS configuration

kTBS must first be *installed* and run independently, listening on a local port. It must also be aware of the public URI under which it is published (in the example above: *https://your.domain.com/path/to/ktbs/*). This is achieved with the following configuration directives:

```
[server]
scheme = http
host-name = localhost
port = 8002

fixed-root-uri = https://your.domain.com/path/to/ktbs/
```

#### Apache configuration

A dedicated configuation file can be created, typically in `/etc/apache2/conf.d`, with the following directives:

```
# file: /etc/apache2/conf.d/ktbs.conf

<Location /path/to/ktbs/>
    ProxyPass http://localhost:8002/
    ProxyPassReverse http://localhost:8002/
</Location>
```

This configuration requires that Apache modules `proxy` and `proxy_http` are enabled; this can be ensured with:

```
$ a2enmod proxy proxy_http
```

Then Apache must be restarted to load the new configuration file:

```
$ apache2ctl graceful
```

#### Restricting access to kTBS with Apache

Traces can contain very sensitive information, so you will probably want to restrict access to your kTBS. To do this, you will need to add some **access control directives** to the Apache configuration file described above.

For the following examples, we use the htpasswd utility provided with Apache to create (`-c`) a **password file** that will be used as source for Apache authentication.

```
$ htpasswd -c ktbs-users jdoe
New password:
Re-type new password:
Adding password for user jdoe
```

Later on, you can add users to your password file with the same utility:

```
$ htpasswd /opt/ktbs-data/ktbs-users aduran
New password:
```

#### Basic global restriction

In order to restrict access to kTBS as a whole, the Apache configuration file above can be augmented to restrict access to kTBS, as illustrated below.

```
<Location /path/to/ktbs/>
    ProxyPass http://localhost:8002/
    ProxyPassReverse http://localhost:8002/

    AuthType Basic
    AuthName "kTBS"
    AuthBasicProvider file
    AuthUserFile /path/to/ktbs-users
    Require valid-user
</Location>
```

#### Finer-grain restriction

It might be tempting to define finer-grained ACL through multiple `Location` directives, to allow different users to access different part of your kTBS. Note however the following drawbacks:

- kTBS is not aware of these different ACL, and it may leak some information from one user to another one;
- when the structure of your trace bases changes in kTBS, you must reflect the changes in the Apache configuration.

Eventually, kTBS will provide its own authorization mechanisms, making this workaround moot.

#### What about mod_wsgi?

`mod_wsgi` is an Apache module dedicated to hosting Python WSGI applications entirely in Apache. In previous versions of kTBS, this was the recommended method for integrating with Apache.

The problem with `mod_wsgi` is that it must be compiled with the exact same version of Python as the one used by kTBS. Not only is it a problem if your distribution does not have the correct version of `mod_wsgi`, but it also prevents using different applications based on different versions of Python in the same Apache instance.

### 2.1.3 Installing kTBS behind an Nginx HTTP server

Nginx can be used as a proxy in front of a running instance of kTBS. This has a number of advantages:

- adds HTTPS support;
- allows kTBS to co-exist with other services under the same domain name and port number;

- allows to add access control (not documented yet).

#### kTBS configuration

kTBS must first be *installed* and run independently, listening on a local port. It must also be aware of the public URI under which it is published (in the example above: *https://your.domain.com/path/to/ktbs/*). This is achieved with the following configuration directives:

```
[server]
scheme = http
host-name = localhost
port = 8002

fixed-root-uri = https://your.domain.com/path/to/ktbs/
```

#### Nginx configuration

Here is an example Nginx configuration file:

```
server {
        server_name your.domain.com;
        listen 443;
        ssl on;
        # ... other parameters

        location /path/to/ktbs {
            proxy_pass http://localhost:8002/;
        }
}
```

### 2.1.4 Common Prerequisites

#### Dependencies

kTBS is a Python application, so you need Python installed; more precisely, you need **version 3.7** of Python. kTBS is not compatible anymore with Python 2.

As some dependencies need to be compiled, you will need

- the `gcc` compiler
- the Python developer files,
- the Berkeley DB developer files.

If you intend to install kTBS from the source (which is the recommended way), you will also need `git`.

On a Debian or Ubuntu, you should be able to get these dependencies by typing:

```
$ sudo apt-get install python3 gcc python3-dev libdb5.3-dev git
```

#### Create the Python vitual environment

A Python virtual environment creates an isolated version of Python, where you can install a Python application (such as kTBS) and all its dependencies without "polluting" your operating system. Conversely, the Python libraries already installed in your system do not interfere with the virtual environment.

Let us create a virtual environment for kTBS.

```
$ python3 -m venv ktbs-env
```

This will create a directory named `ktbs-env` (but you can choose another name, it makes no difference to kTBS).

The virtual environnement is then activated by **sourcing** the `activate` script. Once it is done, you can notice that

- the prompt has changed to remind you that the virtual environment is active, and
- the default Python interpreter is the one from the virtual environment.

```
$ source ktbs-env/bin/activate

(ktbs-env) $ which python
/current-dir/ktbs-env/bin/python
```

You "leave" the virtual environment by running the `deactivate` command.

```
(ktbs-env) $ deactivate

$ which python
/usr/bin/python
```

## 2.2 Using kTBS with REST and JSON

This tutorial aims at showing how to create *kTBS elements* directly through the REST API with JSON descriptions. If you are familiar with Turtle or RDF, you might prefer the *Turtle* version of that tutorial.

### 2.2.1 Tools

For interacting with the kTBS, we will use the simple HTTP client that is embeded within every HTML page generated by kTBS.

Don't forget, though, that this embeded client is only a convenience shortcut for easily interacting with kTBS. You can just as well use any HTTP client, either interactive (such as curl) or programmatic.

Note also that the JSON code displayed by the editor might differ from the one presented in those examples, for example in the order of the properties. However, they represent the same data.

### 2.2.2 Create and populate a Stored Trace

In this first part, we will create and populate a stored trace. But for this, we first need to create a Base that will host our traces.

### The kTBS Root

The kTBS root is where all bases live. It is automatically created when the kTBS is first launched. Its URI is that of the kTBS server, in our case: http://localhost:8001/ .

### Create a new base

To create a new base in our kTBS root, we have to perform an HTTP POST request to it.

Visit the kTBS root. Select the `POST` operation (this opens a text-area), and ensure that the selected content-type is `application/json`. Then copy the following JSON code in the text-area, and press `Send`.

```
{
    "@id": "base1/",
    "@type": "Base",
    "label": "My new base"
}
```

The URI of the newly created base should appear below the text-area (in our example: http://localhost:8001/base1/). By clicking on it, you will see a JSON description of your base.

You will notice that, besides the properties that you set when posting, the kTBS created two other properties, `@context` and `inRoot`. The latter links the base to the kTBS root it belongs to. We will explain the `@context` property later in this tutorial.

---

**Note:** A number of JSON properties expect URIs, as for example `@id` or `inBase`. In the example above, all URIs are relative to the URI of the resource *to which we post it*; for example:

- `base1/` is interpreted as `http://localhost:8001/base1/`;

this rule is true for all POST and PUT requests to the kTBS.

---

### Create a stored trace

Creating a stored trace inside a base is very similar to creating a base in the kTBS root. We first need to visit the base (you should already be there after the previous step).

Again, select the `POST` operation, ensure that the content-type is `application/json`, copy the following JSON code in the text-area, and click `Send`. Then click on the link appearing between the text-area.

```
{
    "@id": "t01/",
    "@type": "StoredTrace",
    "hasModel": "http://liris.cnrs.fr/silex/2011/simple-trace-model",
    "origin": "1970-01-01T00:00:00Z"
}
```

You will notice that, besides the properties that you set for the new trace, the kTBS created three other properties:

- `@context` (which we will explain later, be patient),
- `inBase` linking to the base containing this trace, and
- `hasObselList` pointing to http://localhost:8001/base1/t01/@obsels. This is where all the obsels that we are going to add to this trace will be created.

### Add obsels to trace

Adding an obsel to a trace should be no surprise to you at this point: it is simply done by POSTing a description of the obsel to the trace itself.

Simply visit the trace and POST the following content to it:

```
{
    "@id": "obs1",
    "@type": "m:SimpleObsel"
}
```

Note that `m:SimpleObsel` is a so-called compact URI, where the prefix `m:` stands for the URI of the model of the trace (followed by a hash `#`). So the type of the obsel is actually http://liris.cnrs.fr/silex/2011/simple-trace-model#SimpleObsel .

In the description of the new obsel, you will notice that this time the kTBS added a number of properties in addition to the ones you specified above. More precisely:

- The `begin` and `end` of the obsel have been automatically set based on the moment you posted the obsel; this is expressed in milliseconds since the origin of the trace.
- The `hasTrace` links the obsel to the trace containing it.
- The `@context` property.

It would have been possible to specify some of those properties explicitly, if we wanted to override the values automatically computed by the kbBS.

For example, let's go back to the trace and POST the following content to it:

```
{
    "@id": "obs0",
    "@type": "m:SimpleObsel",
    "begin": 1361462605000,
    "end":   1361462647000
}
```

We also note that, as with the base and the trace earlier, we had to mint a URI for our new obsels. As we are likely to create a large number of obsels, it sounds like a good idea to leave it to the kTBS to mint a fresh URI for each of them. For our third obsel, we will therefore use a blank node. We will also add attributes and relations to our new obsel to make it more interesting.

Let's go back to the trace and POST the following content to it:

```
{
    "@type": "m:SimpleObsel",
    "m:value": "a new obsel",
    "m:hasRelatedObsel": { "@id": "obs1" }
}
```

**Note:** Every element of the kTBS can be created with a blank node instead of an explicit URI. The URI minted by kTBS is returned by the `POST` operation.

If we follow the hasObselCollection link from our trace, to the obsel collection, we can see the three obsels we have created so far (your timestamps will obviously differ):

```
{

    "@context": [
        "http://liris.cnrs.fr/silex/2011/ktbs-jsonld-context",
        { "m": "http://liris.cnrs.fr/silex/2011/simple-trace-model#" }
    ],
    "@id": "./",
    "hasObselList": {"@id":"", "@type": "StoredTraceObsels" },
    "obsels": [
        {
            "@id": "obs0",
            "@type": "m:SimpleObsel",
            "begin": 1361462605000,
            "end": 1361462647000
        },
        {
            "@id": "obs1",
            "@type": "m:SimpleObsel",
            "begin": 1394791006055,
            "end": 1394791006055,
            "@reverse": {
                "m:hasRelatedObsel": {"hasTrace": "./", "@id": "o-8g"}
            }
        },
        {
            "@id": "o-8g",
            "@type": "m:SimpleObsel",
            "begin": 1394791489228,
            "end": 1394791489228,
            "m:hasRelatedObsel": {"hasTrace": "./", "@id": "obs1"},
            "m:value": "a new obsel"
        }
    ]
}
```

### 2.2.3 Creating computed traces

The kTBS has a number of *builtin methods* to create Computed Traces. As their name implies, computed trace differ from stored trace by the fact that their obsels are *computed* by the kTBS (in application of the corresponding method) rather than provided by external collectors.

#### Create a Computed Trace with a filter method

Let's go back to the base and create a new computed trace by POSTing the following:

```
{
    "@id": "filtered1/",
    "@type": "ComputedTrace",
    "hasMethod": "filter",
    "hasSource": [ "t01/" ],
    "parameter": [ "after=1361462641000" ]
}
```

This create a computed trace named `filtered1` based on a *temporal filter* which copies the obsels from `t01` obsels situated *after* timestamp 1361462641000. You may notice that we did not provide any model nor origin for the

computed trace; those are automatically computed.

If you go and check the obsel collection of this computed trace, you will find two obsels. More precisely, all obsels from `t01` have been copied, except for `obs0` which has been filtered out, as it is not entierly after timestamp 1361462641000.

### Create a Computed Trace with a SPARQL query

We will now define a more sophisticated computed trace, using the powerful query language SPARQL.

Let's go back to the base and create a new computed trace by POSTing the following:

```
{
    "@id": "joinRelated1/",
    "@type": "ComputedTrace",
    "hasMethod": "sparql",
    "hasSource": [ "t01/" ],
    "parameter": [ "sparql=    PREFIX : <http://liris.cnrs.fr/silex/2009/ktbs#>
↪\nPREFIX m:  <http://liris.cnrs.fr/silex/2011/simple-trace-model#>\n\nCONSTRUCT {\n␣
↪   [ a m:SimpleObsel ;\n      m:value ?value ;\n      :hasTrace <%(__destination__
↪)s> ;\n      :hasBegin ?begin ;\n      :hasEnd ?end ;\n      :hasSourceObsel ?o1, ?
↪o2 ;\n    ] .\n} WHERE {\n    ?o1 :hasBegin ?begin .\n    ?o2 :hasEnd ?end ;\n     ␣
↪  m:hasRelatedObsel ?o1 .\n    OPTIONAL { ?o2 m:value ?value }\n}\n" ]
}
```

This create a computed trace named `joinRelated1` using a SPARQL construct query to builds an obsel for each pair of related obsels in `t01`, inheriting its `begin` and `end` timestamps respectively from each of them.

As the SPARQL query is not very legible when encoded as a JSON string, it is provided below:

```
PREFIX : <http://liris.cnrs.fr/silex/2009/ktbs#>
PREFIX m:  <http://liris.cnrs.fr/silex/2011/simple-trace-model#>

CONSTRUCT {
    [ a m:SimpleObsel ;
      m:value ?value ;
      :hasTrace <%(__destination__)s> ;
      :hasBegin ?begin ;
      :hasEnd ?end ;
      :hasSourceObsel ?o1, ?o2 ;
    ] .
} WHERE {
    ?o1 :hasBegin ?begin .
    ?o2 :hasEnd ?end ;
        m:hasRelatedObsel ?o1 .
    OPTIONAL { ?o2 m:value ?value }
}
```

**Note:** It is frequent that SPARQL construct queries build obsels that comply with a model different from the source trace's. The target model can be specified with the special `model` parameter supported by the *sparql method*.

### Create a Computed Trace with a fusion method

We will now use the `fusion` method, used to aggregate in a computed trace the obsels from several source traces.

Let's go back to the base and create a new computed trace by POSTing the following:

```
{
    "@id": "fusioned1/",
    "@type": "ComputedTrace",
    "hasMethod": "fusion",
    "hasSource": [ "filtered1/", "joinRelated1/" ]
}
```

This creates a computed trace named `fusioned1` which is a merge of the `filtered1` and the `joinRelated1` traces.

### 2.2.4 So what about this `@context` thing?

Internally, kTBS uses RDF to represent its data. The JSON representations are therefore converted to/from RDF data. For this, kTBS uses a technology called JSON-LD. The `@context` property is JSON-LD specific, and provides the additional information required for the conversion to/from RDF.

It is worth noting that kTBS accepts both content types `application/json` (generic JSON) and `application/ld+json` (JSON-LD). When posting `application/json`, you may omit the `@context` property (as well as other properties, such as `inRoot`, `inBase` and `inTrace`), as we have done along this tutorial, but your JSON has to comply more closely to the structure expected by kTBS. When posting `application/json-ld`, you are free to structure your JSON as you wish as long as it translates into an RDF graph acceptable by kTBS; this usually implies that you provide the `@context` property explicitly.

## 2.3 Describing a model in JSON

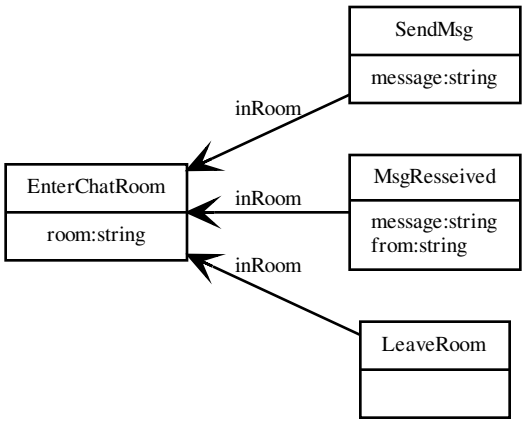
This tutorial explains how to describe a trace model for kTBS.

### 2.3.1 Preparation

First, you have to create a new base as described in the *REST tutorial*. We will assume that this base is named `http://localhost:8001/base01/`.

### 2.3.2 Running example

Here is a UML representation of the Trace Model we will create. This is a minimal trace model of a typical online chat activity.




### 2.3.3 Creating the model

On the page of your base, select the POST request, the `application/json` content type, and copy the following data in the text area:

```
{
    "@id": "model1",
    "@type": "TraceModel"
}
```

then press the `Send` button. The model is created and is now available at http://localhost:8001/base1/model1 .

If you visit that IRI, you will notice two things.

- The JSON object we have just POSTed is embeded in another object, with two attributes `@context` and `@graph`. The former has been described *in a previous tutorial*, we will come back to the latter further in this tutorial.
- In addition to its original properties, our object has a new property `"hasUnit": "millisecond"`. Every trace model must have a unit, specifying how time will be represented in the traces complying with this model (see *Trace time management*).

### 2.3.4 Modifying the model

#### Creating obsel types

We will first create the obsel types. Visit the model at http://localhost:8001/base1/model1 , select the PUT request, the `application/json` content type, and modify the code with the following:

```
{
    "@context": "http://liris.cnrs.fr/silex/2011/ktbs-jsonld-context",
    "@graph": [
        {
```

(continues on next page)

(continued from previous page)

```
            "@id": "http://localhost:8001/base1/model1",
            "@type": "TraceModel",
            "inBase": "./" ,
            "hasUnit": "millisecond"
        },
        {
            "@id": "#EnterChatRoom",
            "@type": "ObselType"
        },
        {
            "@id": "#SendMsg",
            "@type": "ObselType"
        },
        {
            "@id": "#MsgReceived",
            "@type": "ObselType"
        },
        {
            "@id": "#LeaveRoom",
            "@type": "ObselType"
        }
    ]
}
```

then press the `Send` button. The page should reload and show the new obsel types.

Since the obsel types are not explicitly linked to the model (apart from being defined in the same HTTP resource), we need to root the JSON representation in a single object, holding the set of resources together in its `@graph` attribute.

---

**Note:** Note that all relative IRIs in this example are interpreted against the IRI *of the model* (as it is the target IRI of the PUT request). For the sake of clarity, the `@id` of the obsel contains its full IRI, but the empty relative IRI `""` would work as well. All components of the models have their IRI starting with `#`, so `#EnterChatRoom` is a shorthand for `http://localhost:8001/base1/model1#EnterChatRoom`, for example.

Note that you could not have POSTed the JSON code above as is, as relative IRIs in a POST are interpreted against the IRI of the *base* (`http://localhost:8001/base01/` in this case).

It is still possible to create the obsel types together with the model at POST time, but then you need change the relative IRIs accordingly, `model1#EnterChatRoom` instead of `#EnterChatRoom`, etc.

---

### Adding attributes

We will now associate attributes to our newly created obsel types.

As obsel types, each attribute has a unique IRI, relative to that of the model: `#room`, `#message` and `#from`. It is related to the obsel type(s) in which it may appear by the `hasAttributeObselType` attribute.

The datatype(s) of an attribute is specified using `hasAttributeDatatype`. kTBS supports a subset of the primitive datatypes defined in XML-Schema, including the most usual datatypes such as `xsd:string`, `xsd:integer`, `xsd:boolean` and `xsd:float`.

```
{
    "@context": "http://liris.cnrs.fr/silex/2011/ktbs-jsonld-context",
    "@graph": [
        {
```

(continues on next page)

(continued from previous page)

```
            "@id": "http://localhost:8001/base1/model1",
            "@type": "TraceModel",
            "inBase": "./" ,
            "hasUnit": "millisecond"
        },
        {
            "@id": "#EnterChatRoom" ,
            "@type": "ObselType"
        },
        {
            "@id": "#SendMsg" ,
            "@type": "ObselType"
        },
        {
            "@id": "#MsgReceived" ,
            "@type": "ObselType"
        },
        {
            "@id": "#LeaveRoom" ,
            "@type": "ObselType"
        },
        {
            "@id": "#room" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#EnterChatRoom"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "room"
        },
        {
            "@id": "#message" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#SendMsg", "#MsgReceived"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "message"
        },
        {
            "@id": "#from" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#MsgReceived"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "from"
        }
    ]
}
```

**Note:** In UML, attributes belong to a given class, and their name is scoped to that class. It is therefore possible to have two different classes `A` and `B`, both having an attribute named `foo`, and still have `A.foo` mean something completely different from `B.foo` (they could for example have different datatypes).

In kTBS on the other hand, attributes are first-class citizens of the model, their name (IRI) is scoped to the entire model. In our example above, the attribute `<#message>` is shared by two obsel types, it is therefore the *same* attribute, with the same meaning and the same datatype[1].

[1] In order to achieve this in UML, we would need an abstract class (*e.g.* `WithMessage`) defining the attribute `message`, and have both classes `SendMsg` and `MsgReceived` inherit that abstract class.

Note that this design is still possible with kTBS, and can be useful when multiple attributes and/or relations are shared together in several obsel types (see (using *Inheritance of obsel types*).

If we wanted to consider `SendMsg.message` and `MsgReceived.message` as two distinct attributes more in the line of UML design, then we would need to create two attribute types with distinct IRIs, for example `<#SendMsg/message>` and `<#MsgReceived/message>`.

### Adding relations

We now define the types of relation that may exist between obsels in our model. Just like obsel types and attributes, relation types are named with an IRI relative to that of the model. The type(s) of the obsels from which the relation can originate is specified with `:hasRelationDomain`. The type(s) of the obsels to which the relation can point is specified with `:hasRelationRange`.

```
{
    "@context": "http://liris.cnrs.fr/silex/2011/ktbs-jsonld-context",
    "@graph": [
        {
            "@id": "http://localhost:8001/base1/model1",
            "@type": "TraceModel",
            "inBase": "./" ,
            "hasUnit": "millisecond"
        },
        {
            "@id": "#EnterChatRoom" ,
            "@type": "ObselType"
        },
        {
            "@id": "#SendMsg" ,
            "@type": "ObselType"
        },
        {
            "@id": "#MsgReceived" ,
            "@type": "ObselType"
        },
        {
            "@id": "#LeaveRoom" ,
            "@type": "ObselType"
        },
        {
            "@id": "#room" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#EnterChatRoom"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "room"
        },
        {
            "@id": "#message" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#SendMsg", "#MsgReceived"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "message"
        },
        {
            "@id": "#from" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#MsgReceived"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
```

(continues on next page)

(continued from previous page)

```
            "label": "from"
        },
        {
            "@id": "#inRoom" ,
            "@type": "RelationType" ,
            "hasRelationOrigin": ["#MsgReceived", "#LeaveRoom", "#SendMsg"] ,
            "hasRelationDestination": ["#EnterChatRoom"]
        }
    ]
}
```

### Inheritance of obsel types

While we can be satisfied with the model above and keep it that way, we can also notice that obsel types `SendMsg` and `MsgReceived` share a lot of things (namely the attribute `message` and being in the domain of `inRoom`). This creates some redundancy in the model definition.

To avoid that redundancy, and capture explicitly the commonalities between those obsel types, we can refactor those commonalities into a new obsel type `MsgEvent` which both `SendMsg` and `MsgReceived` would inherit.

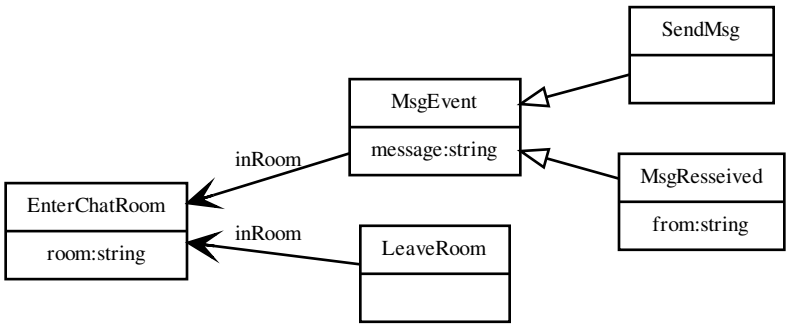
```
{
    "@context": "http://liris.cnrs.fr/silex/2011/ktbs-jsonld-context",
    "@graph": [
        {
            "@id": "http://localhost:8001/base1/model1",
            "@type": "TraceModel",
            "inBase": "./" ,
            "hasUnit": "millisecond"
        },
        {
            "@id": "#MsgEvent" ,
            "@type": "ObselType"
        },
        {
            "@id": "#EnterChatRoom" ,
            "@type": "ObselType"
        },
        {
            "@id": "#SendMsg" ,
            "@type": "ObselType" ,
            "hasSuperObselType": ["#MsgEvent"]
        },
        {
            "@id": "#MsgReceived" ,
            "@type": "ObselType" ,
            "hasSuperObselType": ["#MsgEvent"]
        },
        {
            "@id": "#LeaveRoom" ,
            "@type": "ObselType"
        },
        {
            "@id": "#room" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#EnterChatRoom"] ,
```

(continues on next page)

(continued from previous page)

```
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "room"
        },
        {
            "@id": "#message" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#MsgEvent"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "message"
        },
        {
            "@id": "#from" ,
            "@type": "AttributeType" ,
            "hasAttributeObselType": ["#MsgReceived"] ,
            "hasAttributeDatatype": ["xsd:string"] ,
            "label": "from"
        },
        {
            "@id": "#inRoom" ,
            "@type": "RelationType" ,
            "hasRelationOrigin": ["#MsgEvent", "#LeaveRoom"] ,
            "hasRelationDestination": ["#EnterChatRoom"]
        }
    ]
}
```

This new trace model can be represented by the following UML diagram:




## 2.4 Using kTBS with REST and Turtle

This tutorial aims at showing how to create *kTBS elements* directly through the REST API with Turtle descriptions. If you are not familiar with Turtle or RDF, you might prefer the *JSON* version of that tutorial.

### 2.4.1 Tools

For interacting with the kTBS, we will use the simple HTTP client that is embeded within every HTML page generated by kTBS.

Don't forget, though, that this embeded client is only a convenience shortcut for easily interacting with kTBS. You can just as well use any HTTP client, either interactive (such as curl) or programmatic.

Note also that the Turtle code displayed by the editor might differ from the one presented in those examples; this is because the namespace declarations may not be used in the exact same way. However, they represent the same data.

### 2.4.2 Create and populate a Stored Trace

In this first part, we will create and populate a stored trace. But for this, we first need to create a Base that will host our traces.

#### The kTBS Root

The kTBS root is where all bases live. It is automatically created when the kTBS is first launched. Its URI is that of the kTBS server, in our case: http://localhost:8001/ .

#### Create a new base

To create a new base in our kTBS root, we have to perform an HTTP POST request to it.

Visit the kTBS root and open the embeded HTTP client by clicking to `edit` at the lower-left of the page.

Visit the kTBS root. Select the `POST` operation (this opens a text-area), and ensure that the selected content-type is `text/turtle`. Then copy the following Turtle code in the text-area, and press `Send`.

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

<> :hasBase <base1/>.

<base1/>
    a :Base ;
    skos:prefLabel "My new base" .
```

The URI of the newly created base should appear below the text-area (in our example: http://localhost:8001/base1/). By clicking on it, you will see a Turtle description of your base.

---

**Note:** All URIs in the example above are relative to the URI of the resource *to which we post it*; for example:

- `<>` is interpreted as `<http://localhost:8001/>`,
- `<base1/>` is interpreted as `<http://localhost:8001/base1/>`;

this rule is true for all POST and PUT requests to the kTBS.

---

### Create a stored trace

Creating a stored trace inside a base is very similar to creating a base in the kTBS root. We first need to visit the base (you should already be there after the previous step).

Again, select the `POST` operation, ensure that the content-type is `text/turtle`, copy the following Turtle code in the text-area, and click `Send`. Then click on the link appearing between the text-area.

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .

<> :contains <t01/> .

<t01/>
    a :StoredTrace ;
    :hasModel <http://liris.cnrs.fr/silex/2011/simple-trace-model> ;
    :hasOrigin "1970-01-01T00:00:00Z" .
```

You will notice that, besides the properties that you set for the new trace, the kTBS created another property, hasObselCollection, pointing to http://localhost:8001/base1/t01/@obsels. This is where all the obsels that we are going to add to this trace will be created.

### Add obsels to trace

Adding an obsel to a trace should be no surprise to you at this point: it is simply done by POSTing a description of the obsel to the trace itself.

Simply visit the trace and POST the following content to it:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix m: <http://liris.cnrs.fr/silex/2011/simple-trace-model#> .

<obs1> a m:SimpleObsel ;
    :hasTrace <> .
```

In the description of the new obsel, you will notice that this time the kTBS added a number of properties in addition to the ones you specified above. More precisely, the `begin` and `end` of the obsel have been automatically set based on the moment you posted the obsel; this is expressed in milliseconds since the origin of the trace.

It would have been possible to specify those properties explicitly, if we wanted to override the values automatically computed by the kTBS.

For example, let's go back to the trace and POST the following content to it:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix m: <http://liris.cnrs.fr/silex/2011/simple-trace-model#> .

<obs0> a m:SimpleObsel ;
    :hasTrace <> ;
    :hasBegin 1361462605000 ;
    :hasEnd   1361462647000 .
```

We also note that, as with the base and the trace earlier, we had to mint a URI for our new obsels. As we are likely to create a large number of obsels, it sounds like a good idea to leave it to the kTBS to mint a fresh URI for each of them. For our third obsel, we will therefore use a blank node. We will also add attributes and relations to our new obsel to make it more interesting.

Let's go back to the trace and POST the following content to it:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix m: <http://liris.cnrs.fr/silex/2011/simple-trace-model#> .

[ a m:SimpleObsel ;
  :hasTrace <> ;
  m:value "a new obsel" ;
  m:hasRelatedObsel <obs1> ;
].
```

**Note:** Every element of the kTBS can be created with a blank node instead of an explicit URI. The URI minted by kTBS is returned by the `POST` operation.

If we follow the hasObselCollection link from our trace, to the obsel collection, we can see the three obsels we have created so far (your timestamps will obviously differ):

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix m: <http://liris.cnrs.fr/silex/2011/simple-trace-model#> .

<@obsels> a :StoredTraceObsels .
<.> :hasObselCollection <@obsels> .

<obs0> a m:SimpleObsel;
    :hasBegin 1361462605000;
    :hasEnd 1361462647000;
    :hasTrace <.> .

<obs1> a m:SimpleObsel;
    :hasBegin 1361462685837;
    :hasEnd 1361462685837;
    :hasTrace <.> .

<o-3k> a m:SimpleObsel;
    :hasBegin 1361462707201;
    :hasEnd 1361462707201;
    :hasTrace <.>;
    m:hasRelatedObsel <obs1>;
    m:value "a new obsel" .
```

### 2.4.3 Creating computed traces

The kTBS has a number of *builtin methods* to create Computed Traces. As their name implies, computed trace differ from stored trace by the fact that their obsels are *computed* by the kTBS (in application of the corresponding method) rather than provided by external collectors.

#### Create a Computed Trace with a filter method

Let's go back to the base and create a new computed trace by POSTing the following:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .

<> :contains <filtered1/> .
```

(continues on next page)

(continued from previous page)

```
<filtered1/>
    a :ComputedTrace ;
    :hasMethod :filter ;
    :hasSource <t01/> ;
    :hasParameter "after=1361462641000" .
```

This create a computed trace named `filtered1` based on a *temporal filter* which copies the obsels from `t01` obsels situated *after* timestamp 1361462641000. You may notice that we did not provide any model nor origin for the computed trace; those are automatically computed.

If you go and check the obsel collection of this computed trace, you will find two obsels. More precisely, all obsels from `t01` have been copied, except for `obs0` which has been filtered out, as it is not entierly after timestamp 1361462641000.

### Create a Computed Trace with a SPARQL query

We will now define a more sophisticated computed trace, using the powerful query language SPARQL.

Let's go back to the base and create a new computed trace by POSTing the following:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .

<> :contains <joinRelated1/> .

<joinRelated1/>
    a :ComputedTrace ;
    :hasMethod :sparql ;
    :hasSource <t01/> ;
    :hasParameter """sparql=
PREFIX : <http://liris.cnrs.fr/silex/2009/ktbs#>
PREFIX m:  <http://liris.cnrs.fr/silex/2011/simple-trace-model#>

CONSTRUCT {
    [ a m:SimpleObsel ;
      m:value ?value ;
      :hasTrace <%(__destination__)s> ;
      :hasBegin ?begin ;
      :hasEnd ?end ;
      :hasSourceObsel ?o1, ?o2 ;
    ] .
} WHERE {
    ?o1 :hasBegin ?begin .
    ?o2 :hasEnd ?end ;
        m:hasRelatedObsel ?o1 .
    OPTIONAL { ?o2 m:value ?value }
}""" .
```

This create a computed trace named `joinRelated1` using a SPARQL construct query to builds an obsel for each pair of related obsels in `t01`, inheriting its `begin` and `end` timestamps respectively from each of them.

---

**Note:** It is frequent that SPARQL construct queries build obsels that comply with a model different from the source trace's. The target model can be specified with the special `model` parameter supported by the *sparql method*.

---

#### Create a Computed Trace with a fusion method

We will now use the `fusion` method, used to aggregate in a computed trace the obsels from several source traces.

Let's go back to the base and create a new computed trace by POSTing the following:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .

<> :contains <fusioned1/> .

<fusioned1/>
    a :ComputedTrace ;
    :hasMethod :fusion ;
    :hasSource <filtered1/>, <joinRelated1/> .
```

This creates a computed trace named `fusioned1` which is a merge of the `filtered1` and the `joinRelated1` traces.

## 2.5 Describing a model in Turtle

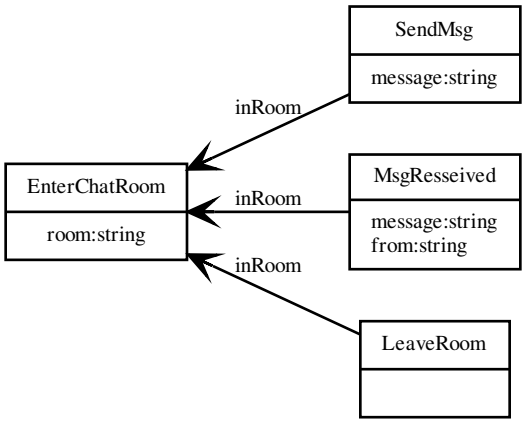
This tutorial explains how to describe a trace model for kTBS.

### 2.5.1 Preparation

First, you have to create a new base as described in the *REST tutorial*. We will assume that this base is named `http://localhost:8001/base01/`.

### 2.5.2 Running example

Here is a UML representation of the Trace Model we will create. This is a minimal trace model of a typical online chat activity.



### 2.5.3 Creating the model

On the page of your base, select the POST request, the `text/turtle` content type, and copy the following data in the text area:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .

<.> :contains <model1> .
<model1> a :TraceModel .
```

then press the `Send` button. The model is created and is now available at http://localhost:8001/base1/model1 .

If you visit that IRI, you will notice that the new model has an additional property: `:hasUnit :millisecond`. Every trace model must have a unit, specifying how time will be represented in the traces complying with this model (see *Trace time management*).

### 2.5.4 Modifying the model

#### Creating obsel types

We will first create the obsel types. Visit the model at http://localhost:8001/base1/model1 , select the PUT request, the `text/turtle` content type, and modify the code with the following:

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

<.> :contains <model1> .

<model1> a :TraceModel ;
    :hasUnit :millisecond .

<#EnterChatRoom> a :ObselType .
<#SendMsg> a :ObselType .
<#MsgReceived> a :ObselType .
<#LeaveRoom> a :ObselType .
```

then press the `Send` button. The page should reload and show the new obsel types.

---

**Note:** Note that all relative IRIs in this example are interpreted against the IRI *of the model* (as it is the target IRI of the PUT request). For the sake of readability, we keep `<model1>` to identify the model itsel, but `<>` would work as well. All components of the models have their IRI starting with `#`, so `<#EnterChatRoom>` is a shorthand for `<http://localhost:8001/base1/model1#EnterChatRoom>`, for example.

Note that you could not have POSTed the Turtle code above as is, as relative IRIs in a POST are interpreted against the IRI of the *base* (`<http://localhost:8001/base01/>` in this case).

It is still possible to create the obsel types together with the model at POST time, but then you need change the relative IRIs accordingly, `<model1#EnterChatRoom>` instead of `<#EnterChatRoom>`, etc.

---

#### Adding attributes

We will now associate attributes to our newly created obsel types.

As obsel types, each attribute has a unique IRI, relative to that of the model: `<#room>`, `<#message>` and `<#from>`. It is related to the obsel type(s) in which it may appear by the `:hasAttributeDomaine` property.

The datatype of an attribute is specified using `:hasAttributeRange`. kTBS supports a subset of the primitive datatypes defined in XML-Schema, including the most usual datatypes such as `xsd:string`, `xsd:integer`, `xsd:boolean` and `xsd:float`.

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

<.> :contains <model1> .

<model1> a :TraceModel ;
    :hasUnit :millisecond .

<#EnterChatRoom> a :ObselType .
<#SendMsg> a :ObselType .
<#MsgReceived> a :ObselType .
<#LeaveRoom> a :ObselType .

<#room> a :AttributeType ;
    skos:prefLabel "room" ;
    :hasAttributeDomain <#EnterChatRoom> ;
    :hasAttributeRange xsd:string .

<#message> a :AttributeType ;
    skos:prefLabel "message" ;
    :hasAttributeDomain <#SendMsg>, <#MsgReceived> ;
    :hasAttributeRange xsd:string .

<#from> a :AttributeType ;
    skos:prefLabel "from" ;
    :hasAttributeDomain <#MsgReceived> ;
    :hasAttributeRange xsd:string .
```

**Note:** In UML, attributes belong to a given class, and their name is scoped to that class. It is therefore possible to have two different classes `A` and `B`, both having an attribute named `foo`, and still have `A.foo` mean something completely different from `B.foo` (they could for example have different datatypes).

In kTBS on the other hand, attributes are first-class citizens of the model, their name (IRI) is scoped to the entire model. In our example above, the attribute `<#message>` is shared by two obsel types, it is therefore the *same* attribute, with the same meaning and the same datatype[1].

If we wanted to consider `SendMsg.message` and `MsgReceived.message` as two distinct attributes more in the line of UML design, then we would need to create two attribute types with distinct IRIs, for example `<#SendMsg/message>` and `<#MsgReceived/message>`.

### Adding relations

We now define the types of relation that may exist between obsels in our model. Just like obsel types and attributes, relation types are named with an IRI relative to that of the model. The type(s) of the obsels from which the relation can originate is specified with `:hasRelationDomain`. The type(s) of the obsels to which the relation can point is specified with `:hasRelationRange`.

[1] In order to achieve this in UML, we would need an abstract class (*e.g.* `WithMessage`) defining the attribute `message`, and have both classes `SendMsg` and `MsgReceived` inherit that abstract class.

Note that this design is still possible with kTBS, and can be useful when multiple attributes and/or relations are shared together in several obsel types (see (using *Inheritance of obsel types*).

```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

<.> :contains <model1> .

<model1> a :TraceModel ;
    :hasUnit :millisecond .

<#EnterChatRoom> a :ObselType .
<#SendMsg> a :ObselType .
<#MsgReceived> a :ObselType .
<#LeaveRoom> a :ObselType .

<#room> a :AttributeType ;
    skos:prefLabel "room" ;
    :hasAttributeDomain <#EnterChatRoom> ;
    :hasAttributeRange xsd:string .

<#message> a :AttributeType ;
    skos:prefLabel "message" ;
    :hasAttributeDomain <#SendMsg>, <#MsgReceived> ;
    :hasAttributeRange xsd:string .

<#from> a :AttributeType ;
    skos:prefLabel "from" ;
    :hasAttributeDomain <#MsgReceived> ;
    :hasAttributeRange xsd:string .

<#inRoom> a :RelationType ;
    :hasRelationDomain <#SendMsg>, <#MsgReceived>, <#LeaveRoom> ;
    :hasRelationRange <#EnterChatRoom> .
```

### Inheritance of obsel types

While we can be satisfied with the model above and keep it that way, we can also notice that obsel types `SendMsg` and `MsgReceived` share a lot of things (namely the attribute `message` and being in the domain of `inRoom`). This creates some redundancy in the model definition.

To avoid that redundancy, and capture explicitly the commonalities between those obsel types, we can refactor those commonalities into a new obsel type `MsgEvent` which both `SendMsg` and `MsgReceived` would inherit.

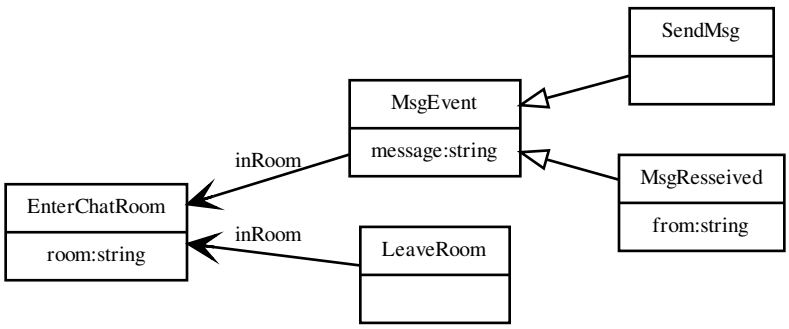
```
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

<.> :contains <model1> .

<model1> a :TraceModel ;
    :hasUnit :millisecond .

<#EnterChatRoom> a :ObselType .
<#MsgEvent> a :ObselType .
<#SendMsg> a :ObselType ;
    :hasSuperObselType <#MsgEvent> .
<#MsgReceived> a :ObselType ;
```

(continues on next page)

(continued from previous page)

```
    :hasSuperObselType <#MsgEvent> .
<#LeaveRoom> a :ObselType .

<#room> a :AttributeType ;
    skos:prefLabel "room" ;
    :hasAttributeDomain <#EnterChatRoom> ;
    :hasAttributeRange xsd:string .

<#message> a :AttributeType ;
    skos:prefLabel "message" ;
    :hasAttributeDomain <#MsgEvent> ;
    :hasAttributeRange xsd:string .

<#from> a :AttributeType ;
    skos:prefLabel "from" ;
    :hasAttributeDomain <#MsgReceived> ;
    :hasAttributeRange xsd:string .

<#inRoom> a :RelationType ;
    :hasRelationDomain <#MsgEvent>, <#LeaveRoom> ;
    :hasRelationRange <#EnterChatRoom> .
```

This new trace model can be represented by the following UML diagram:



CHAPTER 3

# RESTful API

This chapter describes the API of kTBS. The alternative is to use a dedicated API in your programming language of choice, ideally mapped to the *abstract API*. The *developper's section* of this documentation contains a (still incomplete) client API for Python, but Java, ActionScript and PHP implementations are also in development.

## 3.1 REST in kTBS

### 3.1.1 REST basic notions

The central notion in REST is the notion of *resource*. Each resource is identified by an URI and accessed through HTTP operations to that URI. Those operations manipulate *representations* of the resource; the resource itself is an abstract entity that is never directly reached.

**GET:** Return a representation of the resource.

**PUT:** Alter a resource by submitting a new representation.

**POST:** Creates a new resource by submitting its representation to a "parent" resource.

**DELETE:** Deletes a resource.

For more information on the REST architecture, refer to http://en.wikipedia.org/wiki/Representational_state_transfer.

### 3.1.2 Resource relations

kTBS uses two special relations between resources: parent/child resource and aspect resource.

#### Parent/child resource

Resources in kTBS are organized in a hierarchy, and every resource (except for the *Ktbs Root*), has exactly one parent. This hierarchy is naturally reflected by the path of the resources' URIs.

Although the description of the parent resource may contain references to its children resource and some information about them, the general rule is that this information can not be altered by PUT requests to the parent resource. Instead, child resources are:

- created by a POST request to the parent resource;
- altered by a PUT request to themselves;
- deleted by a DELETE request to themselves.

#### Aspect resources

Some resources are too complex to be handled only through the four HTTP verbs. Those resources are therefore linked to one or several *aspect resources*, each of them representing an additional aspect of the original resource.

As a convention, a resource with aspect resources will have a URI ended with '/', and all its aspect resources will have a suffix starting with '@'. The number, types and names of aspect resources depends only on the type of the original resource. Aspect resources are automatically created and deleted with the original resource and can not be created or deleted independandly.

---

**example**

A *trace* has exactly two aspect resources called *@obsels* and *@stats*. If the URI of the trace is http://example.com/ws1/t01/, the URI of the aspect resources will be http://example.com/ws1/t01/@obsels and http://example.com/ws1/t01/@stats, respectively. The first one holds its *obsel* collection, while the second one holds some general statistics about the trace.

---

Although each type defines the name of its aspect resources, it is consider a better practice to *discover* their names through inspection of the resource descriptions, rather than relying on their naming convention. The semantics of the description vocabulary is indeed considered more stable across implementations than the naming conventions.

### 3.1.3 Representations

Resource representations in kTBS are typically in RDF (except for some *aspect resources*), and represented by default in JSON-LD. Through content-negotoation, other syntaxes can be used, such as RDF/XML, Turtle or N-Triples. Support for additional mimetypes can be added to kTBS by using the `plugin mechanism`.

The representations in GET and PUT requests are about existing resources with a known URI. It is therefore quite straightforward to represent those resources in RDF: they appear in the graph as a URI node, and the arcs in the graph represent their relations with other resources and literals. Depending on the type of the represented resources, some *representation constraints* (see below) may apply to the structure of the graph.

On the other hand, the representation of a created resource (POST request) deserves more explanations, since their URI is not necessarily known in advance. The POSTed RDF graph must comply with the following rules:

- the target URI of the POST request must be present in the graph as a URI node;
- it must be linked, through exactly one *postable* property (see below), to a URI or blank node presenting the to-be-created resource;
- appart from the arc described above, the graph must be a valid description of the to-be-created resource (i.e. be compliant with the constraints imposed by its type).

The list of *postable* properties, i.e. property which can be used to link a target resource to a newly created resource, is determined by the type of the target resource.

**example**

*Traces* have only one *postable* property: *has_trace*, which links theirs obsels (created resources) to themselves.

If the node representing the resource to create is a blank node, kTBS will make a fresh URI for it. If it is a URI node, kTBS will check that the URI is not in use, or the creation will fail. In any case, the URI of the newly created resource will be provided in the *Location* header field of the response, as specified by HTTP.

**TODO**

Document how it works with JSON, as it is slightly different.

#### Representation constraints

A common constraint imposed by resource types on the description of their instances is that the graph be *star-shaped*. This implies that:

- every arc in the graph involved the resource being described by the graph;
- the other node in every arc is either a URI or literal node (i.e. no blank node).

Additionnaly, there may be some restrictions on the properties belonging to the following namespaces, since they have a special meaning for kTBS:

- http://liris.cnrs.fr/silex/2009/rdfrest#
- http://liris.cnrs.fr/silex/2009/ktbs#

Properties from those namespaces may be:

**GET-only:** those properties are automatically generated by kTBS. They are part of the GET description, but can not be part of the POSTed description. They may be included in the payload of a PUT only if their value is not modified.

**POST-only:** those properties can be initialized at POST time, but after that, they behave exactly like GET-only properties.

## 3.2 Ktbs Root

This class has exactly one instance in each kTBS, whose URI is the base URI of the kTBS.

### 3.2.1 GET

Retrieve a description of the root. This description contains configuration settings of the kTBS, and references to its children resources, of type *Base*.

It is also possible to enrich the graph returned by GET with information about the bases in this root, using the property `prop` containing a comma-separated list of the following properties:

- `comment` to display the `rdfs:comment`s of bases,
- `label` to display the `rdfs:label`s of `skos:prefLabel` of bases.

### 3.2.2 PUT

Alters the configuration settings of the kTBS. Children resources can not be modified, and may therefore be ommitted from the payload.

### 3.2.3 POST

Creates a new *Base*.

### 3.2.4 Description constraints

The description of a kTBS root must be *star shaped*.

## 3.3 Base

Base resources represent the trace bases hosted by the system.

### 3.3.1 Creation

A base is created by a POST query to the *KtbsRoot*. It must have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#hasBase linking from the KtbsRoot

### 3.3.2 GET

Retrieve a description of the base. This description contains information about the base (label, owner, etc.) and the list of items (*Trace*, *Model*, *Method*) it contains, with their types.

It is also possible to enrich the graph returned by GET with information about the items of the base, using the property `prop` containing a comma-separated list of the following properties:

- `comment` to display the `rdfs:comment`s of items,
- `hasModel` to display the model of traces,
- `hasSource` to display the sources of traces,
- `label` to display the `rdfs:label`s of `skos:prefLabel` of items,
- `obselCount` to display the number of obsels of traces.

### 3.3.3 PUT

This allows to change some information about the base. Items of a base can not be modified that way, and may therefore be ommitted from the payload.

### 3.3.4 DELETE

Deletes the base.

TODO: decide whether a base can be deleted if it is not empty.

### 3.3.5 Description constraints

The description of a base must be *star shaped*.

## 3.4 Trace

This class is the base class of *Stored Trace* and *Computed Trace*. It has no direct instances (only the two subclasses above can be instantiated), but everything described here is valid for any indirect instance.

Traces have two *aspect resource*:

- @*obsels*, linked through http://liris.cnrs.fr/silex/2009/ktbs#hasObselCollection
- @*stats*, linked through http://liris.cnrs.fr/silex/2009/ktbs#hasTraceStatistics

### 3.4.1 Original resource

#### GET

Retrieve the description of the trace, augmented with the following generated properties:

- http://liris.cnrs.fr/silex/2009/ktbs#compliesWithModel can be either "yes", "no" or "?" (if the model is not available)
- links to the @obsel aspect resource.

#### PUT

This allows to change the description of the trace itself.

#### DELETE

Deletes the trace and its aspect resources.

TODO: what happens if the trace is the source of a computed trace?

#### Description Constraints

The description of a trace must be *star shaped*.

### 3.4.2 @obsels

This aspect resource stands for the obsel collection of the trace.

#### GET

Return the description of all the obsels of the trace. This description can be filtered by passing the following query-string arguments:

**after** an obsel URI, only obsels after this one will be returned

**before** an obsel URI, only obsels before this one will be returned

**limit** an int, only that many obsels (at most) will be returned

**minb** an int, the minimum begin value for returned obsels

**mine** an int, the minimum end value for returned obsels

**maxb** an int, the maximum begin value for returned obsels

**maxe** an int, the maximum end value for returned obsels

**offset** an int, skip that many obsels

**reverse** a boolean[1], reverse the order (see below)

For example http://localhost:8001/base1/t01/@obsels?minb=42&maxe=101 will return only those obsel beginning at or after 42 and ending at or before 101.

Some of these parameters (`after`, `before`, `limit`, `offset` and `reverse`) rely on the *Obsel total ordering*. For example, `@obsels?limit=10` will return the first ten obsels, while `@obsels?reverse&limit=10` will return the last ten obsels. Remember however that most RDF serializations have no notion of order (they convey a *set* of triples) so the representation of those resources may appear unordered. The JSON-LD serializer in kTBS is a notable exception: the obsel list is sorted according to the obsel ordering.

Even with unordered serializations, however, this still allows to retrieve obsels of a big trace in a paginated fashion, using `limit` to specify the size of the page, and `after` to browse from one page to another (setting its value to the latest obsel of the previous page), or `before` when paginating in the `reverse` order[2]. To make it easier, kTBS provides a `next` Link HTTP header (per **RFC 5988**) pointing to the next page.

### Representation completeness

Obsels with a complex structure (see below) may not be entirely described in the representations of `@obsels`. More precisely, all attributes (*i.e.* outgoing properties) of obsels, and all inter-obsel relations will *always* be represented. However, if an attribute has a complex value, represented as a blank node with its own properties, then the representations of `@obsels` will usually[3] truncate such property paths to a length of 3.

This limitation has been introduced to ensure good performances, and is deemed acceptable as obsels typically have a flat structure (depth of 1), and occasionnally a depth of 2. In order to get the full description of an obsel, you can of course still get it from the obsel URI.

Also, note that transformation methods sill have access to the whole structure of obsels

### PUT

This method can be used to modify the obsels of a trace. Be aware that kTBS performs *absolutely no control* on the put graph, so this is at your own risks.

### DELETE

This method deletes the obsel collection, and with it all the obsels of the trace.

Note that a trace must always have an obsel collection, so a new (empty) one will immediately be created. Hence, this method can be thought of as simply deleting all obsels, and leaving the obsel collection resource intact.

[1] The value is case insensitive, and any value different from `false`, `no` or `0` will be considered true. Note that the empty string is considered true, so that this parameter can be used without any value, as in `@obsels?reverse&limit=10`.

[2] The `offset` option would be simpler to use, but its use is not always allowed on big traces (for example, Virtuoso forbids it beyond a certain amount of obsels), so using `after`/`before` is more robust (and potentially more efficient).

[3] You *may* retrieve longer paths in some situations, but this should not be relied upon.

### 3.4.3 @stats

This aspect resource provides general statistics about the trace.

#### GET

Provide the statistics; by default, those statistics include:

- the total number of obsels,
- the time-span of the obsels of the traces (min and max timestamp, and duration).

Plugins can add more information to the statistics.

## 3.5 Obsel

### 3.5.1 Creation

An obsel is created by a POST query to a *StoredTrace*. It must have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#hasTrace linking to the Trace

Optionally, it may have the following properties:

- `rdf:type` valued with one or several obsel type defined by the trace model
- http://liris.cnrs.fr/silex/2009/ktbs#hasBegin (see below)
- http://liris.cnrs.fr/silex/2009/ktbs#hasBeginDT (see below)
- http://liris.cnrs.fr/silex/2009/ktbs#hasEnd (see below)
- http://liris.cnrs.fr/silex/2009/ktbs#hasEndDT (see below)
- http://liris.cnrs.fr/silex/2009/ktbs#hasSubject identifying the person/agent being traced
- any attribute or relation defined by the trace model.

#### Specifying temporal bounds

If the trace's *origin* is opaque:

- `hasBegin` must be specified;
- `hasBeginDT` and `hasEndDT` must not used;
- if `hasEnd` is not specified, the obsel will be considered to end at the same timestamp as its begin.

Otherwise, the trace's *origin* is a timestamp:

- if neither `hasBegin` nor `hasBeginDT` is specified, the obsel will be considered to begin at the current time;
- if neither `hasEnd` nor `hasEndDT` is specified, the obsel will be considered to end at the same timestamp as its begin.

### 3.5.2 GET

Return the description of this obsel.

### 3.5.3 DELETE

Deletes this obsel from the trace. Note that this is a *non-monotonic* change.

NB: any other modification to an obsel is made through an amendment (PUT) of the whole *obsel collection*.

## 3.6 Stored Trace

A trace whose obsels are created externally by POSTing them to the trace, and stored by the system. In addition to the interface specified by its superclass *Trace*, this class accepts the following requests.

### 3.6.1 Original resource

#### Creation

A stored trace is created by a POST query to a *Base*. It must have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#contains linking from the Base
- http://liris.cnrs.fr/silex/2009/ktbs#hasModel
- http://liris.cnrs.fr/silex/2009/ktbs#hasOrigin

#### POST

Add an *obsel* to that trace. See the section about *Monotonicity* for a discussion on the constraints.

### 3.6.2 @obsels

#### PUT

This allows to amend the content of the trace.

Note that this allows to modify obsels, although they appear to be child resources of the trace. This is an exception to the *rule* stating that a resource can not be modified by altering its parent resource.

## 3.7 Computed Trace

A trace whose obsels are computed according to a *Method*, either from a number of *source* traces (transformed trace), or from external information (automatically collected trace). Although its obsel collection may be stored for performance issues, it can be assumed to be computed dynamically and always up-to-date w.r.t. the sources.

### 3.7.1 Original resource

#### Creation

A computed trace is created by a POST query to a *Base*. It must have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#contains linking from the Base

- http://liris.cnrs.fr/silex/2009/ktbs#hasMethod

It can optionally have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#hasSource
- http://liris.cnrs.fr/silex/2009/ktbs#hasParameter

### 3.7.2 @obsels

#### GET

In addition to the query-string parameters accepted by any trace, computed traces allow for the parameter `refresh` to control the computation process. Recognized values are:

- **`default` (which is the default value!)** will refresh the trace only if needed (i.e. if its method, its parameters or its sources have changed).
- `no` will prevent any re-computation.
- `yes` or `force` will force the re-computation of the trace, even if nothing has changed.
- `recursive` will recursively force the re-computation of all the sources of the trace, then of the trace itself.

## 3.8 Model

A model is a simple RDF graph defining the obsel types, attributes and relations according to the kTBS vocabulary.

### 3.8.1 Creation

A model is created by a POST query to a *Base*. It must have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#contains linking from the Base
- http://liris.cnrs.fr/silex/2009/ktbs#hasParentModel

It can optionally have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#hasParameter

### 3.8.2 GET

Return the content of the graph.

### 3.8.3 PUT

Change the content of the model.

### 3.8.4 DELETE

Not implemented yet. TODO decide what to do if computed traces use this model.

## 3.9 Method

### 3.9.1 Creation

A computed trace is created by a POST query to a *Base*. It must have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#contains linking from the Base
- http://liris.cnrs.fr/silex/2009/ktbs#inherits

It can optionally have the following properties:

- http://liris.cnrs.fr/silex/2009/ktbs#hasParameter

### 3.9.2 GET

In addition to the query-string parameters accepted by any trace, computed traces allow for the parameter `quick` to force the retrieval of the *current* state of the trace, even if its content is not up to date.

### 3.9.3 PUT

Not implemented yet.

### 3.9.4 DELETE

Not implemented yet. TODO decide what to do if computed traces use this method.

## 3.10 Authentication

For the moment, kTBS does not include any authentication mechanism. However, as a WSGI application, it can be either:

- wrapped in a WSGI middleware managing authentication (see the WSGI website);
- hosted by a standard web server, such as Apache, with its own authentication mechanism.

CHAPTER 4

# Built-in methods

## 4.1 Filter

This method copies the obsels of the source trace if they pass the filter.

**sources** 1

**parameters**

**model** the model of the computed trace

**origin** the origin of the computed trace

**after** the integer timestamp below which obsels are filtered out

**before** the integer timestamp above which obsels are filtered out

**afterDT** the datetime timestamp below which obsels are filtered out

**beforeDT** the datetime timestamp above which obsels are filtered out

**otypes** space-separated list of obsel type URIs; only obsels of these types (or their sub-types) will be kept

**bgp** a SPARQL Basic Graph Pattern used to express additional criteria (see below)

**extensible** no

If parameter `model` (resp. `origin`) is not provided, the model (resp. origin) of the source trace will be used instead.

All filtering parameters are optional. If not specified, they will not constrain the obsels at all. For example, if `otypes` is not provided, obsels will be kepts regardless of their type; on the other hand, if `otypes` is provided, only obsels with their type in the list will be kept.

Datetime timestamps can only be used if the source origin is itself a datetime. If a temporal boundary is given both as an integer and a datetime timestamp, the datetime will be ignored. Note that temporal boundaries are *inclusive*, but obsels must be entirely contained in them.

The `bgp` parameter accepts any SPARQL BGP (i.e. triple patterns, FILTER clauses) used to add further criteria to the obsels to keep in the computed trace. The SPARQL variables `?obs`, `?b` and `?e` are bound respectively to the obsel, its begin timestamp and its end timestamp. The prefix `m:` is bound to the source trace model URI.

## 4.2 Fusion

This method merges the content of all its sources.

**sources** any number

**parameters**

**model** the model of the computed trace

**origin** the origin of the computed trace

**extensible** no

If all source traces have the same model (resp. origin), then parameter `model` (resp. `origin`) is not required, and in that case the computed trace will have the same model (resp. origin) as its source(s).

## 4.3 Finite State Automaton (FSA)

This method applies a Finite State Automaton to detect patterns of obsels in the source trace, and produce an obsel in the transformed trace for each pattern occurence. It is based on the FSA4streams library.

**sources** 1

**parameters**

**model** the model of the computed trace

**origin** the origin of the computed trace

**fsa** the description of the FSA

**extensible** no

If parameter `model` (resp. `origin`) is not provided, the model (resp. origin) of the source trace will be used instead.

The `fsa` parameter expects a JSON description of the FSA, as described in the FSA4streams documentation, with the following specificities:

- The default matcher (even if not explicitly specified) is `obseltype`: each `transition.condition` is interpreted as an obsel type URI (either absolute or relative to the source trace's model URI), and an obsel matches the transition if it has the corresponding obsel type.
- An additional matcher is also provided: `sparql-ask`, which allows for more expressive condition as the previous one. It interprets conditions as the WHERE clause of a SPARQL ASK query, where
  - prefix `:` is bound to the kTBS namespace,
  - prefix `m:` is bound to the source trace model,
  - variable `?obs` is bound to the considered obsel,
  - variable `?pred` is bound to the previous matching obsel (if any),
  - variable `?first` is bound to the first matching obsel of the current match (if any).
- The `max_duration` constraints (as specified in the documentation) apply to the *end* timestamps of the obsels.

- Terminal states may have two additional attributes `ktbs_obsel_type` and `ktbs_attribute`, described below.

For each match found by the FSA, a new obsel is generated:

- The source obsels of the new obsel are all the obsels contributing to the match.
- The begin and end timestamps of the new obsel are, respectively, the begin of the first source obsel and the end of the last source obsel.
- The type of the new obsel is the value of the `ktbs_obsel_type` of the terminal state, interpreted as a URI relative to the computed trace's model URI. If this attribute is omitted, the state identifier is used instead. If this attribute is `null`, no obsel will be generated for the corresponding match.
- If the terminal state has a `ktbs_attributes` model, additional attributes will be generated for the new obsel. The value of `ktbs_attributes` must be a JSON object, whose keys are the *target* attribute type URIs (relative to the computed trace's model URI), and whose values are *source* attribute type URIs (relative to the source trace's model URI). Each target attribute will receive the value of the source attribute of the source obsels. If several values are available, the value of the latest source obsel will be kept. If none of the source obsel has a source attribute, the corresponding target attribute will not be set.

  Additionally, the source attributes can be preceded by one of the following operators, in which case the value of the target operator will be the result of applying the operator to all the values of the source attributes in the source obsels:

  - `last`: returns the last value in chronological order (this is the default, see above);
  - `first`: returns the first value in chronological order;
  - `count`: returns the number of source obsel having the source attribute;
  - `sum`: returns the sum of all values;
  - `avg`: returns the average of all values;
  - `min`: returns the minimum value;
  - `max`: returns the maximum value;
  - `span`: returns the difference between the maximum and the minimum values;
  - `concat`: returns a space-separated concatenation of all the values.

## 4.4 Hubble Rules (HRules)

This method is named after the Hubble project, in which they have been proposed. Those rules can be used both as a stylesheet in the Taaabs timeline component, and as a transformation, thanks to this method. A benefit is that such a transformation can be built interactively in the timeline, with a direct visual feedback of its effect, then "materialized" as a user-defined method and applied to other traces.

**sources** 1

**parameters**

**model** the model of the computed trace

**origin** the origin of the computed trace

**rules** the description of the rules

**extensible** no

If parameter `origin` is not provided, the origin of the source trace will be used instead.

The parameter *model* specified the model of the computed trace. It must be specified (as traces computed from rules generally hava a different model from their source trace).

The parameter *rules* is a JSON string complying with the model below.

### 4.4.1 Structure of the `rules` parameter

The `rule` parameter contains a JSON array. Each item of this array is called a *rule*.

Each *rule* is a JSON object with the following attributes:

- `id` is an obsel type IRI (from the target model), no two rules must have the same `id`.
- `rules` is an array of *subrules*.
- `visible` is an optional boolean, defaulting to `true`.

Each *subrule* is a JSON object with the following attributes:

- `type` is an optional obsel type IRI (from the source trace's model).
- `attribute` is an optional JSON array of *attribute constraints*.

Each *attribute constraint* is a JSON object with the following attributes:

- `uri` is an attribute type IRI (from the source trace's model).
- `operator` is one of the following strings:
  - `==`, `!=`, `<`, `>`, `<=`, `>=` have their usual meaning;
  - `contains` checks that a string attribute contains the given value as a substring.
- `value` is either a JSON string or a JSON-LD value object.

### 4.4.2 Semantics of Hubble rules

- An obsel (from the source trace) matches an attribute constraint if it has the corresponding attribute, and its value satisfies the corresponding operator and value.

  If the `value` of the attribute constraint is a string, and if the attribute has a single datatype as its range in the source model, then the value will be cast to that datatype before the comparison is computed. Otherwise, it will be converted to a literal as specified by JSON-LD.
- An obsel matches a subrule with a `type` if it matches *all* the attribute constraints of the subrule.
- An obsel matches a subrule with a `type` if
  - it has the corresponding obsel type, and
  - it matches *all* the attribute constraints of the subrule.
- An obsel matches a rule if it matches at least one of its subrule.

This method produces a new obsel for each source obsel matching a rule (unless this rule has `visible` set to false); the obsel type of the new obsel is the `id` of the matching rule; the attributes of the source obsel are copied in the new obsel.

### 4.4.3 Precedence of subrules

Whenever an obsel matches several subrules (belonging to different rules), the following precedence applies:

- a subrule which specifies an obsel type has precedence over a subrule which does not, regardless of their number of attribute constraints or position in the rule structure;
- a subrules with more attribute constraints has precedence over a subrule with less attributes constraints, regardless of their position in the rule structure;
- a subrule higher in the rule structure has precedence over a subrule lower in the rule structure.

These criteria induce a total order on subrules, making the process totally deterministic.

## 4.5 Incremental Sparql (ISparql)

This method applies a SPARQL SELECT query to the source trace, and builds new obsels with the result.

**sources** 1

**parameters**

**model** the model of the computed trace

**origin** the origin of the computed trace

**sparql** a SPARQL SELECT query (required)

**extensible** yes (see below)

The SPARQL query must be a SELECT query. Its WHERE clause must contain the magic string `%(__subselect__)s`, which will be replaced with a subquery. For each obsel of the source trace, this subquery will yield its URI, begin timestamp and end timestamp as `?sourceObsel`, `?sourceBegin` and `?sourceEnd`. You may then complement the WHERE clause as you see fit.

Each row returned by your SELECT query will create a new obsel in the computed trace; each variable will add information to the obsel, based on the name of the variable, as explained by the table below. Note that variables followed with a star (*) are mandatory :

| | |
|---|---|
| `sourceObsel` * | a source obsel (`ktbs:hasSourceObsel`), also used to mint the URI of the computed obsel |
| `type` * | the obsel type (`rdf:type`) |
| `begin` * | the begin timestamp (`ktbs:hasBegin`) |
| `end` | the end timestamp (`ktbs:hasEnd`), copied from `begin` if not provided |
| `beginDT` | the begin datetime (`ktbs:hasBeginDT`); note that kTBS does *not* check the consistency with `begin` |
| `endDT` | the end datetime (`ktbs:hasEndDT`), note that kTBS does *not* check the consistency with `end` |
| `subject` | the subject of the obsel (`ktbs:hasSubject`) |
| (any name starting with `sourceObsel`) | an additional source obsel (`ktbs:hasSourceObsel`) |
| (any other name) | an attribute built by concatenating the variable name to the namespace of the computed trace's model |

If parameter `model` (resp. `origin`) is not provided, the model (resp. origin) of the source trace will be used instead.

The SPARQL query can contain magic strings of the form `%(param_name)s`, that will be replaced by the value of an additional parameter named `param_name`.

**Important:** For the sake of performance, this method works *incrementally*: everytime the trace is re-computed, the subquery inserted at `%(__subselect__)s` magically selects only source obsels that have not been considered yet.

As a consequence, each results of the query should **not depend** on information that appears "later" in the trace than `?sourceObsel`. Otherwise, the content of the computed trace may vary unpredictably, depending on when the trace is actually computed.

If you can not guarantee the property above, then you should probably use the *sparql* method below instead, understanding that it will not behave as well performancewise.

## 4.6 Sparql

**Important:** Unlike other methods, this method does not work incrementally: each time the source trace is modified, the whole computed trace is re-generated.

Therefore, you should consider using the *incremental sparql* method instead, unless *its limitations* are too constraining for your needs.

This method applies a SPARQL CONSTRUCT query to the source trace.

**sources** 1

**parameters**

**model** the model of the computed trace

**origin** the origin of the computed trace

**sparql** a SPARQL CONSTRUCT query (required)

**scope** graph against which the SPARQL query must be executed (see below)

**inherit** inherit properties from source obsel (see below)

**extensible** yes (see below)

If parameter `model` (resp. `origin`) is not provided, the model (resp. origin) of the source trace will be used instead.

The `scope` parameter accepts three values:

- `trace` (the default): the SPARQL query only has access to the obsels of the source trace.*
- `base`: the default graph is the union of all the information contained in the base (including subbases). The GRAPH keyword can be used to filter information per graph. Note that this is concetually clean, but very inefficient with the current implementation.
- `store`: the default graph is the entire content of the underlying triple-sore. The GRAPH keyword can be used to filter information per graph. Note that this is only safe if all users are allowed to access any stored information. For this reason, this option is disable by default. To enable it, the configuration `sparql.allow-scope-store` must be set to `true`.

If `inherit` is set (with any value), then the produced obsels will inherit from their source obsel all the properties that are not explicitly set by the CONSTRUCT. That includes properties in the `ktbs` namespace. This allows to greatly simplify SPARQL queries that are mostly filtering and or augmenting obsels, rather than synthetizing new ones. Note however that if the obsel has several source obsels, the behabiour is unspecified. Note also that this mechanism can access the source obsels regardless of the `scope`.

The SPARQL query can contain magic strings of the form `%(param_name)s`, that will be replaced by the value of an additional parameter named `param_name`. Note that the following special parameters are automatically provided:

| special parameter name | replaced by |
|---|---|
| `__destination__` | The URI of the computed trace. |
| `__source__` | The URI of the source trace. |

## 4.7 Composite methods

Composite methods are useful to name a given combination of other methods, in order to make this combination reusable.

There are two distinct methods in this category. Both are currently experimental, and have the following limitations.

**Warning:**

- They create intermediate traces named `_x_name` where `x` is a number, and `name` is the name of the computed trace using the combined method.
- These intermediate traces are currently like any other trace in the base; they can be read, modified, or even deleted. Of course, tampering with them is unadvisable, as composite methods are not robust to such unexpected changes.
- When a computed trace using a composite method is deleted, the intermediate traces are not cleaned up; they must be deleted manually.
- For the moment, composite methods do not allow to set parameters in their component methods.

All composite methods have a required parameter `method`, which is the list of methods that it combines. It is encoded as a space-separated list of absolute URIs.

### 4.7.1 Pipe

This method applies the component methods in sequence.

**sources** any number

**parameters**

**methods** a space-separated list of absolute URIs

**extensible** no

Unlike most methods, the `pipe` method does not accept the `model` or `origin` parameters, as those are specified by the last component method.

### 4.7.2 Parallel

This method applies the component methods in parallel, and merges all resulting traces.

**sources** any number

**parameters**

**methods** a space-separated list of absolute URIs

**model** the model of the computed trace

**origin** the origin of the computed trace

**extensible** no

The `model` and `origin` parameters are handled exactly as the *Fusion* method does it; they are required whenever the outcome of the component methods have a different model (resp. origin).

## 4.8 External

---

**Important:** Unlike other methods, this method does not work incrementally: each time the source trace is modified, the whole computed trace is re-generated.

Also, this method can raise security issues, as it allows users to run arbitrary commands on the kTBS server. For this reason, this method is not anymore provided by default. It is available as a plugin, which must be explicitly enabled.

---

This method invokes an external program to compute a computed trace. The external program is given as a command line, expected to produce the obsels graph of the computed trace.

**sources** any number

**parameters**

**model** the model of the computed trace

**origin** the origin of the computed trace

**command-line** the command line to execute (required)

**format** the format expected and produced by the command line

**min-sources** the minimum number of sources expected by the command-line

**max-sources** the maximum number of sources expected by the command-line

**feed-to-stdin** whether to use the external command standard input (see below)

**extensible** yes (see below)

If parameter `model` (resp. `origin`) is not provided, the model (resp. origin) of the source trace will be used instead.

The command line query can contain magic strings of the form `%(param_name)s`, that will be replaced by the value of an additional parameter named `param_name`. Note that the following special parameters are automatically provided:

| special parameter name | replaced by |
|---|---|
| `__destination__` | The URI of the computed trace. |
| `__sources__` | The space-separated list of the source traces' URIs. |

Parameter `format` is used to inform the kTBS of the format produced by the command line. Default is `turtle`.

Parameters `min-sources` and `max-sources` are used to inform the kTBS of the minimum (resp. maximum) number of sources traces expected by the command line. This is especially useful in user-defined methods, to control that the computed traces using them are consistent with their expectations.

In the general case, the command line is expected to receive the source trace(s) URI(s) as arguments, and query the kTBS to retrieve their obsels. As an alternative, parameter `feed-to-stdin` can be set to have the kTBS send the

source trace obsels directly to the standard input of the external command process. Note that this is only possible when there is exactly one source, and the format used to serialize the obsels will be the same as parameter `format`.

CHAPTER 5

# Developers' documentation

KTBS: Kernel for Trace-Based Systems.

## 5.1 KTBS Client

## 5.2 KTBS API

I provide an implementation of the *Abstract KTBS API*.

This implementation is defined as a set of mix-in classes that rely on the uniform interface of `rdfrest.cores.ICore`. This allows to use those classes in different ways:

- as local instances (for a standalone kTBS embeded in one's application),
- as an HTTP client to a remote kTBS,
- or even as a client through any protocol implemented on top of `rdfrest`.

### 5.2.1 Extensions to the abstract API

I implement the following extensions to the default *Abstract KTBS API*.

#### Extensions suggested by the abstract API

- read-only or read-write properties corresponding to get/set methods
- read-only properties corresponding to list methods
- iter methods corresponding to list methods

Note also that those adaptations are automatically generated by the `extend_api()` class decorator.

#### Specific extensions for function parameters

- Anytime a kTBS element is expected, passing a URI should also work (including a URI relative to the target object). For example, the 'model' argument of `create_stored_trace()` can be a URI (as a unicode or an `URIRef`) rather than an instance of `TraceModel`.
- Datetimes can be used instead of integers for representing timecodes in traces when the trace model and origin allow the conversion.

#### Specific extensions for creation methods

- The *id* parameter is specified in the *Abstract KTBS API* in all creation methods: it is often optional but can be used to set the URI of the resource to create (else, the KTBS will generate a URI). If provided, it must of course be an acceptable URI (not already in use, and subordinated to the parent's URI). As specified by the *Abstract KTBS API*, *id* can be provided as character string, representing a URI either absolute or relative to the parent's URI. The Python implementation also accepts a `rdflib.URIRef` or any object with a *uri* attribute returning a `URIRef`.
- Creation methods also accept an additional parameter *graph* where the user can specify arbitrary properties for the resource to create. This assumes that the resource can be identified in the graph, which is trivial is *id* is provided. However, if the user wants to provide a *graph* but also wants to let the KTBS mint a URI for the created resource, they can use a `blank node`) to represent the resource in *graph*, and pass it to *id*.

### 5.2.2 Resource API

### 5.2.3 KtbsRoot API

### 5.2.4 Base API

### 5.2.5 TraceModel API

### 5.2.6 Trace API

### 5.2.7 Obsel API

### 5.2.8 Method API

### 5.2.9 BuiltinMethod API

### 5.2.10 TraceObsels API

## 5.3 KTBS engine

I provide an implementation of the kTBS engine.

This implementation is defined as a `rdfrest.cores.local.Service`; it reuses the mix-in classes defined in *`ktbs.api`*.

To obtain a KTBS engine conforming with the Ktbs interface of the *Abstract KTBS API*, use `service.make_ktbs()`.

### 5.3.1 Service implementation

### 5.3.2 Resource implementation

### 5.3.3 KtbsRoot implementation

### 5.3.4 Base implementation

### 5.3.5 TraceModel implementation

### 5.3.6 Trace implementation

### 5.3.7 Obsel implementation

### 5.3.8 Method implementation

### 5.3.9 BuiltinMethod implementation

### 5.3.10 TraceObsels implementation

## 5.4 KTBS built-in method implementations

Implementation of the built-in methods shipped with kTBS.

### 5.4.1 Interface

I define the interface of a method implementation.

**class** ktbs.methods.interface.**IMethod**

I define the interface of a method implementation.

**compute_trace_description**(*computed_trace*)

I set the computed properties (model, origin) of the given trace

**Parameters** **computed_trace** – a engine.trace.ComputedTrace

**Return type** rdfrest.util.Diagnosis

The returned diagnosis must be non-empty if the model and/or the origin could not be set, or if it is predicatable that compute_obsels will fail. It can be non-empty in other situations, but the message should then make it clear that it is a mere warning (rather than an error).

Note also that after this method is called, *compute_obsels()* is expected to start afresh.

**compute_obsels**(*computed_trace*, *from_scratch=False*)

I update the obsels of the given computed trace

**Parameters**

- **computed_trace** – a engine.trace.ComputedTrace
- **from_scratch** – force a complete recalculation, regardless of the state of the sources

**Return type** rdfrest.util.Diagnosis

### 5.4.2 Filter

### 5.4.3 Fusion

### 5.4.4 FSA

### 5.4.5 SPARQL

### 5.4.6 External

### 5.4.7 Utilities for implementing method

## 5.5 KTBS shipped plugins

I contain plugins for kTBS.

Additional plugins can be provided by third party. A plugin is a python module containing:

- a *start_plugin(config)* function (accepting a service config, see rdfrest/config.py)
- a *stop_plugin()* if the plugin can be stopped dynamically

(none for the moment)

## 5.6 KTBS auxiliary modules

### 5.6.1 KTBS Namespace

### 5.6.2 KTBS Standalone

### 5.6.3 KTBS Time library

### 5.6.4 KTBS utilities

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

**Note:** This documentation is kept as reStructuredText documents, managed with Sphinx.

# Python Module Index

## k

# Index

## R



## S