
KTBS Documentation

Release 1.0

P.A Champin, F.Conil

July 06, 2017

1	General concepts	3
1.1	Overview of the General Concepts	3
1.2	Obsel	3
1.3	Trace Model	4
1.4	Trace	4
1.5	Method	6
1.6	Monotonicity	6
1.7	Trace time management	7
1.8	Abstract KTBS API	9
2	Tutorials	19
2.1	Installing and running ktBS	19
2.2	Using ktBS with REST and JSON	34
2.3	Describing a model in JSON	39
2.4	Using ktBS with REST and Turtle	45
2.5	Describing a model in Turtle	49
3	RESTful API	55
3.1	REST in ktBS	55
3.2	Ktbs Root	57
3.3	Base	58
3.4	Trace	58
3.5	Obsel	60
3.6	Stored Trace	61
3.7	Computed Trace	61
3.8	Model	62
3.9	Method	63
3.10	Authentication	63
4	Built-in methods	65
4.1	Filter	65
4.2	Fusion	66
4.3	Finite State Automaton (FSA)	66
4.4	Hubble Rules (HRules)	67
4.5	Incremental Sparql (ISparql)	69
4.6	Sparql	70
4.7	Composite methods	71

4.8	External	72
5	Developers' documentation	73
5.1	KTBS Client	73
5.2	KTBS API	73
5.3	KTBS engine	96
5.4	KTBS built-in method implementations	106
5.5	KTBS shipped plugins	107
5.6	KTBS auxiliary modules	107
6	Indices and tables	111
	Python Module Index	113

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:

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 trace 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.

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*.

Obsel total ordering

When orders 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.

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.

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.

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.

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.

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.

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).

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*.

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,
- optionally, a number of parameters,
- optionally, 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.

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.

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).

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.

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.

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.

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.

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*.

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](#)

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, a random opaque origin will be generated and associated to the stored trace.

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`

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.

Abstract KTBS API

Below is a language independent 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`

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.

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*

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*

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*

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?, at-tributes:[AttributeType=>any]?, relations:[(RelationType, Obsel)]?, in-verse_relations:[(Obsel, RelationType)]?, source_obsels:[Obsel]?, label:str?*)

Return type *Obsel*

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.

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

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.

DataGraph (Resource)

This class has no additional method.

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

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*)

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)`

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 [RelationType]

`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)`

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.

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.

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

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 wheezy (> 7.n) and [Ubuntu server](#) (from 12.10 to 14.10), but should be applicable with only minor changes (if any) to other flavours of Linux, and a few adaptation on MacOS or MS Windows¹.

Installing a local kTBS

Make sure you have read and executed *Common Prerequisites* instructions, i.e installed **python developer files** and **virtualenv** system packages.

Create the Python virtual environment

Let us now create a Python virtual environment for kTBS. Once the `python-virtualenv` package has been installed, you have a `virtualenv` command to create your isolated Python virtual environment.

```
$ cd /home/user
$ virtualenv ktbs-env
New python executable in ktbs-env/bin/python
```

¹ a tutorial for installing Python and Virtualenv on Windows is available at <http://www.tylerbutler.com/2012/05/how-to-install-python-pip-and-virtualenv-on-windows-with-powershell/>

```
Installing distribute.....
↪.....
↪.....done.
Installing pip.....done.
```

The virtual environment is then activated by **sourcing** the activate script. Once it is done, you can notice that the Python interpreter used is the virtual environment Python interpreter.

```
user@mymachine:/home/user$ cd ktbs-env/

user@mymachine:/home/user/ktbs-env$ source bin/activate

(ktbs-env)user@mymachine:/home/user/ktbs-env$ which python
/home/user/ktbs-env/bin/python
```

You leave the virtual environment by running the deactivate command.

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ deactivate
user@mymachine:/home/user/ktbs-env#
```

Install kTBS itself

In the activated Python virtual environment, use the **pip**¹ command simply as below, it will install kTBS and its dependencies from the **PyPI**² repository.

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ pip install ktbs
Downloading/unpacking ktbs
  Downloading ktbs-0.3.tar.gz (126Kb): 126Kb downloaded
  Running setup.py egg_info for package ktbs
    /usr/lib/python2.7/distutils/dist.py:267: UserWarning: Unknown distribution_
↪option: 'console'
      warnings.warn(msg)

Downloading/unpacking rdflib==4.1.2 (from ktbs)
...
```

Update kTBS

If you further want to update kTBS to the latest version, just type:

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ pip install ktbs -U
```

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)user@mymachine:/home/user/ktbs-env$ ktbs
INFO 08/06/2015 05:20:49 PM ktbs Using IPV4
INFO 08/06/2015 05:20:49 PM ktbs KTBS server at http://localhost:8001/
```

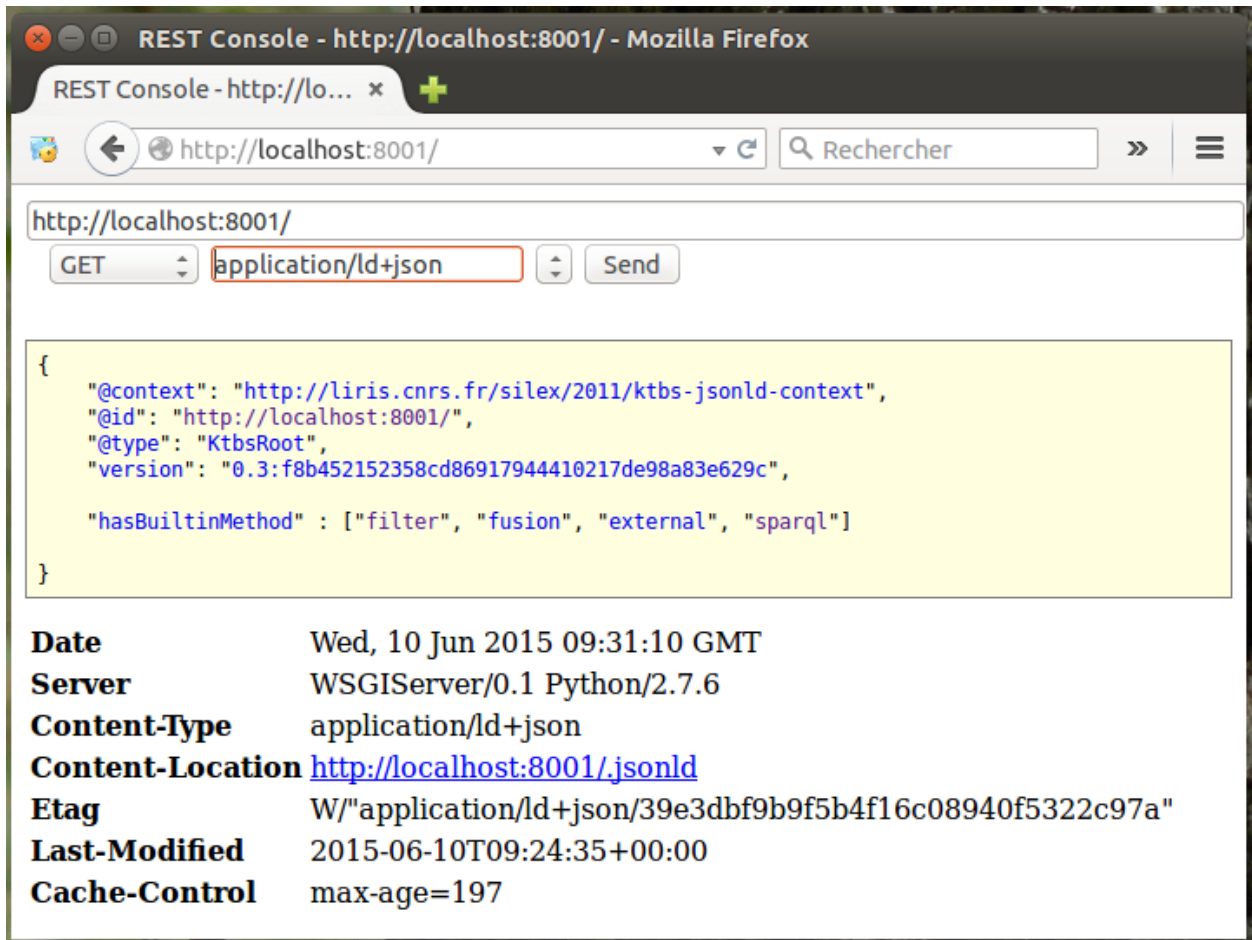
You stop kTBS with **Ctrl-C**.

¹ Python Package Installer

² the Python Package Index

REST console

Enter the kTBS root URL in a browser and use the built-in web interface, named *REST console*, to interact with kTBS.



Command line clients

You can send HTTP requests to kTBS with the `wget` (or `curl`) command line tools, here a GET request with `wget` asking kTBS a response using the Turtle format.

```
$ wget -qO- --header="Accept: text/turtle" http://localhost:8001/

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<> a :KtbsRoot ;
    :hasBuiltinMethod :external,
        :filter,
        :fusion,
        :sparql ;
    :hasVersion "0.3" .
```

The same request with curl :

```
$ curl -H "Accept: text/turtle" http://localhost:8001/
@prefix : <http://liris.cnrs.fr/silex/2009/ktbs#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<> a :KtbsRoot ;
    :hasBuiltinMethod :external,
        :filter,
        :fusion,
        :sparql ;
    :hasVersion "0.3" .
```

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)user@mymachine:/home/user/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).

KTBS help

There are a number of other options for configuring kTBS; to display them with their documentation, type.

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ ktbs --help

Usage: ktbs [options]

HTTP-based Kernel for Trace-Based Systems

Options:
  -h, --help                show this help message and exit
  -H HOST_NAME, --host-name=HOST_NAME
  -p PORT, --port=PORT
  -b BASE_PATH, --base-path=BASE_PATH
  -r REPOSITORY, --repository=REPOSITORY
                           the filename/identifier of the RDF database (default:
                           in memory)
  -c CONFIGFILE, --configfile=CONFIGFILE
  -n NS_PREFIX, --ns-prefix=NS_PREFIX
                           a namespace prefix declaration as 'prefix:uri'
  -P PLUGIN, --plugin=PLUGIN
```

```
loads the given plugin
```

```
Advanced options:
```

```
...
```

KTBS configuration file

If you use a regular set of configuration parameters, you may be interested to use a KTBS configuration file.

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ ktbs -c my-specific-options.conf
```

Here is KTBS configuration file example that you find in KTBS source code³.

```
[server]
# kTBS root uri = "(scheme)://(host-name):(port)(base-path)/"
#scheme = http
#host-name = localhost
#port = 8001

# kTBSroot path, setting "/foo/ktbs" will produce
# "http://localhost:8001/foo/ktbs/" as root uri
# (if you do not change host-name and port)
#base-path =

# Sometimes root-uri is arbitrary (unit tests, ...)
#fixed-root-uri =

# Advanced options
# Force IPv4
#force-ipv4 = false
# Sets the maximum number of bytes of payloads(no limit if unset)
#max-bytes = -1
# Customize Cache-Control header of HTTP server
#cache-control = max-age=1
#no-cache = false ## deprecated, set 'cache-control' to empty string instead
# Serve a policy file allowing Flash applets to connect
#flash-allow = false
# Sets the maximum number of bytes of payloads(no limit if unset)
#max-triples = -1
# Reset connection to RDF store at every HTTP request
# (slower, but safer on some storage systems)
#reset-connection = false
# Include exception traceback in the message of 5xx errors
#send-exception-trueback = false

[ns_prefix]
# A namespace prefix declaration as 'prefix:uri'
# The following hack is used to define a key which will be an empty string
#_ = http://liris.cnrs.fr/silex/2009/ktbs#
#skos = http://www.w3.org/2004/02/skos/core#

[plugins]
post_via_get = false
sparql_endpoints = true
cors = true
```

³ <https://github.com/ktbs/ktbs/blob/develop/examples/conf/ktbs.conf>

```
[sparql]
## WARNING: allowing scope=store in SPARQL methods grants any user
## access to the *whole* triple store. Do *not* use it if
## 1/ you intent to restrict access to some parts of your ktbs to some users, or
## 2/ you are hosting other non-public data in your triple store.
# allow-scope-store = false

[cors]
# Additional plugin options
# Space separated list of allowed origins
# allow-origin = http://trusted.example.org http://another.example.org:12345

[rdf_database]
# The filename/identifier of the RDF database (default: in memory)
#repository =
# Force initialization of repository (assumes -r),
#force-init = false

[logging]
# Choose the modules to log (default None = root ?)
# example : rdfrest ktbs
#loggers =

# Specify the console logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
#console-level = INFO
# Specify the console format
#console-format = %(levelname)s\t%(asctime)s\t%(name)s\t%(message)s

# Specify the filelog filename, no filename implies no logging to file
# example /var/log/ktbs.log
#filename =
# Specify the filelog logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
#file-level = WARN

# Specify the ktbs url for logging to an external ktbs (the ktbs will not
# send log to itself)
# example http://localhost:8008/reflexive/logs
# ktbs-logurl =
# Specify the filelog logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
#ktbs-level = WARN

# Specify a json logging configuration file
# example /etc/ktbs/logging.json
#json-configuration-filename = logging.json
```

Installing a ktBS development version

Instead of installing a stable ktBS version from the **PyPI** repository, you may need to install ktBS from a source code repository to get the latest changes.

Prerequisites

Make sure you have read and executed *Common Prerequisites* instructions, i.e installed **gcc**, **python developer files** and **virtualenv** system packages.

The source code of kTBS is hosted on [GitHub](#). This allows you to get the latest developer version. For this, you need to have [Git](#) installed; if you don't, type.

```
$ sudo apt-get install git
```

Also look for a detailed explanation on how to *create a Python virtual environment*.

Installing kTBS

In the activated Python virtual environment, get the source code and use the `-e` option of the `pip` command to install kTBS from source.

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ git clone https://github.com/ktbs/ktbs.
↪git
Clonage dans 'ktbs'...
remote: Counting objects: 3896, done.
remote: Total 3896 (delta 0), reused 0 (delta 0), pack-reused 3896
Réception d'objets: 100% (3896/3896), 1.40 MiB | 1.66 MiB/s, done.
Résolution des deltas: 100% (2268/2268), done.
Vérification de la connectivité... fait.

(ktbs-env)user@mymachine:/home/user/ktbs-env$ pip install -e ktbs/
```

Note: The `-e` option makes `pip` install the current project in editable mode (i.e. `setuptools` “develop mode”).

It means that whenever you update the repository with `git pull`, or if you edit the code, the changes will be automatically taken into account.

Developer dependencies

If you plan to work on the source code, you might want to install developer dependencies as well.

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ pip install -r requirements.d/dev.txt
```

You can get information on the currently installed version of kTBS with the `ktbs-info` command.

```
(ktbs-env)user@mymachine:/home/user/ktbs-env$ ktbs-infos
(system packages python-dev and zlib2g-dev required by dependencies)
-----
Platform information
-----
System: Linux
Release: 3.13.0-39-generic
Machine: x86_64
sys.platform: linux2
-----
kTBS general information
-----
kTBS version: 0.3
kTBS directory: /home/user/ktbs-env/ktbs
-----
kTBS repository information
-----
```

```
git branch: develop
commit: f8b452152358cd86917944410217de98a83e629c
```

Installing kTBS behind an Apache HTTP server

kTBS can be used behind an Apache HTTP server, this has a number of advantages:

- kTBS does not have to listen on a separate port,
- Apache takes care of running/restarting kTBS when needed,
- kTBS can benefit from the functionalities provided by Apache modules, for example **HTTPS support** or **user authentication**.

To communicate with Apache, kTBS uses the **WSGI** interface, so you need to install the corresponding Apache module: `mod_wsgi`.

Python Web Server Gateway Interface (WSGI)

The Python Web Server Gateway Interface (**WSGI**) is a **Python standard**¹ which proposes a simple and universal interface between web servers and web applications or frameworks.

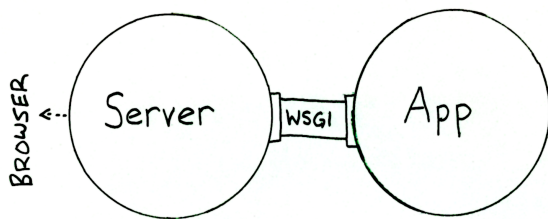


Image from Ian Bicking “WSGI a series of tubes” presentation

Installing and configuring Apache `mod_wsgi`

Installing `mod_wsgi`

The simplest way is to use the system package manager, the module is automatically enabled in Apache.

```
$ sudo apt-get install libapache2-mod-wsgi
```

Note: When using the module packaged in the system, it is linked to a given Python version but it does not matter for the current version of kTBS. This limitation can be solved using the new `mod_wsgi-express` project.

¹ see Python Enhancement Proposals 333 and 3333: <https://www.python.org/dev/peps/pep-0333/>, <https://www.python.org/dev/peps/pep-3333/>

Setting up a kTBS WSGI application interface

Apache needs to be able to **call kTBS** through a **WSGI application interface** - that is just a **Python script** declaring a function called `application` which must be compliant to this specified interface.

That script is provided in the kTBS source tree² at `examples/wsgi/application.wsgi`.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""Runs kTBS behind a WSGI-enabled HTTP server.

This file MUST be accompanied, in the same directory,
by a file with the same name plus the ".conf" extension.

All configuration happens in the .conf file,
you should normally not have to modify this file (the .wsgi file).
"""

####
import atexit

from ktbs.engine.service import get_ktbs_configuration, KtbsService
from rdfrest.util.config import apply_global_config
from rdfrest.http_server import HttpFrontend

ktbs_config_path = __file__ + ".conf"
ktbs_config = get_ktbs_configuration(open(ktbs_config_path))
apply_global_config(ktbs_config)

ktbs_service = KtbsService(ktbs_config)
atexit.register(lambda: ktbs_service.store.close())

application = HttpFrontend(ktbs_service, ktbs_config)
```

As indicated in the header, this script is supposed to be used **as is** and should not be modified: customization is performed through a *kTBS configuration file*.

A configuration file example is provided in the kTBS source tree² at `examples/wsgi/application.wsgi.conf`. Don't forget to adapt it to your own configuration.

```
# this is an example minimal configuration
# for more info on the available configuration options, see examples/conf/ktbs.conf

[server]
host-name = your.domain.org
port = 1234
#send-traceback = false

[rdf_database]
repository =/path/to/sleepycat/database

[logging]
loggers = rdfrest ktbs
console-level = INFO
```

² <https://github.com/ktbs/ktbs>

```
# NB: typically, the WSGI server will redirect the console to its own log,
# including timestamps, so we don't need to include timestamps ourselves.
# On the other hand, being able to distinguish between processes/threads may be
↪useful.
console-format = %(levelname)s [% (process) s/% (thread) d]      %(name) s      %(message) s

[plugins]
sparql_endpoints = true
cors = false

[cors]
# Space separated list of allowed origins
#allow-origin = trusted.example.org another.example.org

# uWSGI users may use the same configuration file (using --ini), as below
#[uwsgi]
#strict = 1
## the line below magically points to the application script
#wsgi-file = %d%n
#http = your.domain.org:1234
```

Warning: The WSGI application will be run as **the user that Apache runs as**. As such, the user that Apache runs as **must have read access** to the WSGI application script file, to all the parent directories that contain it and to the rdf database³.

Configuring Apache

Once the WSGI application interface script is done, you need to add some `mod_wsgi` directives⁴ to Apache configuration files.

On Debian-Ubuntu, this would typically be in `/etc/apache2/sites-available/default` (or `/etc/apache2/sites-available/default-ssl` for https configuration).

For a server configuration, we strongly advise you to use the `/opt` folder instead of the `/home/user` when installing KTBS using *Installing a local KTBS* procedure.

Hint: It is advised to avoid using home directory because any misconfiguration of Apache could end up exposing your whole account for downloading³.

Inside the `<VirtualHost xxx>` Apache directive, add the following lines :

```
<IfModule mod_wsgi.c>
    WSGIScriptAlias /ktbs /opt/ktbs-env/application.wsgi
    WSGIDaemonProcess myktbs processes=1 threads=2 display-name=myktbs python-path=/
↪opt/ktbs-env/ktbs/lib/python2.7/site-packages
    WSGIProcessGroup myktbs
</IfModule>
```

and at the end of the file, **outside the** `<VirtualHost xxx>` **Apache directive**, add the following lines:

³ <https://code.google.com/p/modwsgi/wiki/QuickConfigurationGuide>

⁴ <https://code.google.com/p/modwsgi/wiki/ConfigurationDirectives>


```
<IfModule mod_wsgi.c>
    WSGIPythonHome /opt/ktbs-env
</IfModule>
```

The configuration above may require some adaptation :

- `WSGIScriptAlias` will map `/ktbs` URL to the `/opt/ktbs-env/application.wsgi` WSGI script ;
 - if you want to publish it at a different URL path⁵, change the first argument of `WSGIScriptAlias` accordingly;
 - if you named your WSGI script otherwise and/or stored it elsewhere, change the second argument of `WSGIScriptAlias` accordingly;
- `WSGIDaemonProcess` and `WSGIProcessGroup` define a `Process_group` that `mod_wsgi` uses to manages this group of WSGI applications ;
 - you can adjust the number of threads to your needs in `WSGIDaemonProcess` directive ;
- It assumes that your Python virtual environment is in `/opt/ktbs-env`; if it has a different name, change all occurrences of that path accordingly

For a detailed information on the WSGI directives, please refer to the `mod_wsgi` documentation⁴.

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 your Apache configuration files.

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`.

```
$ cd /opt/ktbs-data
$ 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

Assuming that `/ktbs` is the root url of your kTBS (as specified in the `WSGIScriptAlias` directive above), you can use a `Location` directive to require that access to kTBS is only granted to authenticated users configured in the password file:

```
<Location /ktbs>
    AuthType Basic
    AuthName "Restricted area"
    AuthBasicProvider file
    AuthUserFile /opt/ktbs-data/ktbs-users
    Require valid-user
</Location>
```

⁵ the protocol, server name and port number depend on the enclosing `VirtualHost` directive

Restricting access only to obsel “viewing”

Still assuming that `/ktbs` is the root url of your kTBS, we use here the `LocationMatch` directive to set up different rules for kTBS root, bases and traces access than for obsels access. `LocationMatch` applies the enclosed directives to URLs matching the given “regular expressions”.

The first group of directives let any user send GET and POST request to kTBS root, bases and traces. Any other HTTP request type, such as PUT or DELETE, is only allowed for user “jdoe” once authenticated.

The second group of directives let any user send POST request to kTBS traces. Any other HTTP request type, GET, PUT or DELETE, is only allowed for user “jdoe” once authenticated. Thus the obsels can only be viewed, changed or deleted by “jdoe”.

```
<LocationMatch "^/(ktbs|ktbs/|ktbs/./|ktbs/././)$">
  AuthType Basic
  AuthName "Restricted area"
  AuthBasicProvider file
  AuthUserFile /opt/ktbs-data/ktbs-users

  <LimitExcept GET POST>
    Require user jdoe
  </LimitExcept>
</LocationMatch>

<LocationMatch "^/ktbs/././.$">
  AuthType Basic
  AuthName "Restricted area"
  AuthBasicProvider file
  AuthUserFile /opt/ktbs-data/ktbs-users

  <LimitExcept POST>
    Require user jdoe
  </LimitExcept>
</LocationMatch>
```

per Base or per Trace access control with Apache

If you want to define access control on a per Base or per Trace basis, you can do this by adding several `Location` or `LocationMatch` directives using various authorization schemes that Apache provides. For example:

```
<Location /ktbs>
  # ... # global access control rules
</Location>

<Location /ktbs/base1/>
  # ... # access control for Base base1/
</Location>

<Location /ktbs/base1/t1/>
  # ... # access control for Trace base1/t1/
</Location>

<Location /ktbs/base2/>
```

```
# ... # access control for Base base2/
</Location>
```

Warning: Note that access control in Apache 2.4 differs significantly from Apache 2.2, so check your version and use the appropriate documentation.

Managing access control with kTBS plugins

The `authx` plugin handles authentication (based on OAuth2) and authorization.

Eventually, kTBS may provide more such plugins.

Note that, whenever you want to use HTTP authentication with such a plugin, you will need the `mod_wsgi WSGIPassAuthorization` directive:

```
<Location /ktbs>
    WSGIPassAuthorization On
</Location>
```

Troubleshooting

Does my Apache WSGI work ?

Use the basic `examples/wsgi/hello.wsgi` script provided in the kTBS source tree² as WSGI application interface to check that Apache WSGI correctly works.

```
def application(environ, start_response):
    status = '200 OK'

    if environ.has_key('mod_wsgi.version'):
        output = 'Hello mod_wsgi version {0} !\n\n'.format(environ.get('mod_wsgi.
↪version'))
    else:
        output = 'Hello other WSGI hosting mechanism!'

    output += 'environ content is :\n-----\n'.format(str(environ))
    e_keys = environ.keys()
    e_keys.sort()
    for k in e_keys:
        output += '{0:<30}: {1}\n'.format(k, environ.get(k))

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

Replace the `WSGIScriptAlias WSGI` directive to point to this `hello.wsgi` script.

```
<VirtualHost *:80>

    ...
```

```
<IfModule mod_wsgi.c>
    WSGIScriptAlias /ktbs /opt/ktbs-env/hello.wsgi
    WSGIDaemonProcess myktbs processes=1 threads=2 python-path=/opt/ktbs-env/ktbs/
↪lib/python2.7/site-packages
    WSGIProcessGroup myktbs
</IfModule>

...

```

Invalid data in the repository

It may take some steps to tune kTBS configuration as you want (changing port, base-url, ...) and this may lead to store invalid data to kTBS repository if configured.

In that case try to remove the created repository, kTBS will create a new one.

Do not use mod_python and mod_wsgi on the same server

This is because mod_python will in that case be responsible for initialising the Python interpreter, thereby overriding what mod_wsgi is trying to do. For best results, you should therefore use only mod_wsgi and not try and use mod_python on the same server at the same time⁶.

Check Python version

Check that Python 2.7 is used for main Python, python-dev, virtualenv and mod_wsgi.

Notes

Installing kTBS behind an nginx HTTP server

TODO

Installing kTBS with a Virtuoso backend

kTBS will typically use its own triple-store, which is good for small to medium traces, but does not scale well to big traces (10k obsels or more).

For bigger traces, it is possible to use [Virtuoso](#), a state-of-the-art triple store supporting high volumes of data.

Here are the steps necessary to do so:

- You need to setup a Virtuoso server; you can for example use the Docker image provided at <https://hub.docker.com/r/joernhees/virtuoso/>, and ensure that both ports 8890 and 1111 are accessible to kTBS.
- You need the client ODBC library for accessing Virtuoso, as well as development files for ODBC and Python. On Debian/Ubuntu Linux, you can get it with:

```
apt-get install libvirtodbc0 unixodbc-dev python2.7-dev
```

⁶ <https://code.google.com/p/modwsgi/wiki/VirtualEnvironments>

- Add the following lines in `~/.odbc.ini` or `/etc/odbc.ini`:

```
[VOS]
Description = Open Virtuoso
Driver      = /usr/lib/odbc/virtodbcu_r.so
Address    = localhost:1111
Locale     = en.UTF-8
```

- You need a customized version of the `pyodbc` module, available at <https://github.com/maparent/pyodbc>, in the branch `v3-virtuoso`.

One way to achieve that is the following:

```
pip install git+https://github.com/maparent/pyodbc@v3-virtuoso
```

- Finally, you need the `virtuoso-python` module, available at <https://github.com/maparent/virtuoso-python>.

One way to achieve that is the following:

```
pip install git+https://github.com/maparent/virtuoso-python
```

- Finally, all you need to do is to configure the RDF store used by kTBS to:

```
:Virtuoso:DSN=VOS;UID=dba;PWD=dba;WideAsUTF16=Y
```

Important: On the first run, you need the `--force-init` option to force kTBS to initialize the RDF store.

Warning: The `virtuoso-python` library does not allow several threads of the same process to process queries in parallel, so if you kTBS to do so, you will need to configure it to use distinct *processes* rather than threads (for example in `wsgi_mod` *configuration*).

Common Prerequisites

kTBS is a `Python` application, so you need Python installed; more precisely, you need **version 2.7** of Python. kTBS is not compatible with older version, nor with the newer Python 3. Python 2.7 will typically be already pre-installed on your Linux distribution.

As some dependencies need to be compiled, you will need the **gcc** compiler and **Python developer files**. You can get them with:

```
$ sudo apt-get install gcc python-dev
```

We also advise you to use `virtualenv`, this tool creates an isolated Python environment, so that kTBS and its dependencies can be installed without interference with Python packages installed in your system. To install `virtualenv`, type:

```
$ sudo apt-get install python-virtualenv
```

Notes

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.

Tools

For interacting with the kTBS, we will use the simple HTTP client that is embedded 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.

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"
  }
]
}

```

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 entirely 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": "fused1/",
  "@type": "ComputedTrace",
  "hasMethod": "fusion",
  "hasSource": [ "filtered1/", "joinRelated1/" ]
}
```

This creates a computed trace named `fused1` which is a merge of the `filtered1` and the `joinRelated1` traces.

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.

Describing a model in JSON

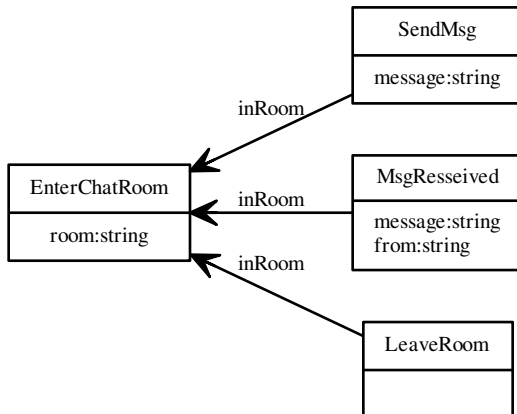
This tutorial explains how to describe a trace model for kTBS.

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/`.

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.



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 embedded 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*).

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": [
    {
      "@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](http://localhost:8001/base1/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": [
    {
      "@id": "http://localhost:8001/basel/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 datatype¹.

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/basel/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"
    }
  ],
}
```

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*).

```

    {
      "@id": "#message" ,
      "@type": "AttributeType" ,
      "hasAttributeObselType": ["#SendMsg", "#MsgReceived"] ,
      "hasAttributeDatatype": ["xsd:string"] ,
      "label": "message"
    },
    {
      "@id": "#from" ,
      "@type": "AttributeType" ,
      "hasAttributeObselType": ["#MsgReceived"] ,
      "hasAttributeDatatype": ["xsd:string"] ,
      "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/basel/model1",
      "@type": "TraceModel",
      "inBase": "./" ,
      "hasUnit": "millisecond"
    },
    {
      "@id": "#MsgEvent" ,
      "@type": "ObselType"
    },
    {
      "@id": "#EnterChatRoom" ,
      "@type": "ObselType"
    },
    {
      "@id": "#SendMsg" ,
      "@type": "ObselType" ,
      "hasSuperObselType": ["#MsgEvent"]
    },
    {
      "@id": "#MsgReceived" ,
      "@type": "ObselType" ,

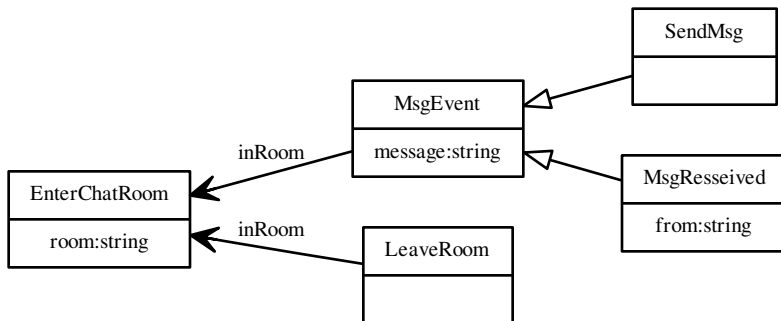
```

```

    "hasSuperObselType": ["#MsgEvent"]
  },
  {
    "@id": "#LeaveRoom" ,
    "@type": "ObselType"
  },
  {
    "@id": "#room" ,
    "@type": "AttributeType" ,
    "hasAttributeObselType": ["#EnterChatRoom"] ,
    "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:



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.

Tools

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

Don't forget, though, that this embedded 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.

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 embedded 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" .
```

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/> .

<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 entirely 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.

Describing a model in Turtle

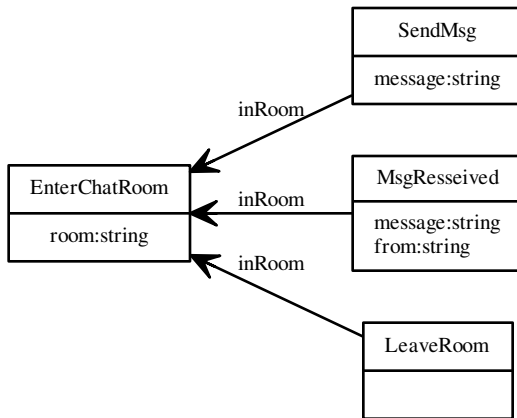
This tutorial explains how to describe a trace model for KTBS.

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/`.

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.



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 <modell> .
<modell> a :TraceModel .
    
```

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

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*).

Modifying the model

Creating obsel types

We will first create the obsel types. Visit the model at <http://localhost:8001/base1/modell> , 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 <modell> .

<modell> 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 <modell> to identify the model itself, 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/modell#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, <modell#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 `:hasAttributeDomain` 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 <modell> .

<modell> 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 datatype¹.

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`.

```
@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 <modell> .

<modell> 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> .
```

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*).

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 <modell> .

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

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

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

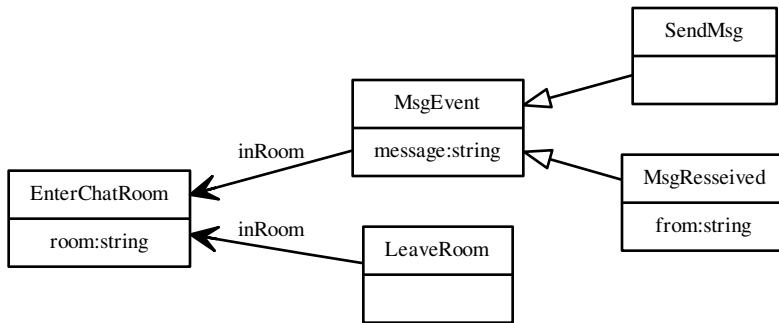
<#message> a :AttributeType ;
          skos:prefLabel "message" ;
          :hasAttributeDomain <#MsgEvent> ;
          :hasAttributeRange xsd:string .

<#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:



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 *developer's section* of this documentation contains a (still incomplete) client API for Python, but Java, ActionScript and PHP implementations are also in development.

REST in kTBS

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.

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 several *aspect resources*, each of them representing only one 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 independently.

example

A *trace* has exactly two aspect resources called *@about* and *@obsels*. 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/@about> and <http://example.com/ws1/t01/@obsels>, respectively. The first one holds the metadata about the trace (date of creation, compliance with its *model*, etc.), while the second one holds its *obsel* collection.

Although each type defines the name of its aspect resources, it is considered 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.

Representations

Resource representations in kTBS are typically in RDF (except for some *aspect resources*), either using the standard [RDF/XML](#) syntax, or the alternative [Turtle](#) syntax. Note also that the [N3](#) syntax is supported, as it is a superset of Turtle. 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;
- apart 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 their 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.

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).

Additionally, 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.

Ktbs Root

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

GET

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

PUT

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

POST

Creates a new *Base*.

Description constraints

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

Base

Base resources represent the trace bases hosted by the system.

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*

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

PUT

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

DELETE

Deletes the base.

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

Description constraints

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

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 one *aspect resource*:

- `@obsels`, linked through <http://liris.cnrs.fr/silex/2009/ktbs#hasObselCollection>

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*.

@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 boolean¹, reverse the order (see below)

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`.

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` order². 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 usually³ 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 occasionally 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 still 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.

Obsel

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 Base
- <http://liris.cnrs.fr/silex/2009/ktbs#hasBegin> (see below)
- <http://liris.cnrs.fr/silex/2009/ktbs#hasEnd> (see below)

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.

It can also have any attribute or relation defined by the trace model.

GET

Return the description of this obsel.

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*.

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.

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.

@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.

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.

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>

@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.

Model

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

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>

GET

Return the content of the graph.

PUT

Change the content of the model.

DELETE

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

Method

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>

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.

PUT

Not implemented yet.

DELETE

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

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.

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 URIs indicating which obsel types must be kept in the computed trace

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 kept 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.

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).

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 occurrence. 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 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.

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` specifies the model of the computed trace. It must be specified (as traces computed from rules generally have a different model from their source trace).

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

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: `==`, `<`, `>`, `<=`, `>=`.
- `value` is either a JSON string or a JSON-LD value object.

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.

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.

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 :

<code>sourceObsel *</code>	a source obsel (<code>ktbs:hasSourceObsel</code>), also used to mint the URI of the computed obsel
<code>type *</code>	the obsel type (<code>rdf:type</code>)
<code>begin *</code>	the begin timestamp (<code>ktbs:hasBegin</code>)
<code>end</code>	the end timestamp (<code>ktbs:hasEnd</code>), copied from <code>begin</code> if not provided
<code>beginDT</code>	the begin datetime (<code>ktbs:hasBeginDT</code>); note that KTBS does <i>not</i> check the consistency with <code>begin</code>
<code>endDT</code>	the end datetime (<code>ktbs:hasEndDT</code>), note that KTBS does <i>not</i> check the consistency with <code>end</code>
<code>subject</code>	the subject of the obsel (<code>ktbs:hasSubject</code>)
(any name starting with <code>sourceObsel</code>)	an additional source obsel (<code>ktbs:hasSourceObsel</code>)
(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 performance-wise.

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 conceptually clean, but very inefficient with the current implementation.
- `store`: the default graph is the entire content of the underlying triple-store. 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 disabled 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 synthesizing new ones. Note however that if the obsel has several source obsels, the behaviour 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
<code>__destination__</code>	The URI of the computed trace.
<code>__source__</code>	The URI of the source trace.

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.

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.

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).

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
<code>__destination__</code>	The URI of the computed trace.
<code>__sources__</code>	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`.

KTBS: Kernel for Trace-Based Systems.

KTBS Client

I provide access to any kTBS (either local or remote).

`ktbs.client.get_ktbs(uri)`

I return the root of a kTBS.

Parameters `uri` (*basestring*) – the URI of this kTBS

Return type `ktbs.api.ktbs_root.KtbsRootMixin`

This assumes that the kTBS already exists, either as a remote server or as a local service. If your goal is to *create* a local kTBS service, you must use `ktbs.engine.service.make_ktbs()` instead.

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 `rdfrast.cores.ICore`. This allows to use those classes in different ways:

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

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 if `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`.

Resource API



I provide the pythonic interface common to all kTBS resources.

```

class ktbs.api.resource.KtbsResourceMixin
    I provide the pythonic interface common to all kTBS resources.

    get_id()
        Return the identifier of this resource.

    get_uri()
        Return the URI of this resource.
  
```

get_label ()

Return a user-friendly label for this resource.

If no label has been set, a default label is generated from this resource's URI.

set_label (value)

Set a user-friendly label for this resource.

reset_label ()

Reset the user-friendly label to its default value.

remove ()

Remove this resource from the kTBS.

get_readonly ()

I return True if this resource can not be modified

get_sync_status ()

I return True if this resource is up to date

This is meaningful only if the resource is a “view” on a remote resource. If the resource is self-contained, it is always up to date.

state

I store and return the result of *get_state*.

This property makes it more concise and hopefully slightly more efficient than to call `rdfrst.cores.ICore.get_state()` each time we need this resource's graph.

Note that this graph must not be edited. We must use `rdfrst.cores.ICore.get_state()` instead.

id

Return the identifier of this resource.

label

Return a user-friendly label for this resource.

If no label has been set, a default label is generated from this resource's URI.

readonly

I return True if this resource can not be modified

sync_status

I return True if this resource is up to date

This is meaningful only if the resource is a “view” on a remote resource. If the resource is self-contained, it is always up to date.

KtbsRoot API



I provide the pythonic interface of `ktbs:KtbsRoot`.

class `ktbs.api.ktbs_root.KtbsRootMixin`

I provide the pythonic interface common to ktbs root.

iter_builtin_methods ()

I list all the builtin methods implemented by this kTBS.

Return type an iterable of *MethodBuiltinMixin*

iter_bases ()

I iter over all elements owned by this base.

Return type an iterable of *BaseMixin*

get_builtin_method (*uri*)

I return the built-in method identified by *uri* if supported.

Return type *MethodBuiltinMixin*

get_base (*id*)

I return the base corresponding to the given URI.

Return type *BaseMixin*

create_base (*id=None, label=None, graph=None*)

Create a new base in this kTBS.

Parameters

- **id** – see *Specific extensions for creation methods*
- **label** – TODO DOC explain
- **graph** – see *Specific extensions for creation methods*

Return type *BaseMixin*

bases

Make a list from `iter_bases`

builtin_methods

Make a list from `iter_builtin_methods`

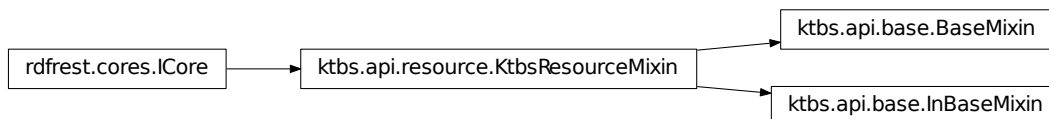
list_bases (**args, **kw*)

Make a list from `iter_bases`

list_builtin_methods (**args, **kw*)

Make a list from `iter_builtin_methods`

Base API



I provide the pythonic interface of `ktbs:Base` .

class `ktbs.api.base.BaseMixin`

I provide the pythonic interface common to bases.

iter_bases ()

Iter over all the sub-bases of this base.

Return type an iterable of *BaseMixin*

iter_traces ()

Iter over all the traces (stored or computed) of this base.

Return type an iterable of *AbstractTraceMixin*

iter_models ()

Iter over all the trace models of this base.

Return type an iterable of *TraceModelMixin*

iter_methods ()

Iter over all the methods of this base.

Return type an iterable of *MethodMixin*

iter_data_graphs ()

Iter over all the methods of this base.

Return type an iterable of *MethodMixin*

get (*id*)

Return one of the element contained in the base.

Parameters *id* (*str*) – the URI of the element; can be relative to the URI of the base

Return type *BaseMixin*, *TraceModelMixin*, *Method* or *AbstractTraceMixin*

get_parent ()

Return the root of the KTBS containing this base.

Return type *KtbsRootMixin*

get_depth ()

Return the distance of this base to the kTBS root.

create_base (*id=None*, *label=None*, *graph=None*)

Create a new base in this kTBS.

Parameters

- **id** – see *Specific extensions for creation methods*
- **label** – TODO DOC explain
- **graph** – see *Specific extensions for creation methods*

Return type *BaseMixin*

create_model (*id=None*, *parents=None*, *label=None*, *graph=None*)

Create a new model in this trace base.

Parameters

- **id** – see *Specific extensions for creation methods*
- **parents** – either None, or an iterable of models from which this model inherits
- **label** – explain.
- **graph** – see *Specific extensions for creation methods*

Return type `TraceModelMixin`

create_method (*id=None, parent=None, parameters=None, label=None, graph=None*)

Create a new computed trace in this trace base.

Parameters

- **id** – see *Specific extensions for creation methods*
- **parent** – parent method (required)
- **parameters** – method parameters
- **label** – explain.
- **graph** – see *Specific extensions for creation methods*

Return type `MethodMixin`

create_stored_trace (*id=None, model=None, origin=None, default_subject=None, label=None, graph=None*)

Create a new store trace in this trace base.

Parameters

- **id** – see *Specific extensions for creation methods*
- **model** – mode of the trace to create (required)
- **origin** – Typically a timestamp. It can be an opaque string, meaning that the precise time when the trace was collected is not known
- **default_subject** – The subject to set to new obsels when they do not specify a subject
- **label** – explain.
- **graph** – see *Specific extensions for creation methods*

Return type `StoredTraceMixin`

create_computed_trace (*id=None, method=None, parameters=None, sources=None, label=None, graph=None*)

Create a new computed trace in this trace base.

Parameters

- **id** – see *Specific extensions for creation methods*
- **method** – method to apply for computation (required)
- **parameters** – parameters of the method
- **sources** – source traces to which the method is applied
- **label** – explain.
- **graph** – see *Specific extensions for creation methods*

Return type `ComputedTraceMixin`

create_data_graph (*id=None, label=None, graph=None*)

Create a new base in this kTBS.

Parameters

- **id** – see *Specific extensions for creation methods*
- **label** – TODO DOC explain

- **graph** – see *Specific extensions for creation methods*

Return type *InBaseMixin*

remove ()

Delete this base from the kTBS.

bases

Make a list from iter_bases

data_graphs

Make a list from iter_data_graphs

depth

Return the distance of this base to the kTBS root.

list_bases (*args, **kw)

Make a list from iter_bases

list_data_graphs (*args, **kw)

Make a list from iter_data_graphs

list_methods (*args, **kw)

Make a list from iter_methods

list_models (*args, **kw)

Make a list from iter_models

list_traces (*args, **kw)

Make a list from iter_traces

methods

Make a list from iter_methods

models

Make a list from iter_models

parent

Return the root of the KTBS containing this base.

Return type *KtbsRootMixin*

traces

Make a list from iter_traces

class `ktbs.api.base.InBaseMixin`

Common mixin for all elements of a trace base.

get_base ()

Return the trace base this element belongs to.

Return type *BaseMixin*

remove ()

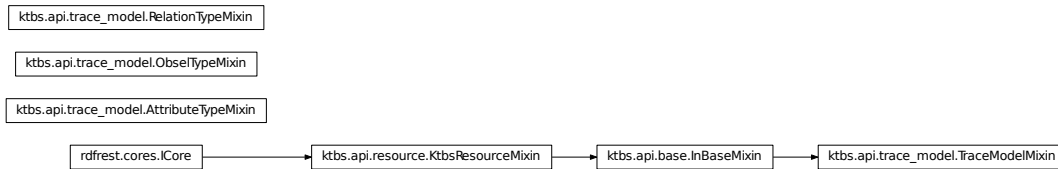
Delete this element from its base.

base

Return the trace base this element belongs to.

Return type *BaseMixin*

TraceModel API



I provide the pythonic interface of `ktbs:TraceModel`.

class `ktbs.api.trace_model.TraceModelMixin`

I provide the pythonic interface to a trace model.

get_unit ()

Return the temporal unit used by this model.

set_unit (*unit*)

Return the temporal unit used by this model.

get (*id*)

Return the pythonic instance corresponding to the given id, or None.

Parameters *id* (*str*) – a URI; can be relative to this Model’s URI

Return type *AttributeTypeMixin*:class, *ObseITypeMixin* or *RelationTypeMixin*

NB: None if the id is not found *but also* if it does not identify a Model element (ObseIType, AttributeType or RelationType).

iter_parents (*include_indirect=False*)

I iter over all the parent models of this model.

iter_attribute_types (*include_inherited=True*)

I iter over the attribute types used in this trace model.

iter_obseI_types (*include_inherited=True*)

I iter over the obseI types used in this trace model.

iter_relation_types (*include_inherited=True*)

I iter over the relation types used in this trace model.

add_parent (*model*)

I add model as a parent model to this model.

remove_parent (*model*)

I remove model as a parent model to this model.

create_obseI_type (*id=None*, *supertypes=()*, *label=None*)

I create a new obseI type in this model.

Parameters

- **id** – see *Specific extensions for creation methods*
- **supertypes** – explain.
- **label** – explain.

Return type *ObselTypeMixin*

create_relation_type (*id=None, origins=None, destinations=None, supertypes=(), label=None*)
I create a new relation type in this model.

Parameters

- **id** – see *Specific extensions for creation methods*
- **origins** – explain.
- **destinations** – explain.
- **supertypes** – explain.
- **label** – explain.

Return type *RelationTypeMixin*

create_attribute_type (*id=None, obsel_types=None, data_types=None, value_is_list=False, label=None*)
I create a new obsel type in this model.

Parameters

- **id** – see *Specific extensions for creation methods*
- **obsel_types** – explain.
- **data_types** – explain.
- **value_is_list** – explain.
- **label** – explain.

Return type *AttributeTypeMixin*

add_supertype (*element=None, element_type=None, super_type=None*)
I add a super type to the element (obsel or relation) type in this model.

Parameters

- **element** – ObselType or RelationType or their URI.
- **element_type** – Type of the concerned element.
- **super_type** – SuperObselType or SuperRelationType.

remove_supertype (*element=None, element_type=None, super_type=None*)
I remove a super type from the element (obsel or relation) type in this model.

Parameters

- **element** – ObselType or RelationType or their URI.
- **element_type** – Type of the concerned element.
- **super_type** – SuperObselType or SuperRelationType.

attribute_types

Make a list from iter_attribute_types

list_attribute_types (**args, **kw*)

Make a list from iter_attribute_types

list_obsel_types (**args, **kw*)

Make a list from iter_obsel_types

list_parents (*args, **kw)

Make a list from iter_parents

list_relation_types (*args, **kw)

Make a list from iter_relation_types

obsel_types

Make a list from iter_obsel_types

parents

Make a list from iter_parents

relation_types

Make a list from iter_relation_types

unit

Return the temporal unit used by this model.

class ktbs.api.trace_model.**AttributeTypeMixin**

I provide the pythonic interface to an attribute type.

iter_obsel_types ()

I iter over the obsel types containing this attribute.

Return type iterable of *ObselTypeMixin*

add_obsel_type (obsel_type)

I add an obsel type to this attribute.

remove_obsel_type (obsel_type)

I remove an obsel type from this attribute.

iter_data_types ()

I iter over the data types allowed for this attribute.

Returns the types as URIRefs

add_data_type (data_type)

I add the data type to this attribute.

remove_data_type (data_type)

I remove the data type from this attribute.

data_types

Make a list from iter_data_types

list_data_types (*args, **kw)

Make a list from iter_data_types

list_obsel_types (*args, **kw)

Make a list from iter_obsel_types

obsel_types

Make a list from iter_obsel_types

class ktbs.api.trace_model.**ObselTypeMixin**

I provide the pythonic interface to an obsel type.

iter_attribute_types (include_inherited=True)

I iter over the attribute types allowed for this type.

iter_relation_types (include_inherited=True)

I iter over the relation types having this obsel type as origin.

iter_inverse_relation_types (*include_inherited=True*)
 I iter over the relation types having this obsel type as destination.

create_attribute_type (*id=None, data_types=None, value_is_list=False, label=None*)
 Call the associated Model method.

create_relation_type (*id=None, destinations=None, supertypes=(), label=None*)
 Call the associated Model method.

attribute_types
 Make a list from iter_attribute_types

inverse_relation_types
 Make a list from iter_inverse_relation_types

list_attribute_types (**args, **kw*)
 Make a list from iter_attribute_types

list_inverse_relation_types (**args, **kw*)
 Make a list from iter_inverse_relation_types

list_relation_types (**args, **kw*)
 Make a list from iter_relation_types

relation_types
 Make a list from iter_relation_types

class `ktbs.api.trace_model.RelationTypeMixin`
 I provide the pythonic interface to a relation type.

iter_origins ()
 I iter over the origin obsel types of this relation.
Return type iterable of *ObselTypeMixin*

iter_destinations ()
 I iter over the destination obsel types of this relation.
Return type iterable of *ObselTypeMixin*

iter_all_origins ()
 I iter over all the origins (direct or inherited) of this relation.
Return type iterator of *ObselTypeMixin*

iter_all_destinations ()
 I iter over all the destinations (direct or inherited) of this relation.
Return type iterator of *ObselTypeMixin*

add_origin (*origin*)
 I add an obsel type to this attribute.

remove_origin (*origin*)
 I remove an obsel type from this attribute.

add_destination (*destination*)
 I add an obsel type to this attribute.

remove_destination (*destination*)
 I remove an obsel type from this attribute.

all_destinations
 Make a list from iter_all_destinations

all_origins

Make a list from iter_all_origins

destinations

Make a list from iter_destinations

list_all_destinations (*args, **kw)

Make a list from iter_all_destinations

list_all_origins (*args, **kw)

Make a list from iter_all_origins

list_destinations (*args, **kw)

Make a list from iter_destinations

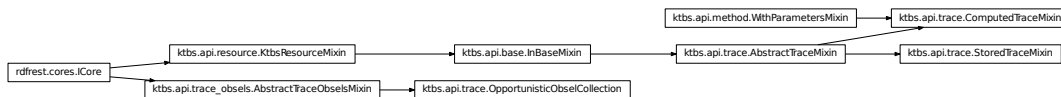
list_origins (*args, **kw)

Make a list from iter_origins

origins

Make a list from iter_origins

Trace API



I provide the pythonic interface of ktbs:StoredTrace and ktbs:ComputedTrace.

class ktbs.api.trace.**AbstractTraceMixin**

I provide the pythonic interface common to all kTBS traces.

get_obsel (*id*)

Return the obsel with the given uri.

Parameters *id* (*str*) – the URI of the obsel; may be relative to the URI of the trace

Return type *ObselMixin*

get_model ()

I return the trace model of this trace.

Return type *TraceModelMixin*

get_origin (*as_datetime=False*)

I return the origin of this trace.

If *as_datetime* is true, *get_origin* will try to convert the return value to datetime, or return it unchanged if that fails.

iter_obsels (*begin=None, end=None, after=None, before=None, reverse=False, bgp=None, limit=None, offset=None, refresh=None*)

Iter over the obsels of this trace.

Return type an iterable of *ObselMixin*

The obsels are sorted by their end timestamp, then their begin timestamp, then their identifier. If `reverse` is true, the order is inverted.

If `given`, `begin` and/or `end` are interpreted as the (included) boundaries of an interval; only obsels entirely contained in this interval will be yielded.

- `begin`: an int, datetime
- `end`: an int, datetime
- `after`: an obsel, URIRef
- `before`: an obsel, URIRef
- `reverse`: an object with a truth value
- `bgp`: an additional SPARQL Basic Graph Pattern to filter obsels
- `limit`: an int
- `offset`: an int
- `refresh`:
 - if “no”, prevent `force_state_refresh` to be called
 - if None or “default”, `force_state_refresh` will be called
 - if “yes” or “force”, will force recomputation of a `ComputedTrace` even if the sources have not change
 - if “recursive”, will recursively force recomputation as above

In the `bgp` parameter, notice that:

- the variable `?obs` is bound each obsel
- the `m`: prefix is bound to the trace model

NB: the order of “recent” obsels may vary even if the trace is not amended, since collectors are not bound to respect the order in begin timestamps and identifiers.

`iter_source_traces ()`

I iter over the sources of this computed trace.

`iter_transformed_traces ()`

Iter over the traces having this trace as a source.

`iter_contexts ()`

I iter over the contexts of this computed trace.

`add_source_trace (val)`

I add a source trace to this trace

`del_source_trace (val)`

I break the link between this trace and one of its source traces

`get_model_uri ()`

I return the URI of the trace model of this trace.

`get_model_prefix (_NICE_SUFFIX=set(['#', '/']))`

I return a prefix-friendly version of the model URI for this trace.

`obsel_collection`

This trace’s obsel collection as a raw resource.

Return type `trace_obsels.AbstractTraceObselsMixin`

trace_statistics

This trace's statistics.

Return type `trace_stats.TraceStatistics`

get_pseudomon_range ()

Return the pseudo-monotonicity range of this trace.

If the pseudo-monotonicity range is X, all changes applied less than X time-units before the last obsel will be considered pseudo-monotonic.

TODO DOC reference to a detailed explanation about monotonicity.

set_pseudomon_range (val)

Return the pseudo-monotonicity range of this trace.

See-also `get_pseudomon_range ()`

TODO DOC reference to a detailed explanation about monotonicity.

get_trace_begin ()

I return the begin timestamp of the obsel.

get_trace_begin_dt ()

I return the begin dateteime of the obsel.

get_trace_end ()

I return the end timestamp of the obsel.

get_trace_end_dt ()

I return the end timestamp of the obsel.

iter_context_uris ()

I iter over the context URIs of this computed trace.

context_uris

Make a list from `iter_context_uris`

contexts

Make a list from `iter_contexts`

list_context_uris (*args, **kw)

Make a list from `iter_context_uris`

list_contexts (*args, **kw)

Make a list from `iter_contexts`

list_obsels (*args, **kw)

Make a list from `iter_obsels`

list_source_traces (*args, **kw)

Make a list from `iter_source_traces`

list_transformed_traces (*args, **kw)

Make a list from `iter_transformed_traces`

model

I return the trace model of this trace.

Return type `TraceModelMixin`

model_prefix

I return a prefix-friendly version of the model URI for this trace.

model_uri

I return the URI of the trace model of this trace.

obsels

Make a list from iter_obsels

origin

I return the origin of this trace.

If *as_datetime* is true, get_origin will try to convert the return value to datetime, or return it unchanged if that fails.

pseudomon_range

Return the pseudo-monotonicity range of this trace.

If the pseudo-monotonicity range is X, all changes applied less than X time-units before the last obsel will be considered pseudo-monotonic.

TODO DOC reference to a detailed explanation about monotonicity.

source_traces

Make a list from iter_source_traces

trace_begin

I return the begin timestamp of the obsel.

trace_begin_dt

I return the begin dateteime of the obsel.

trace_end

I return the end timestamp of the obsel.

trace_end_dt

I return the end timestamp of the obsel.

transformed_traces

Make a list from iter_transformed_traces

class ktbs.api.trace.StoredTraceMixin

I provide the pythonic interface common to kTBS stored traces.

set_model (*model*)

I set the model of this trace. model can be a Model or a URI; relative URIs are resolved against this trace's URI.

set_origin (*origin*)

I set the origin of this trace. origin can be a string or a datetime.

get_default_subject ()

I return the default subject of this trace.

set_default_subject (*subject*)

I set the default subject of this trace.

set_trace_begin (*val*)

I set the begin timestamp of the obsel.

This will automatically unset the trace_begin_dt property.

set_trace_begin_dt (*val*)

I return the begin datetime of the obsel.

This will automatically update the trace_begin property.

set_trace_end (*val*)

I set the end timestamp of the obsel.

This will automatically unset the `trace_end_dt` property.

set_trace_end_dt (*val*)

I return the end datetime of the obsel.

This will automatically update the `trace_end` property.

create_obsel (*id=None, type=None, begin=None, end=None, subject=None, attributes=None, relations=None, inverse_relations=None, source_obsels=None, label=None, no_return=False*)

Creates a new obsel for the stored trace.

Parameters

- **id** – see *Specific extensions for creation methods*.
- **type** – Obsel type, defined by the Trace model.
- **begin** – Begin timestamp of the obsel, can be an int.
- **end** – End timestamp of the obsel, can be an int.
- **subject** – Subject of the obsel.
- **attributes** – explain.
- **relations** – explain.
- **inverse_relations** – explain.
- **source_obsels** – explain.
- **label** – explain.
- **no_return** – if True, None will be returned instead of the created obsel; this saves time and (in the case of a remote KTBS) network traffic

Return type *ktbs.client.obsel.Obsel*

default_subject

I return the default subject of this trace.

model

I return the trace model of this trace.

Return type *TraceModelMixin*

origin

I return the origin of this trace.

If *as_datetime* is true, `get_origin` will try to convert the return value to datetime, or return it unchanged if that fails.

trace_begin

I return the begin timestamp of the obsel.

trace_begin_dt

I return the begin datetime of the obsel.

trace_end

I return the end timestamp of the obsel.

trace_end_dt

I return the end timestamp of the obsel.

class `ktbs.api.trace.ComputedTraceMixin`

I provide the pythonic interface common to kTBS computed traces.

get_method ()

I return the method used by this computed trace

set_method (*val*)

I set the method that this computed trace will use

add_source_trace (*val*)

I override `AbstractTraceMixin.add_source_trace()`

I force the refresh of this trace after the change, as it may have changed computed properties.

del_source_trace (*val*)

I override `AbstractTraceMixin.del_source_trace()`

I force the refresh of this trace after the change, as it may have changed computed properties.

set_parameter (*key, value*)

I override `method.WithParametersMixin.set_parameter()`

I force the refresh of this trace after the change, as it may have changed computed properties.

del_parameter (*key*)

I override `method.WithParametersMixin.del_parameter()`

I force the refresh of this trace after the change, as it may have changed computed properties.

get_diagnosis (*force_refresh=False*)

I return the diagnosis of the last execution of this computed trace.

Note that the diagnosis is None iff the execution succeeded.

get_method_uri ()

I return the URI of the method used by this computed trace

diagnosis

I return the diagnosis of the last execution of this computed trace.

Note that the diagnosis is None iff the execution succeeded.

method

I return the method used by this computed trace

method_uri

I return the URI of the method used by this computed trace

class `ktbs.api.trace.OpportunisticObselCollection` (*trace*)

I implement `rdfrst.cores.ICore` for obsel collections.

Obsel collections in kTBS can become very big, possibly to the point where a server will refuse to serve the full graph at once. Fortunately, they accept a number of query-string parameters to request only “slices” of the collection.

This implementation retrieves slices of the obsel collection in an opportunistic way (*i.e.* finding a trade-off between how much the kTBS is willing to provide at a time and what is actually requested by the code using it).

The graph returned by `get_state()` will always contain all the obsels specified by the *parameters* passed to it, *but possibly more*.

factory (*uri, rdf_types=None, _no_spawn=False*)

I implement `cores.ICore.factory()`.

I simply rely on the factory of my trace.

get_state (*parameters=None*)
 I implement `cores.ICore.get_state()`.

force_state_refresh (*parameters=None*)
 I implement `interface.ICore.force_state_refresh`.
 I simply force a state refresh on my trace.

edit (*parameters=None, clear=False, _trust=False*)
 I implement `cores.ICore.edit()`.
 I try to edit the whole graph.

post_graph (*graph, parameters=None, _trust=False, _created=None, _rdf_type=None*)
 I implement `cores.ICore.post_graph()`.
 Obsel collection do not support `post_graph`.

delete (*parameters=None, _trust=False*)
 I implement `cores.ICore.delete()`.
 Delegate to proper obsel resource.

state
 I implement `cores.ICore.get_state()`.

Obsel API



I provide the pythonic interface `ktbs:Obsel`.

class `ktbs.api.obsel.ObselMixin`

I provide the pythonic interface of `ktbs:Obsel`.

get_trace ()
 I return the trace containing this obsel.

get_obsel_type ()
 I return the obsel type of this obsel.

get_begin ()
 I return the begin timestamp of the obsel.

get_begin_dt ()
 I return the begin timestamp of the obsel.
 We use a better implementation than the standard one.

get_end ()
 I return the end timestamp of the obsel.

get_end_dt ()
 I return the end timestamp of the obsel.

get_subject ()
I return the subject of the obsel.

iter_source_obsels ()
I iter over the source obsels of the obsel.

iter_attribute_types ()
I iter over all attribute types set for this obsel.

iter_relation_types ()
I iter over all outgoing relation types for this obsel.

iter_related_obsels (*rtype*)
I iter over all obsels pointed by an outgoing relation.

iter_inverse_relation_types ()
I iter over all incoming relation types for this obsel.

iter_relating_obsels (*rtype*)
I iter over all incoming relation types for this obsel.

get_attribute_value (*atype*)
I return the value of the given attribut type for this obsel, or None.

attribute_types
Make a list from iter_attribute_types

begin
I return the begin timestamp of the obsel.

begin_dt
I return the begin timestamp of the obsel.
We use a better implementation than the standard one.

end
I return the end timestamp of the obsel.

end_dt
I return the end timestamp of the obsel.

inverse_relation_types
Make a list from iter_inverse_relation_types

list_attribute_types (**args, **kw*)
Make a list from iter_attribute_types

list_inverse_relation_types (**args, **kw*)
Make a list from iter_inverse_relation_types

list_related_obsels (**args, **kw*)
Make a list from iter_related_obsels

list_relating_obsels (**args, **kw*)
Make a list from iter_relating_obsels

list_relation_types (**args, **kw*)
Make a list from iter_relation_types

list_source_obsels (**args, **kw*)
Make a list from iter_source_obsels

obsel_type
I return the obsel type of this obsel.

relation_types

Make a list from `iter_relation_types`

source_obsels

Make a list from `iter_source_obsels`

subject

I return the subject of the obsel.

trace

I return the trace containing this obsel.

class `ktbs.api.obsel.ObselProxy(uri, collection, host_graph, host_parameters)`

I provide a lightweight implementation of `ktbs:Obsel`.

As obsel descriptions can be found in obsel collections, this class provides the Obsel API atop an obsel collection; t

factory (*uri, rdf_types=None, _no_spawn=False*)

I implement `cores.ICore.factory()`.

I simply rely on the factory of my obsel collection.

get_state (*parameters=None*)

I implement `cores.ICore.get_state()`.

I simply return

force_state_refresh (*parameters=None*)

I implement `interface.ICore.force_state_refresh`.

I simply force a state refresh on my host.

edit (*parameters=None, clear=False, _trust=False*)

I implement `cores.ICore.edit()`.

If `self.host_graph` is the complete obsel collection (`host_parameters` is `None`), edit it directly; else, try to get “proper” obsel resource and edit it.

Note that the `clear` argument is not supported in all situations (as host graph may be bigger than the obsel’s state); if you need a clear edit context, you should use a “proper” obsel:

```
obs = obs.factory(obs.uri)
graph = obs.get_state()
```

and then ensure that you fill the edit context with what you got from the proper obsel’s state.

post_graph (*graph, parameters=None, _trust=False, _created=None, _rdf_type=None*)

I implement `cores.ICore.post_graph()`.

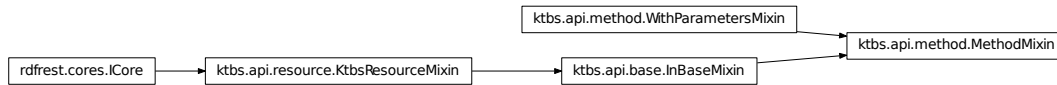
Obsels do not support `post_graph`.

delete (*parameters=None, _trust=False*)

I implement `cores.ICore.delete()`.

Delegate to proper obsel resource.

Method API



I provide the pythonic interface of `ktbs:Method`.

class `ktbs.api.method.WithParametersMixin`

I provide property *parameters* for both `Method` and `ComputedTrace`.

I rely on method `_get_inherited_parameters`, which must return a (possibly empty) fresh dict.

iter_parameters (*include_inherited=True*)

I iter over the parameter of this resource.

get_parameter (*key*)

I return a parameter value.

set_parameter (*key, value*)

I set a parameter value.

del_parameter (*key*)

I delete a parameter value.

parameters_as_dict

I return a fresh dict of the parameters of this computed trace.

NB: the keys and values of the dict are unicode. Values are *not* converted to the datatype expected by the method.

NB: it is assumed that the RDF data is valid, especially that keys are not duplicated.

iter_parameters_with_values (*include_inherited=True*)

I iter over the parameter of this resource.

list_parameters (**args, **kw*)

Make a list from `iter_parameters`

list_parameters_with_values (**args, **kw*)

Make a list from `iter_parameters_with_values`

parameters

Make a list from `iter_parameters`

parameters_with_values

Make a list from `iter_parameters_with_values`

class `ktbs.api.method.MethodMixin`

I provide the pythonic interface of `ktbs:Method`.

get_parent ()

I return the inherited method.

set_parent (*method*)

I set the parent method.

get_parent_uri ()
 I return the URI of the inherited method, if any.

iter_used_by ()
 I iter over all computed traces using this method

iter_children ()
 I iter over all children method of this method

children
 Make a list from iter_children

list_children (*args, **kw)
 Make a list from iter_children

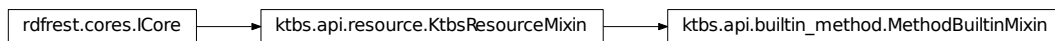
list_used_by (*args, **kw)
 Make a list from iter_used_by

parent
 I return the inherited method.

parent_uri
 I return the URI of the inherited method, if any.

used_by
 Make a list from iter_used_by

BuiltinMethod API



I provide the pythonic interface to kTBS built-in methods.

class `ktbs.api.builtin_method.MethodBuiltinMixin`

I provide the pythonic interface common to kTBS methods.

get_base ()
 A built-in method has no base.

get_parent ()
 A built-in method has no parent.

set_parent (*parent*)
 A built-in method can have no parent.

list_parameters (*include_inherited*)
 A built-in method has no parameter.

get_parameter (*key*)
 A built-in method has no parameters.

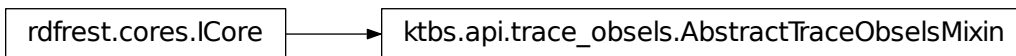
set_paremeter (*key, value*)
 A built-in method can have no parameter.

del_parameter (*key*)
A built-in method has no parameters.

base
A built-in method has no base.

parent
A built-in method has no parent.

TraceObsels API



I provide the pythonic interface to kTBS obsel collections.

class `ktbs.api.trace_obsels.AbstractTraceObselsMixin`
I provide the pythonic interface common to all kTBS obsel collections.

trace
I return the trace owning this obsel collection.

trace_uri
I return the trace owning this obsel collection.

build_select (*begin=None, end=None, after=None, before=None, reverse=False, bgp=None, limit=None, offset=None, selected='?obs'*)
Build a SPARQL query listing the obsels of this trace.

Return type `rdflib.query.Result`

NB: the SPARQL query includes no PREFIX definition, in order to be embeddable as a subquery, but it *requires* the prefix `ktbs` to be declared (bound to the kTBS namespace, of course).

The obsels are sorted by their end timestamp, then their begin timestamp, then their identifier. If *reverse* is true, the order is inverted.

If given, *begin* and/or *end* are interpreted as the (included) boundaries of an interval; only obsels entirely contained in this interval will be yielded.

- begin*: an int, datetime
- end*: an int, datetime
- after*: an obsel, URIRef
- before*: an obsel, URIRef
- reverse*: an object with a truth value
- bgp*: an additional SPARQL Basic Graph Pattern to filter obsels
- limit*: an int
- selected*: the selected variables

In the *bgp* parameter, notice that:

- the variable *?obs* is bound each obsel
- the *m:* prefix is bound to the trace model

NB: the order of “recent” obsels may vary even if the trace is not amended, since collectors are not bound to respect the order in begin timestamps and identifiers.

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()`.

Service implementation

I provide an entry point to create a local kTBS.

`ktbs.engine.service.make_ktbs` (*root_uri=None, repository=None, create=None*)

I create a kTBS engine conforming with the *Abstract KTBS API*.

Parameters

- **root_uri** – the URI to use as the root of this kTBS (defaults to `<ktbs:/>`)
- **repository** – where to store kTBS data
- **create** – whether the data repository should be initialized; (see below)

Parameter *repository* can be either a path (in which case data will be stored in a directory of that name, which will be created if needed), or a string of the form `":store_type:configuration_string"` where *store_type* is a registered store type in `rdflib`, and *configuration_string* is used to initialize this store.

If *repository* is omitted or `None`, a volatile in-memory repository will be created.

If the repository is in-memory or a non-existing path, it will be initialized. In all other cases (i.e. existing path or explicit *store_type*), the repository is assumed to be already initialized; the `force-init` option in the `rdf_database` section can be set to force the initialization in those cases.

class `ktbs.engine.service.KtbsService` (*service_config=None*)

The kTBS service.

get (*uri, rdf_types=None, _no_spawn=False*)

I override `rdfrest.cores.local.Service.get()`

If the original implementation returns `None`, I try to make an `Obsel`.

classmethod `init_ktbs` (*service*)

I populate the root resource’s graph of a new service.

As base creation is protected by the KtbsRoot lock, we are sure that this method can not be called concurrently. So rather than taking the existing semaphore as is and hoping it is correctly set, we force it to 1.

That way, if a previous kTBS didn't clean up its semaphores, it won't block a new instance.

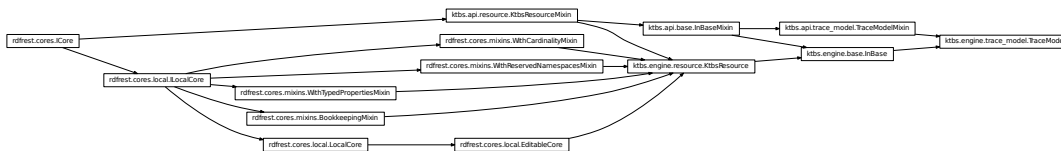
```
class ktbs.engine.base.InBase (service, uri)
    I provide common implementation of all elements contained in a base.

    ack_delete (parameters)
        I override rdfrest.util.DeletableMixin.ack_delete().

    delete (parameters=None, _trust=False)
        I override rdfrest.cores.local.EditableCore.delete().

    edit (*args, **kwds)
        I override rdfrest.cores.local.EditableCore.edit().
```

TraceModel implementation



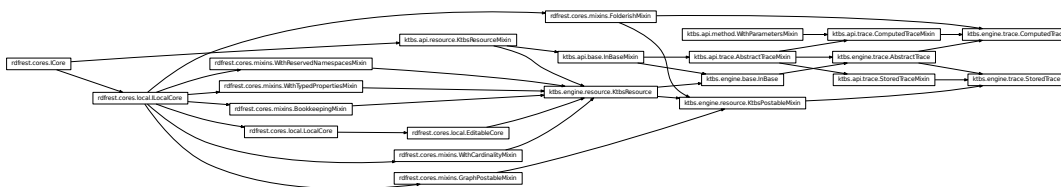
I provide the implementation of ktbs:TraceModel .

```
class ktbs.engine.trace_model.TraceModel (service, uri)
    I provide the implementation of ktbs:TraceModel .

    classmethod complete_new_graph (service, uri, parameters, new_graph, resource=None)
        I implement ILocalCore.complete_new_graph().

        At create time, I add default values for missing information about the trace model.
```

Trace implementation



I provide the implementation of ktbs:StoredTrace and ktbs:ComputedTrace .

class `ktbs.engine.trace.AbstractTrace` (*service, uri*)

I provide the implementation of `ktbs:AbstractTrace` .

unit

I return this trace's time unit.

I get it from the model if available, and store it in the trace's metadata in case the model is not available.

obsel_collection

I override `api.trace.AbstractTrace.obsel_collection`.

Instead of an `OpportunisticObselCollection`, I return the actual resource, as there is no need to optimize transfer in a local implementation.

classmethod `check_new_graph` (*service, uri, parameters, new_graph, resource=None, added=None, removed=None*)

I implement `check_new_graph()`

I check that the sources exist and are in the same base.

classmethod `create` (*service, uri, new_graph*)

I implement `create()`

I create the obsel collection and the statistics resource associated with this trace, and I notify this trace's sources.

prepare_edit (*parameters*)

I overrides `rdfrast.cores.local.ILocalCore.prepare_edit()`

I store old values of some properties (sources, pseudomon range) to handle the change in `ack_edit()`.

ack_edit (*parameters, prepared*)

I overrides `rdfrast.cores.local.ILocalCore.ack_edit()`

I reflect changes in the related resources (sources, obsel collection).

check_deletable (*parameters*)

I implement `check_deletable()`

I refuse to be deleted if I am the source of another trace.

ack_delete (*parameters*)

I implement `ack_delete()`

class `ktbs.engine.trace.StoredTrace` (*service, uri*)

I provide the implementation of `ktbs:StoredTrace` .

classmethod `check_new_graph` (*service, uri, parameters, new_graph, resource=None, added=None, removed=None*)

I implement `check_new_graph()`

I check the temporal extension of this trace.

classmethod `complete_new_graph` (*service, uri, parameters, new_graph, resource=None*)

I implement `ILocalCore.complete_new_graph()`.

At create time, I add default values for missing information about the trace.

check_new (*created*)

I override `GraphPostableMixin.check_new()`.

I check if the created resource exists in my obsel collection.

post_graph (*graph, parameters=None, _trust=False, _created=None, _rdf_type=None*)

I override `rdfrast.util.GraphPostableMixin.post_graph()`.

I allow for multiple obsels to be posted at the same time.

get_created_class (*rdf_type*)

I override `rdfrst.cores.mixins.GraphPostableMixin.get_created_class` Only obsels can be posted to a trace.

class `ktbs.engine.trace.ComputedTrace` (*service, uri*)

I provide the implementation of `ktbs:ComputedTrace` .

classmethod **create** (*service, uri, new_graph*)

I implement `AbstractTrace.create()`

I notify this trace's method, and I ensure that it is run once to compute the computed properties (model, origin) of this trace.

prepare_edit (*parameters*)

I overrides `rdfrst.cores.local.ILocalCore.prepare_edit()`

I store old values of some properties (parameters, method) to handle the change in `ack_edit()`.

ack_edit (*parameters, prepared*)

I overrides `rdfrst.cores.local.ILocalCore.ack_edit()`

I reflect changes in the related resources (method, obsel collection).

ack_delete (*parameters*)

I override `ack_delete()`

I notify my method that I'm no longer using it.

get_state (*parameters=None*)

I override `get_state()`

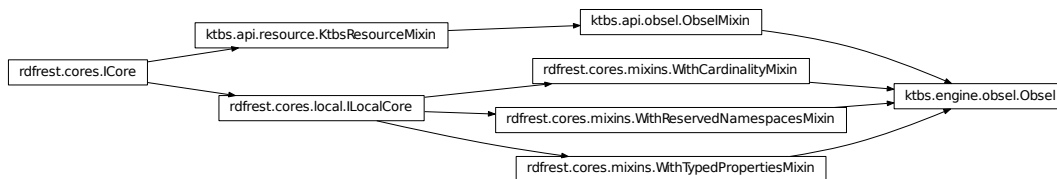
I systematically call `force_state_refresh()` to ensure all computations have been performed.

force_state_refresh (*parameters=None*)

I override `force_state_refresh()`

I recompute my data if needed.

Obsel implementation



I provide the implementation of `ktbs:Obsel` .

class `ktbs.engine.obsel.Obsel` (*trace, uri*)

I provide the implementation of `ktbs:Obsel` .

`ktbs.engine.obsel.get_obsel_bounded_description` (*node, graph, fill=None*)

I override `rdfrst.util.bounded_description()` for obsels.

In other words, it does not change as long as obsels are added after all existing obsels, without any relation to previous obsels.

set_str_mon_tag (*val*)

Set the str_mon_tag.

See-also get-str_mon_tag ()

str_mon_tag

Return the strictly monotonic tag.

This tag only changes when the obsel collection is modified in a non-strictly monotonic way.

In other words, it does not change as long as obsels are added after all existing obsels, without any relation to previous obsels.

get_pse_mon_tag ()

Return the pseudo-monotonic tag.

This tag only changes when the obsel collection is modified in a non-pseudo-monotonic way.

In other words, it does not change as arcs are only added between obsels within the pseudo-monotonicity range (see get_pseudomon_range> ()) of the trace.

set_pse_mon_tag (*val*)

Set the pse_mon_tag.

See-also get-pse_mon_tag ()

pse_mon_tag

Return the pseudo-monotonic tag.

This tag only changes when the obsel collection is modified in a non-pseudo-monotonic way.

In other words, it does not change as arcs are only added between obsels within the pseudo-monotonicity range (see get_pseudomon_range> ()) of the trace.

get_log_mon_tag ()

Return the logically monotonic tag.

This tag only changes when the obsel collection is modified in a non-monotonic way.

In other words, it does not change as long as arcs are only added to the graph, not removed.

set_log_mon_tag (*val*)

Set the log_mon_tag.

See-also get-log_mon_tag ()

log_mon_tag

Return the logically monotonic tag.

This tag only changes when the obsel collection is modified in a non-monotonic way.

In other words, it does not change as long as arcs are only added to the graph, not removed.

add_obsel_graph (*graph*, *_trust=True*)

Add an obsel described in *graph*.

If you need to add only one obsel, you can use this method *without* using any edit () context.

If you need to call this method multiple times, it is more efficient to wrap all the calls to add_obsel_graph inside a single edit () context, which then *must* have the add_obsels_only parameter set.

This should be used instead of the `edit()` context when no arc has to be removed, as it will not change the `log_mon_tag:meth`.

get_state (*parameters=None*)

I override `get_state()`

I support some parameters to get “slices” of the obsel collection. Note that, contrarily to what the interface specifies, slice graphs are static copies of the data; they are not automatically updated, and the slicing parameters are not supported by `~rdfrest.cores.ICore.force_state_refresh:meth`.

I consider an empty dict as equivalent to no dict.

check_parameters (*to_check, parameters, method*)

I implement `check_parameters()`

I also convert parameters values from strings to usable datatypes.

prepare_edit (*parameters*)

I overrides `rdfrest.cores.local.ILocalCore.prepare_edit()`

I store old values related to monotonicity. change in `ack_edit()`.

ack_edit (*parameters, prepared*)

I override `rdfrest.cores.local.ILocalCore.ack_edit()` to update bookkeeping metadata and force transformed trace to refresh.

delete (*parameters=None, _trust=False*)

I override `KtbsResource.delete()`.

Deleting an obsel collection simply empties it, but does not actually destroy the resource.

iter_etags (*parameters=None*)

I override `rdfrest.cores.mixins.BookkeepingMixin._iter_etags()`

I return self.etag, plus the appropriate monotonicity tag depending on the given parameters.

IMPORTANT: the only parameter actually used to determine monotonicity etags are ‘maxe’ and ‘before’, as it would be too costly to determine it for other parameters. Note however that `get_state()` does use this method with an accurate ‘maxe’ value (based on the actual obsels rather than on paremeters), in order to precisely get etags.

class `ktbs.engine.trace_obsels.StoredTraceObsels` (*service, uri*)

I provide the implementation of `ktbs:StoredTraceObsels`

class `ktbs.engine.trace_obsels.ComputedTraceObsels` (*service, uri*)

I provide the implementation of `ktbs:ComputedTraceObsels`

get_state (*parameters=None*)

I override `get_state()`

I support parameter ‘refresh’ to bypass the updating of the obsels, or force a recomputation of the trace.

force_state_refresh (*parameters=None*)

I override `force_state_refresh()`

I recompute the obsels if needed.

edit (*parameters=None, clear=False, _trust=False*)

I override `KtbsResource.edit()`.

delete (*parameters=None, _trust=False*)

I override `AbstractTraceObsels.delete()`.

You can not empty a computed trace.

KTBS built-in method implementations

Implementation of the built-in methods shipped with kTBS.

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`

Filter

Implementation of the filter builtin methods.

Fusion

Implementation of the fusion builtin methods.

FSA

Implementation of the fsa builtin methods.

`ktbs.methods.fsa.match_obseltype` (*transition*, *event*, *token*, *fsa*)

The ‘obseltype’ matcher.

With this matcher, transition conditions are interpreted as obseltype URIs.

`ktbs.methods.fsa.match_sparql_ask` (*transition*, *event*, *token*, *fsa*)

The ‘sparql-ask’ matcher.

With this matcher, transition conditions are interpreted as the WHERE clause of a SPARQL Ask query, where variable ?obs is bound to the considered obsel, and prefix m: is bound to the source trace URI.

SPARQL

Implementation of the sparql builtin methods.

External

Implementation of the external builtin methods.

IMPORTANT: this method allows kTBS users to run arbitrary commands on the server, with the privileges of the user running kTBS. Therefore, it is mostly intended for single-user localhost instances of kTBS.

```
ktbs.plugins.meth_external.start_plugin(_config)
    I get the configuration values from the main kTBS configuration.
```

Note: This function is called automatically by the kTBS. It is called once when the kTBS starts, not at each request.

Utilities for implementing method

Utility functions for method implementations.

```
ktbs.methods.utils.replace_obsels(computed_trace, raw_graph, inherit=False)
    Replace the @obsels graph of computed_trace with raw_graph.
```

If *raw_graph* contains blank obsels, a URI will be generated for them. Except for that, no processing or verification is done on *raw_graph*, so it must be valid.

```
ktbs.methods.utils.translate_node(node, transformed_trace, src_uri, multiple_sources, prevent=None)
    If node is a URI, translate its URI to put it in transformed_trace. Else, leave it unchanged.
```

```
ktbs.methods.utils.copy_obsel(obsel_uri, computed_trace, source_trace, new_obs_uri=None, check_new_obs=None)
    I prepare a graph for an transformed obsel being a copy of obsel.
```

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 `rdffrest/config.py`)
- a `stop_plugin()` if the plugin can be stopped dynamically

(none for the moment)

KTBS auxiliary modules

KTBS Namespace

I contain useful namespace objects, as well as the definition of the kTBS vocabulary.

This module can be run as a program to generate the description:

- with no argument, it will output the original version (Turtle, with a reader-friendly layout);
- with an rdflib format as its argument, it will first convert it to that format, but the result might not be as reader-friendly.

KTBS Standalone

This is a standalone version of an HTTP-based KTBS.

`ktbs.standalone.main()`

I launch KTBS as a standalone HTTP server.

`ktbs.standalone.parse_configuration_options(options=None)`

I get KTBS default configuration options and override them with command line options.

Parameters `options` – Command line options.

Returns Configuration object.

`ktbs.standalone.build_cmdline_options()`

I build ktbs command line options.

`ktbs.standalone.parse_options()`

I parse `sys.argv` for the main.

`ktbs.standalone.number_callback(_option, opt, _value, parser)`

I manage options -R, -1 and -2

`ktbs.standalone.make_server_class(ktbs_config)`

We define this closure so that `MyWSGIServer` class can access the configuration options.

class `ktbs.standalone.NoCache`

A strawman cache doing no real caching, used for debugging.

static get (`_key`)

Always return `None`

class `ktbs.standalone.FlashAllower(app, *domains)`

I wrap a WSGI application in order to make it accessible to Flash applets.

This is done by serving `/crossdomain.xml` .

KTBS Time library

I provide time management for KTBS.

I provide a mechanism for registering time units.

`ktbs.time.register_unit(uri, unit2timedelta, timedelta2unit)`

I register converter functions for a unit URI.

TODO DOC document signature and semantics of converter functions

`ktbs.time.get_converter_from_unit(uri)`

I return a converter function from the given unit to `timedelta`.

`ktbs.time.get_converter_to_unit(uri)`

I return a converter function from `timedelta` to the given unit.

`ktbs.time.ms2timedelta` (*a_int*)
I convert from milliseconds to timedelta

`ktbs.time.timedelta2ms` (*a_timedelta*)
I convert from timedelta to milliseconds

`ktbs.time.sec2timedelta` (*a_int*)
I convert from seconds to timedelta

`ktbs.time.timedelta2sec` (*a_timedelta*)
I convert from timedelta to seconds

`ktbs.time.lit2datetime` (*literal*)
Convert *literal* to datetime if possible, else return None.
If *literal* is None, also return None.

KTBS utilities

I provide utility functions for pythonic interfaces.

`ktbs.utils.extend_api` (*cls*)
I extend *cls* according to the design rationale of the kTBS abstract API.
More precisely: * for every `get_x(...)` method I add a 'x' property (if it accepts 0 args) * for every `set_x(...)` method I add a setter to the 'x' property * for every `iter_xs(...)` method I add a `list_xs(...)` method * for every `iter_xs(...)` method I add a 'xs' property (if it accepts 0 args)

`ktbs.utils.extend_api_ignore` (*func*)
I decorate functions that must be ignored by `extend_api()`.

`ktbs.utils.mint_uri_from_label` (*label*, *target*, *uri=None*, *suffix=''*)
Mint a URI for a resource posted to *target* based on *label*.

Parameters

- **label** – the label for the resource to create
- **target** – the resource “containing” the resource to create
- **uri** – if provided, will be used instead (must be fresh)
- **suffix** – if provided, will be added to the end of the URI

Returns a URI not present in *target.state*

Return type rdflib.URIRef

Raise InvalidDataError if *uri* is provided and not acceptable

`ktbs.utils.short_name` (*uri*)
Return the last part of the URI (fragment or path element).

CHAPTER 6

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

Note: This documentation is kept as reStructuredText documents, managed with [Sphinx](#).

k

- ktbs, 73
- ktbs.api, 73
 - ktbs.api.base, 76
 - ktbs.api.builtin_method, 94
 - ktbs.api.ktbs_root, 75
 - ktbs.api.method, 93
 - ktbs.api.obsel, 90
 - ktbs.api.resource, 74
 - ktbs.api.trace, 84
 - ktbs.api.trace_model, 80
 - ktbs.api.trace_obsels, 95
- ktbs.client, 73
- ktbs.engine, 96
 - ktbs.engine.base, 98
 - ktbs.engine.builtin_method, 103
 - ktbs.engine.ktbs_root, 97
 - ktbs.engine.method, 102
 - ktbs.engine.obsel, 101
 - ktbs.engine.resource, 97
 - ktbs.engine.service, 96
 - ktbs.engine.trace, 99
 - ktbs.engine.trace_model, 99
 - ktbs.engine.trace_obsels, 103
- ktbs.methods, 106
 - ktbs.methods.filter, 106
 - ktbs.methods.fsa, 106
 - ktbs.methods.fusion, 106
 - ktbs.methods.interface, 106
 - ktbs.methods.sparql, 107
 - ktbs.methods.utils, 107
- ktbs.namespace, 107
- ktbs.plugins, 107
 - ktbs.plugins.meth_external, 107
- ktbs.standalone, 108
- ktbs.time, 108
- ktbs.utils, 109

A

- AbstractTrace (class in ktbs.engine.trace), 99
- AbstractTraceMixin (class in ktbs.api.trace), 84
- AbstractTraceObsels (class in ktbs.engine.trace_obsels), 103
- AbstractTraceObselsMixin (class in ktbs.api.trace_obsels), 95
- ack_delete() (ktbs.engine.base.Base method), 98
- ack_delete() (ktbs.engine.base.InBase method), 99
- ack_delete() (ktbs.engine.method.Method method), 102
- ack_delete() (ktbs.engine.trace.AbstractTrace method), 100
- ack_delete() (ktbs.engine.trace.ComputedTrace method), 101
- ack_edit() (ktbs.engine.method.Method method), 102
- ack_edit() (ktbs.engine.trace.AbstractTrace method), 100
- ack_edit() (ktbs.engine.trace.ComputedTrace method), 101
- ack_edit() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 105
- ack_post() (ktbs.engine.base.Base method), 98
- ack_post() (ktbs.engine.ktbs_root.KtbsRoot method), 98
- add_data_type() (built-in function), 16
- add_data_type() (ktbs.api.trace_model.AttributeTypeMixin method), 82
- add_destination() (built-in function), 17
- add_destination() (ktbs.api.trace_model.RelationTypeMixin method), 83
- add_obsel_graph() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 104
- add_obsel_type() (built-in function), 16
- add_obsel_type() (ktbs.api.trace_model.AttributeTypeMixin method), 82
- add_origin() (built-in function), 17
- add_origin() (ktbs.api.trace_model.RelationTypeMixin method), 83
- add_parent() (built-in function), 14
- add_parent() (ktbs.api.trace_model.TraceModelMixin method), 80
- add_related_obsel() (built-in function), 17
- add_source_trace() (ktbs.api.trace.AbstractTraceMixin method), 85
- add_source_trace() (ktbs.api.trace.ComputedTraceMixin method), 89
- add_supertype() (built-in function), 15, 16
- add_supertype() (ktbs.api.trace_model.TraceModelMixin method), 81
- all_destinations (ktbs.api.trace_model.RelationTypeMixin attribute), 83
- all_origins (ktbs.api.trace_model.RelationTypeMixin attribute), 83
- attribute_types (ktbs.api.obsel.ObselMixin attribute), 91
- attribute_types (ktbs.api.trace_model.ObselTypeMixin attribute), 83
- attribute_types (ktbs.api.trace_model.TraceModelMixin attribute), 81
- AttributeTypeMixin (class in ktbs.api.trace_model), 82

B

- Base (class in ktbs.engine.base), 98
- base (ktbs.api.base.InBaseMixin attribute), 79
- base (ktbs.api.builtin_method.MethodBuiltinMixin attribute), 95
- BaseMixin (class in ktbs.api.base), 76
- bases (ktbs.api.base.BaseMixin attribute), 79
- bases (ktbs.api.ktbs_root.KtbsRootMixin attribute), 76
- begin (ktbs.api.obsel.ObselMixin attribute), 91
- begin_dt (ktbs.api.obsel.ObselMixin attribute), 91
- build_cmdline_options() (in module ktbs.standalone), 108
- build_select() (ktbs.api.trace_obsels.AbstractTraceObselsMixin method), 95
- builtin_methods (ktbs.api.ktbs_root.KtbsRootMixin attribute), 76

C

- check_deletable() (ktbs.engine.base.Base method), 98
- check_deletable() (ktbs.engine.method.Method method), 102

- check_deletable() (ktbs.engine.trace.AbstractTrace method), 100
 - check_new() (ktbs.engine.trace.StoredTrace method), 100
 - check_new_graph() (ktbs.engine.method.Method class method), 102
 - check_new_graph() (ktbs.engine.trace.AbstractTrace class method), 100
 - check_new_graph() (ktbs.engine.trace.StoredTrace class method), 100
 - check_parameters() (ktbs.engine.base.Base method), 98
 - check_parameters() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 105
 - check_posted_graph() (ktbs.engine.resource.KtbsPostableMixin method), 97
 - children (ktbs.api.method.MethodMixin attribute), 94
 - complete_new_graph() (ktbs.engine.method.Method class method), 102
 - complete_new_graph() (ktbs.engine.trace.StoredTrace class method), 100
 - complete_new_graph() (ktbs.engine.trace_model.TraceModel class method), 99
 - compute_obsels() (ktbs.methods.interface.IMethod method), 106
 - compute_trace_description() (ktbs.methods.interface.IMethod method), 106
 - ComputedTrace (class in ktbs.engine.trace), 101
 - ComputedTraceMixin (class in ktbs.api.trace), 88
 - ComputedTraceObsels (class in ktbs.engine.trace_obsels), 105
 - context_uris (ktbs.api.trace.AbstractTraceMixin attribute), 86
 - contexts (ktbs.api.trace.AbstractTraceMixin attribute), 86
 - copy_obsel() (in module ktbs.methods.utils), 107
 - create() (ktbs.engine.method.Method class method), 102
 - create() (ktbs.engine.trace.AbstractTrace class method), 100
 - create() (ktbs.engine.trace.ComputedTrace class method), 101
 - create_attribute_type() (built-in function), 14, 16
 - create_attribute_type() (ktbs.api.trace_model.ObselTypeMixin method), 83
 - create_attribute_type() (ktbs.api.trace_model.TraceModelMixin method), 81
 - create_base() (built-in function), 10, 11
 - create_base() (ktbs.api.base.BaseMixin method), 77
 - create_base() (ktbs.api.ktbs_root.KtbsRootMixin method), 76
 - create_computed_trace() (built-in function), 11
 - create_computed_trace() (ktbs.api.base.BaseMixin method), 78
 - create_data_graph() (built-in function), 11
 - create_data_graph() (ktbs.api.base.BaseMixin method), 78
 - create_lock() (ktbs.engine.base.Base class method), 98
 - create_method() (built-in function), 11
 - create_method() (ktbs.api.base.BaseMixin method), 78
 - create_model() (built-in function), 11
 - create_model() (ktbs.api.base.BaseMixin method), 77
 - create_obsel() (built-in function), 12
 - create_obsel() (ktbs.api.trace.StoredTraceMixin method), 88
 - create_obsel_type() (built-in function), 14
 - create_obsel_type() (ktbs.api.trace_model.TraceModelMixin method), 80
 - create_relation_type() (built-in function), 14, 16
 - create_relation_type() (ktbs.api.trace_model.ObselTypeMixin method), 83
 - create_relation_type() (ktbs.api.trace_model.TraceModelMixin method), 81
 - create_stored_trace() (built-in function), 11
 - create_stored_trace() (ktbs.api.base.BaseMixin method), 78
- ## D
- data_graphs (ktbs.api.base.BaseMixin attribute), 79
 - data_types (ktbs.api.trace_model.AttributeTypeMixin attribute), 82
 - default_subject (ktbs.api.trace.StoredTraceMixin attribute), 88
 - del_attribute_value() (built-in function), 17
 - del_parameter() (built-in function), 13, 15
 - del_parameter() (ktbs.api.builtin_method.MethodBuiltinMixin method), 94
 - del_parameter() (ktbs.api.method.WithParametersMixin method), 93
 - del_parameter() (ktbs.api.trace.ComputedTraceMixin method), 89
 - del_related_obsel() (built-in function), 17
 - del_source_trace() (ktbs.api.trace.AbstractTraceMixin method), 85
 - del_source_trace() (ktbs.api.trace.ComputedTraceMixin method), 89
 - delete() (ktbs.api.obsel.ObselProxy method), 92
 - delete() (ktbs.api.trace.OppportunisticObselCollection method), 90
 - delete() (ktbs.engine.base.InBase method), 99
 - delete() (ktbs.engine.ktbs_root.KtbsRoot method), 97
 - delete() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 105
 - delete() (ktbs.engine.trace_obsels.ComputedTraceObsels method), 105
 - depth (ktbs.api.base.BaseMixin attribute), 79
 - destinations (ktbs.api.trace_model.RelationTypeMixin attribute), 84
 - diagnosis (ktbs.api.trace.ComputedTraceMixin attribute), 89

E

edit() (ktbs.api.obsel.ObselProxy method), 92
 edit() (ktbs.api.trace.OpportunisticObselCollection method), 90
 edit() (ktbs.engine.base.InBase method), 99
 edit() (ktbs.engine.trace_obsels.ComputedTraceObsels method), 105
 end (ktbs.api.obsel.ObselMixin attribute), 91
 end_dt (ktbs.api.obsel.ObselMixin attribute), 91
 etag (ktbs.engine.trace_obsels.AbstractTraceObsels attribute), 103
 extend_api() (in module ktbs.utils), 109
 extend_api_ignore() (in module ktbs.utils), 109

F

factory() (ktbs.api.obsel.ObselProxy method), 92
 factory() (ktbs.api.trace.OpportunisticObselCollection method), 89
 find_created() (ktbs.engine.base.Base method), 98
 find_created() (ktbs.engine.ktbs_root.KtbsRoot method), 98
 FlashAllower (class in ktbs.standalone), 108
 force_state_refresh() (built-in function), 9
 force_state_refresh() (ktbs.api.obsel.ObselProxy method), 92
 force_state_refresh() (ktbs.api.trace.OpportunisticObselCollection method), 90
 force_state_refresh() (ktbs.engine.trace.ComputedTraceObsels method), 101
 force_state_refresh() (ktbs.engine.trace_obsels.ComputedTraceObsels method), 105

G

get() (built-in function), 10, 13
 get() (ktbs.api.base.BaseMixin method), 77
 get() (ktbs.api.trace_model.TraceModelMixin method), 80
 get() (ktbs.engine.service.KtbsService method), 96
 get() (ktbs.standalone.NoCache static method), 108
 get_attribute_value() (built-in function), 17
 get_attribute_value() (ktbs.api.obsel.ObselMixin method), 91
 get_base() (built-in function), 10, 11, 13, 14
 get_base() (ktbs.api.base.InBaseMixin method), 79
 get_base() (ktbs.api.builtin_method.MethodBuiltinMixin method), 94
 get_base() (ktbs.api.ktbs_root.KtbsRootMixin method), 76
 get_begin() (built-in function), 17
 get_begin() (ktbs.api.obsel.ObselMixin method), 90
 get_begin_dt() (ktbs.api.obsel.ObselMixin method), 90
 get_builtin_method() (built-in function), 10
 get_builtin_method() (ktbs.api.ktbs_root.KtbsRootMixin method), 76

get_builtin_method_impl() (in module ktbs.engine.builtin_method), 103
 get_converter_from_unit() (in module ktbs.time), 108
 get_converter_to_unit() (in module ktbs.time), 108
 get_created_class() (ktbs.engine.trace.StoredTrace method), 101
 get_default_subject() (built-in function), 12
 get_default_subject() (ktbs.api.trace.StoredTraceMixin method), 87
 get_depth() (ktbs.api.base.BaseMixin method), 77
 get_diagnosis() (ktbs.api.trace.ComputedTraceMixin method), 89
 get_end() (built-in function), 17
 get_end() (ktbs.api.obsel.ObselMixin method), 90
 get_end_dt() (ktbs.api.obsel.ObselMixin method), 90
 get_etag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 103
 get_id() (built-in function), 9
 get_id() (ktbs.api.resource.KtbsResourceMixin method), 74
 get_ktbs() (in module ktbs.client), 73
 get_label() (built-in function), 10
 get_label() (ktbs.api.resource.KtbsResourceMixin method), 74
 get_log_mon_tag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 104
 get_method() (built-in function), 13
 get_method() (ktbs.api.trace.ComputedTraceMixin method), 89
 get_method_uri() (ktbs.api.trace.ComputedTraceMixin method), 89
 get_model() (built-in function), 11, 15, 16
 get_model() (ktbs.api.trace.AbstractTraceMixin method), 84
 get_model_prefix() (ktbs.api.trace.AbstractTraceMixin method), 85
 get_model_uri() (ktbs.api.trace.AbstractTraceMixin method), 85
 get_obsel() (built-in function), 12
 get_obsel() (ktbs.api.trace.AbstractTraceMixin method), 84
 get_obsel_bounded_description() (in module ktbs.engine.obsel), 101
 get_obsel_type() (built-in function), 17
 get_obsel_type() (ktbs.api.obsel.ObselMixin method), 90
 get_origin() (built-in function), 11
 get_origin() (ktbs.api.trace.AbstractTraceMixin method), 84
 get_parameter() (built-in function), 13, 14
 get_parameter() (ktbs.api.builtin_method.MethodBuiltinMixin method), 94
 get_parameter() (ktbs.api.method.WithParametersMixin method), 93
 get_parent() (built-in function), 14

- get_parent() (ktbs.api.base.BaseMixin method), 77
- get_parent() (ktbs.api.builtin_method.MethodBuiltinMixin method), 94
- get_parent() (ktbs.api.method.MethodMixin method), 93
- get_parent_uri() (ktbs.api.method.MethodMixin method), 93
- get_pse_mon_tag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 104
- get_pseudomon_range() (ktbs.api.trace.AbstractTraceMixin method), 86
- get_readonly() (built-in function), 9
- get_readonly() (ktbs.api.resource.KtbsResourceMixin method), 75
- get_state() (ktbs.api.obsel.ObselProxy method), 92
- get_state() (ktbs.api.trace.OpportunisticObselCollection method), 89
- get_state() (ktbs.engine.base.Base method), 98
- get_state() (ktbs.engine.trace.ComputedTrace method), 101
- get_state() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 105
- get_state() (ktbs.engine.trace_obsels.ComputedTraceObsels method), 105
- get_str_mon_tag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 103
- get_subject() (ktbs.api.obsel.ObselMixin method), 90
- get_sync_status() (ktbs.api.resource.KtbsResourceMixin method), 75
- get_trace() (built-in function), 17
- get_trace() (ktbs.api.obsel.ObselMixin method), 90
- get_trace_begin() (built-in function), 11
- get_trace_begin() (ktbs.api.trace.AbstractTraceMixin method), 86
- get_trace_begin_dt() (built-in function), 11
- get_trace_begin_dt() (ktbs.api.trace.AbstractTraceMixin method), 86
- get_trace_end() (built-in function), 11
- get_trace_end() (ktbs.api.trace.AbstractTraceMixin method), 86
- get_trace_end_dt() (built-in function), 12
- get_trace_end_dt() (ktbs.api.trace.AbstractTraceMixin method), 86
- get_unit() (built-in function), 13
- get_unit() (ktbs.api.trace_model.TraceModelMixin method), 80
- get_uri() (built-in function), 9
- get_uri() (ktbs.api.resource.KtbsResourceMixin method), 74
- I**
- id (ktbs.api.resource.KtbsResourceMixin attribute), 75
- IMethod (class in ktbs.methods.interface), 106
- InBase (class in ktbs.engine.base), 99
- InBaseMixin (class in ktbs.api.base), 79
- init_graph() (ktbs.engine.trace_obsels.AbstractTraceObsels class method), 103
- init_ktbs() (ktbs.engine.service.KtbsService class method), 96
- inverse_relation_types (ktbs.api.obsel.ObselMixin attribute), 91
- inverse_relation_types (ktbs.api.trace_model.ObselTypeMixin attribute), 83
- iter_all_destinations() (ktbs.api.trace_model.RelationTypeMixin method), 83
- iter_all_origins() (ktbs.api.trace_model.RelationTypeMixin method), 83
- iter_attribute_types() (ktbs.api.obsel.ObselMixin method), 91
- iter_attribute_types() (ktbs.api.trace_model.ObselTypeMixin method), 82
- iter_attribute_types() (ktbs.api.trace_model.TraceModelMixin method), 80
- iter_bases() (ktbs.api.base.BaseMixin method), 77
- iter_bases() (ktbs.api.ktbs_root.KtbsRootMixin method), 76
- iter_builtin_method_impl() (in module ktbs.engine.builtin_method), 103
- iter_builtin_methods() (ktbs.api.ktbs_root.KtbsRootMixin method), 76
- iter_children() (ktbs.api.method.MethodMixin method), 94
- iter_context_uris() (ktbs.api.trace.AbstractTraceMixin method), 86
- iter_contexts() (ktbs.api.trace.AbstractTraceMixin method), 85
- iter_data_graphs() (ktbs.api.base.BaseMixin method), 77
- iter_data_types() (ktbs.api.trace_model.AttributeTypeMixin method), 82
- iter_destinations() (ktbs.api.trace_model.RelationTypeMixin method), 83
- iter_etags() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 105
- iter_inverse_relation_types() (ktbs.api.obsel.ObselMixin method), 91
- iter_inverse_relation_types() (ktbs.api.trace_model.ObselTypeMixin method), 82
- iter_methods() (ktbs.api.base.BaseMixin method), 77
- iter_models() (ktbs.api.base.BaseMixin method), 77
- iter_obsel_types() (ktbs.api.trace_model.AttributeTypeMixin method), 82
- iter_obsel_types() (ktbs.api.trace_model.TraceModelMixin method), 80
- iter_obsels() (ktbs.api.trace.AbstractTraceMixin method), 84
- iter_origins() (ktbs.api.trace_model.RelationTypeMixin method), 83
- iter_parameters() (ktbs.api.method.WithParametersMixin

- method), 93
 - iter_parameters_with_values() (ktbs.api.method.WithParametersMixin method), 93
 - iter_parents() (ktbs.api.trace_model.TraceModelMixin method), 80
 - iter_related_obsels() (ktbs.api.obsel.ObselMixin method), 91
 - iter_relating_obsels() (ktbs.api.obsel.ObselMixin method), 91
 - iter_relation_types() (ktbs.api.obsel.ObselMixin method), 91
 - iter_relation_types() (ktbs.api.trace_model.ObselTypeMixin method), 82
 - iter_relation_types() (ktbs.api.trace_model.TraceModelMixin method), 80
 - iter_source_obsels() (ktbs.api.obsel.ObselMixin method), 91
 - iter_source_traces() (ktbs.api.trace.AbstractTraceMixin method), 85
 - iter_traces() (ktbs.api.base.BaseMixin method), 77
 - iter_transformed_traces() (ktbs.api.trace.AbstractTraceMixin method), 85
 - iter_used_by() (ktbs.api.method.MethodMixin method), 94
- ## K
- ktbs (module), 73
 - ktbs.api (module), 73
 - ktbs.api.base (module), 76
 - ktbs.api.builtin_method (module), 94
 - ktbs.api.ktbs_root (module), 75
 - ktbs.api.method (module), 93
 - ktbs.api.obsel (module), 90
 - ktbs.api.resource (module), 74
 - ktbs.api.trace (module), 84
 - ktbs.api.trace_model (module), 80
 - ktbs.api.trace_obsels (module), 95
 - ktbs.client (module), 73
 - ktbs.engine (module), 96
 - ktbs.engine.base (module), 98
 - ktbs.engine.builtin_method (module), 103
 - ktbs.engine.ktbs_root (module), 97
 - ktbs.engine.method (module), 102
 - ktbs.engine.obsel (module), 101
 - ktbs.engine.resource (module), 97
 - ktbs.engine.service (module), 96
 - ktbs.engine.trace (module), 99
 - ktbs.engine.trace_model (module), 99
 - ktbs.engine.trace_obsels (module), 103
 - ktbs.methods (module), 106
 - ktbs.methods.filter (module), 106
 - ktbs.methods.fsa (module), 106
 - ktbs.methods.fusion (module), 106
 - ktbs.methods.interface (module), 106
 - ktbs.methods.sparql (module), 107
 - ktbs.methods.utils (module), 107
 - ktbs.namespace (module), 107
 - ktbs.plugins (module), 107
 - ktbs.plugins.meth_external (module), 107
 - ktbs.standalone (module), 108
 - ktbs.time (module), 108
 - ktbs.utils (module), 109
 - KtbsPostableMixin (class in ktbs.engine.resource), 97
 - KtbsResource (class in ktbs.engine.resource), 97
 - KtbsResourceMixin (class in ktbs.api.resource), 74
 - KtbsRoot (class in ktbs.engine.ktbs_root), 97
 - KtbsRootMixin (class in ktbs.api.ktbs_root), 75
 - KtbsService (class in ktbs.engine.service), 96
- ## L
- label (ktbs.api.resource.KtbsResourceMixin attribute), 75
 - list_all_destinations() (ktbs.api.trace_model.RelationTypeMixin method), 84
 - list_all_origins() (ktbs.api.trace_model.RelationTypeMixin method), 84
 - list_attribute_types() (built-in function), 13, 15, 17
 - list_attribute_types() (ktbs.api.obsel.ObselMixin method), 91
 - list_attribute_types() (ktbs.api.trace_model.ObselTypeMixin method), 83
 - list_attribute_types() (ktbs.api.trace_model.TraceModelMixin method), 81
 - list_bases() (built-in function), 10, 11
 - list_bases() (ktbs.api.base.BaseMixin method), 79
 - list_bases() (ktbs.api.ktbs_root.KtbsRootMixin method), 76
 - list_builtin_methods() (built-in function), 10
 - list_builtin_methods() (ktbs.api.ktbs_root.KtbsRootMixin method), 76
 - list_children() (ktbs.api.method.MethodMixin method), 94
 - list_context_uris() (ktbs.api.trace.AbstractTraceMixin method), 86
 - list_contexts() (built-in function), 12
 - list_contexts() (ktbs.api.trace.AbstractTraceMixin method), 86
 - list_data_graphs() (built-in function), 11
 - list_data_graphs() (ktbs.api.base.BaseMixin method), 79
 - list_data_types() (built-in function), 16
 - list_data_types() (ktbs.api.trace_model.AttributeTypeMixin method), 82
 - list_destinations() (built-in function), 17
 - list_destinations() (ktbs.api.trace_model.RelationTypeMixin method), 84
 - list_inverse_relation_types() (built-in function), 15, 17

- list_inverse_relation_types() (ktbs.api.obsel.ObselMixin method), 91
 - list_inverse_relation_types() (ktbs.api.trace_model.ObselTypeMixin method), 83
 - list_methods() (built-in function), 10
 - list_methods() (ktbs.api.base.BaseMixin method), 79
 - list_models() (built-in function), 10
 - list_models() (ktbs.api.base.BaseMixin method), 79
 - list_obsel_types() (built-in function), 14, 16
 - list_obsel_types() (ktbs.api.trace_model.AttributeTypeMixin method), 82
 - list_obsel_types() (ktbs.api.trace_model.TraceModelMixin method), 81
 - list_obsels() (built-in function), 12
 - list_obsels() (ktbs.api.trace.AbstractTraceMixin method), 86
 - list_origins() (built-in function), 16
 - list_origins() (ktbs.api.trace_model.RelationTypeMixin method), 84
 - list_parameters() (built-in function), 13, 14
 - list_parameters() (ktbs.api.builtin_method.MethodBuiltinMixin method), 94
 - list_parameters() (ktbs.api.method.WithParametersMixin method), 93
 - list_parameters_with_values() (ktbs.api.method.WithParametersMixin method), 93
 - list_parents() (built-in function), 13
 - list_parents() (ktbs.api.trace_model.TraceModelMixin method), 81
 - list_related_obsels() (built-in function), 17
 - list_related_obsels() (ktbs.api.obsel.ObselMixin method), 91
 - list_relating_obsels() (ktbs.api.obsel.ObselMixin method), 91
 - list_relation_types() (built-in function), 13, 15, 17
 - list_relation_types() (ktbs.api.obsel.ObselMixin method), 91
 - list_relation_types() (ktbs.api.trace_model.ObselTypeMixin method), 83
 - list_relation_types() (ktbs.api.trace_model.TraceModelMixin method), 82
 - list_source_obsels() (built-in function), 17
 - list_source_obsels() (ktbs.api.obsel.ObselMixin method), 91
 - list_source_traces() (built-in function), 12
 - list_source_traces() (ktbs.api.trace.AbstractTraceMixin method), 86
 - list_subtypes() (built-in function), 15, 16
 - list_supertypes() (built-in function), 15, 16
 - list_traces() (built-in function), 10
 - list_traces() (ktbs.api.base.BaseMixin method), 79
 - list_transformed_traces() (built-in function), 12
 - list_transformed_traces() (ktbs.api.trace.AbstractTraceMixin method), 86
 - list_used_by() (ktbs.api.method.MethodMixin method), 94
 - lit2datetime() (in module ktbs.time), 109
 - log_mon_tag (ktbs.engine.trace_obsels.AbstractTraceObsels attribute), 104
- ## M
- main() (in module ktbs.standalone), 108
 - make_ktbs() (in module ktbs.engine.service), 96
 - make_server_class() (in module ktbs.standalone), 108
 - match_obseltype() (in module ktbs.methods.fsa), 106
 - match_sparql_ask() (in module ktbs.methods.fsa), 106
 - Method (class in ktbs.engine.method), 102
 - method (ktbs.api.trace.ComputedTraceMixin attribute), 89
 - method_uri (ktbs.api.trace.ComputedTraceMixin attribute), 89
 - MethodBuiltinMixin (class in ktbs.api.builtin_method), 94
 - MethodMixin (class in ktbs.api.method), 93
 - methods (ktbs.api.base.BaseMixin attribute), 79
 - mint_uri() (ktbs.engine.resource.KtbsResource class method), 97
 - mint_uri_from_label() (in module ktbs.utils), 109
 - model (ktbs.api.trace.AbstractTraceMixin attribute), 86
 - model (ktbs.api.trace.StoredTraceMixin attribute), 88
 - model_prefix (ktbs.api.trace.AbstractTraceMixin attribute), 86
 - model_uri (ktbs.api.trace.AbstractTraceMixin attribute), 86
 - models (ktbs.api.base.BaseMixin attribute), 79
 - ms2timedelta() (in module ktbs.time), 108
- ## N
- NoCache (class in ktbs.standalone), 108
 - number_callback() (in module ktbs.standalone), 108
- ## O
- Obsel (class in ktbs.engine.obsel), 101
 - obsel_collection (ktbs.api.trace.AbstractTraceMixin attribute), 85
 - obsel_collection (ktbs.engine.trace.AbstractTrace attribute), 100
 - obsel_type (ktbs.api.obsel.ObselMixin attribute), 91
 - obsel_types (ktbs.api.trace_model.AttributeTypeMixin attribute), 82
 - obsel_types (ktbs.api.trace_model.TraceModelMixin attribute), 82
 - ObselMixin (class in ktbs.api.obsel), 90
 - ObselProxy (class in ktbs.api.obsel), 92
 - obsels (ktbs.api.trace.AbstractTraceMixin attribute), 87

ObselTypeMixin (class in ktbs.api.trace_model), 82
 OpportunisticObselCollection (class in ktbs.api.trace), 89
 origin (ktbs.api.trace.AbstractTraceMixin attribute), 87
 origin (ktbs.api.trace.StoredTraceMixin attribute), 88
 origins (ktbs.api.trace_model.RelationTypeMixin attribute), 84

P

parameters (ktbs.api.method.WithParametersMixin attribute), 93
 parameters_as_dict (ktbs.api.method.WithParametersMixin attribute), 93
 parameters_with_values (ktbs.api.method.WithParametersMixin attribute), 93
 parent (ktbs.api.base.BaseMixin attribute), 79
 parent (ktbs.api.builtin_method.MethodBuiltinMixin attribute), 95
 parent (ktbs.api.method.MethodMixin attribute), 94
 parent_uri (ktbs.api.method.MethodMixin attribute), 94
 parents (ktbs.api.trace_model.TraceModelMixin attribute), 82
 parse_configuration_options() (in module ktbs.standalone), 108
 parse_options() (in module ktbs.standalone), 108
 post_graph() (ktbs.api.obsel.ObselProxy method), 92
 post_graph() (ktbs.api.trace.OpportunisticObselCollection method), 90
 post_graph() (ktbs.engine.trace.StoredTrace method), 100
 prepare_edit() (ktbs.engine.method.Method method), 102
 prepare_edit() (ktbs.engine.trace.AbstractTrace method), 100
 prepare_edit() (ktbs.engine.trace.ComputedTrace method), 101
 prepare_edit() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 105
 pse_mon_tag (ktbs.engine.trace_obsels.AbstractTraceObsels attribute), 104
 pseudomon_range (ktbs.api.trace.AbstractTraceMixin attribute), 87

R

readonly (ktbs.api.resource.KtbsResourceMixin attribute), 75
 register_builtin_method_impl() (in module ktbs.engine.builtin_method), 103
 register_unit() (in module ktbs.time), 108
 relation_types (ktbs.api.obsel.ObselMixin attribute), 91
 relation_types (ktbs.api.trace_model.ObselTypeMixin attribute), 83
 relation_types (ktbs.api.trace_model.TraceModelMixin attribute), 82
 RelationTypeMixin (class in ktbs.api.trace_model), 83
 remove() (built-in function), 10
 remove() (ktbs.api.base.BaseMixin method), 79

remove() (ktbs.api.base.InBaseMixin method), 79
 remove() (ktbs.api.resource.KtbsResourceMixin method), 75
 remove_data_type() (built-in function), 16
 remove_data_type() (ktbs.api.trace_model.AttributeTypeMixin method), 82
 remove_destination() (built-in function), 17
 remove_destination() (ktbs.api.trace_model.RelationTypeMixin method), 83
 remove_obsel_type() (built-in function), 16
 remove_obsel_type() (ktbs.api.trace_model.AttributeTypeMixin method), 82
 remove_origin() (built-in function), 17
 remove_origin() (ktbs.api.trace_model.RelationTypeMixin method), 83
 remove_parent() (built-in function), 14
 remove_parent() (ktbs.api.trace_model.TraceModelMixin method), 80
 remove_supertype() (built-in function), 15, 16
 remove_supertype() (ktbs.api.trace_model.TraceModelMixin method), 81
 replace_obsels() (in module ktbs.methods.utils), 107
 reset_label() (built-in function), 10
 reset_label() (ktbs.api.resource.KtbsResourceMixin method), 75

RFC

RFC 5988, 60

S

sec2timedelta() (in module ktbs.time), 109
 set_attribute_value() (built-in function), 17
 set_default_subject() (built-in function), 12
 set_default_subject() (ktbs.api.trace.StoredTraceMixin method), 87
 set_etag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 103
 set_label() (built-in function), 10
 set_label() (ktbs.api.resource.KtbsResourceMixin method), 75
 set_log_mon_tag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 104
 set_method() (built-in function), 13
 set_method() (ktbs.api.trace.ComputedTraceMixin method), 89
 set_model() (built-in function), 12
 set_model() (ktbs.api.trace.StoredTraceMixin method), 87
 set_origin() (built-in function), 12
 set_origin() (ktbs.api.trace.StoredTraceMixin method), 87
 set_parameter() (built-in function), 13, 15
 set_parameter() (ktbs.api.method.WithParametersMixin method), 93

set_parameter() (ktbs.api.trace.ComputedTraceMixin method), 89
 set_paremeter() (ktbs.api.builtin_method.MethodBuiltinMixin method), 94
 set_parent() (built-in function), 14
 set_parent() (ktbs.api.builtin_method.MethodBuiltinMixin method), 94
 set_parent() (ktbs.api.method.MethodMixin method), 93
 set_pse_mon_tag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 104
 set_pseudomon_range() (ktbs.api.trace.AbstractTraceMixin method), 86
 set_str_mon_tag() (ktbs.engine.trace_obsels.AbstractTraceObsels method), 104
 set_trace_begin() (built-in function), 12
 set_trace_begin() (ktbs.api.trace.StoredTraceMixin method), 87
 set_trace_begin_dt() (built-in function), 12
 set_trace_begin_dt() (ktbs.api.trace.StoredTraceMixin method), 87
 set_trace_end() (built-in function), 12
 set_trace_end() (ktbs.api.trace.StoredTraceMixin method), 87
 set_trace_end_dt() (built-in function), 12
 set_trace_end_dt() (ktbs.api.trace.StoredTraceMixin method), 88
 set_unit() (built-in function), 13
 set_unit() (ktbs.api.trace_model.TraceModelMixin method), 80
 short_name() (in module ktbs.utils), 109
 source_obsels (ktbs.api.obsel.ObselMixin attribute), 92
 source_traces (ktbs.api.trace.AbstractTraceMixin attribute), 87
 start_plugin() (in module ktbs.plugins.meth_external), 107
 state (ktbs.api.resource.KtbsResourceMixin attribute), 75
 state (ktbs.api.trace.OpportunisticObselCollection attribute), 90
 StoredTrace (class in ktbs.engine.trace), 100
 StoredTraceMixin (class in ktbs.api.trace), 87
 StoredTraceObsels (class in ktbs.engine.trace_obsels), 105
 str_mon_tag (ktbs.engine.trace_obsels.AbstractTraceObsels attribute), 104
 subject (ktbs.api.obsel.ObselMixin attribute), 92
 sync_status (ktbs.api.resource.KtbsResourceMixin attribute), 75

T

timedelta2ms() (in module ktbs.time), 109
 timedelta2sec() (in module ktbs.time), 109
 trace (ktbs.api.obsel.ObselMixin attribute), 92
 trace (ktbs.api.trace_obsels.AbstractTraceObselsMixin attribute), 95

trace_begin (ktbs.api.trace.AbstractTraceMixin attribute), 87
 trace_begin (ktbs.api.trace.StoredTraceMixin attribute), 88
 trace_begin_dt (ktbs.api.trace.AbstractTraceMixin attribute), 87
 trace_begin_dt (ktbs.api.trace.StoredTraceMixin attribute), 88
 trace_end (ktbs.api.trace.StoredTraceMixin attribute), 88
 trace_end_dt (ktbs.api.trace.AbstractTraceMixin attribute), 87
 trace_end_dt (ktbs.api.trace.StoredTraceMixin attribute), 88
 trace_statistics (ktbs.api.trace.AbstractTraceMixin attribute), 85
 trace_uri (ktbs.api.trace_obsels.AbstractTraceObselsMixin attribute), 95
 TraceModel (class in ktbs.engine.trace_model), 99
 TraceModelMixin (class in ktbs.api.trace_model), 80
 traces (ktbs.api.base.BaseMixin attribute), 79
 transformed_traces (ktbs.api.trace.AbstractTraceMixin attribute), 87
 translate_node() (in module ktbs.methods.utils), 107

U

unit (ktbs.api.trace_model.TraceModelMixin attribute), 82
 unit (ktbs.engine.trace.AbstractTrace attribute), 100
 unregister_builtin_method_impl() (in module ktbs.engine.builtin_method), 103
 used_by (ktbs.api.method.MethodMixin attribute), 94

W

WithParametersMixin (class in ktbs.api.method), 93