The TREE hypermedia specification

Draft Community Group Report, 20 June 2025

▼ More details about this document

This version:
https://w3id.org/tree/specification

Feedback:
public-treecg@w3.org with subject line "`[TREE] … message topic …`" (archives)

Issue Tracking:
GitHub

Editor:
Pieter Colpaert

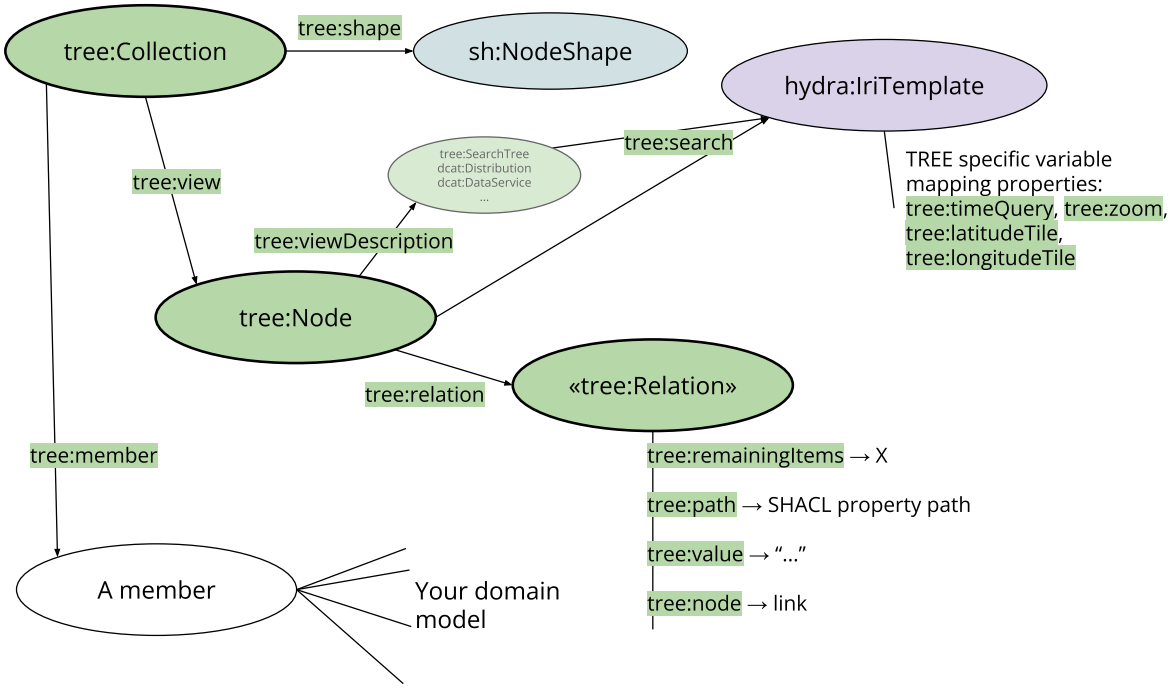
Abstract

The TREE specification provides instructions for clients to interpret and navigate Web APIs structured as search trees. It defines how members (sets of quads) in a dataset can be distributed across multiple pages interlinked through relationships. The specification introduces key concepts such as `tree:Collection` (a set of members), `tree:Node` (the pages in the search tree), and `tree:Relation` (links between nodes). By interpreting such qualified relations and search forms, TREE enables clients to efficiently retrieve their members of interest.

Status of this document

Table of Contents

§ 1. Overview



The TREE specification introduces these core concepts:

- `tree:Collection` is a set of members. It typically has these properties when described in a node:
  - `tree:member` points at the root focus node for each member from which to retrieve and extract all quads of that member
  - `tree:view` points to the `tree:Node` you are currently visiting, or points to one specific root node in the HTTP response after dereferencing the `tree:Collection` identifier.
  - `tree:shape` indicates the [SHACL] shape (exactly one) to which each member in the collection adheres
- `tree:Node` is a page in the search tree
  - `tree:relation` points at relations to other nodes

- `tree:search` describes a search form that allows an agent to jump from this node to a specific `tree:Node` in the (sub)tree
- `tree:viewDescription` points to an entity with a reusable piece of information relevant to the full search tree. Multiple descriptions MUST be combined.

- `tree:Relation` is a relation from one node to another. An extension of this class indicates a specific type of relation (e.g., a `tree:GreaterThanRelation`). A relation typically has these properties:
  - `tree:node` is the URL of the other node
  - `tree:path` indicates to which of the members' properties this relation applies
  - `tree:value` indicates a value constraint on the members' values
  - `tree:remainingItems` defines how many members can be reached when following this relation

A simple collection can be created as illustrated in the following example:

EXAMPLE 1 ¶

```
ex:Collection1 a tree:Collection;
            rdfs:label "A Collection of subjects"@en;
            tree:member ex:Subject1, ex:Subject2 .

ex:Subject1 a ex:Subject ;
            rdfs:label "Subject 1" ;
            ex:value 1 .

ex:Subject2 a ex:Subject ;
            rdfs:label "Subject 2" ;
            ex:value 2 .
```

From the moment this collection of members grows too large for one page, a pagination needs to be created in which an initial set of members can be found through the first `tree:Node`, and more members can be found by interpreting the TREE hypermedia controls. This is illustrated in the next example:

EXAMPLE 2

```
> HTTP GET https://example.org/Node1

ex:Collection1 a tree:Collection;
            tree:view ex:Node1 ;
            tree:member ex:Subject1, ex:Subject2 .

ex:Node1 a tree:Node ;
        tree:relation ex:R1,ex:R2, ex:R3 .

ex:R1 a tree:GreaterThanOrEqualToRelation ;
    tree:node ex:Node3 ; # This is the URL of another page
    tree:value 3;
    tree:path ex:value .

ex:R2 a tree:LessThanRelation ; # This is useful for a client that is looking for a valu
    tree:node ex:Node3 ;
    tree:value 10;
    tree:remainingItems 7 ;
    tree:path ex:value .

ex:R3 a tree:GreaterThanOrEqualToRelation ;
    tree:node ex:Node4 ;
    tree:value 10;
    tree:remainingItems 10 ;
    tree:path ex:value .

ex:Subject1 a ex:Subject ;
            rdfs:label "Subject 1" ;
            ex:value 1 .

ex:Subject2 a ex:Subject ;
            rdfs:label "Subject 2" ;
            ex:value 2 .
```

2. Definitions

A `tree:Collection` is a set of zero or more `tree:Member`s.

A `tree:Member` is a set of (at least one) quad(s) defined by the member extraction algorithm (see further).

A `tree:Node` is a dereferenceable resource containing `tree:Relation`s and a subset of (⊆) members of the collection. In a `tree:Node`, both the set of `tree:Relation`s as the subset of members MAY be empty. The same member MAY be contained in multiple nodes.

A `tree:Relation` is a function denoting a conditional link to another `tree:Node`.

A `tree:Node` is part of a search tree, and apart from the root node, it has exactly one other `tree:Node` of the search tree linking into it through one or more relations.

A `tree:search` form is an IRI template, that when filled out with the right parameters becomes a `tree:Node` IRI, or when dereferenced will redirect to a `tree:Node` from which all members in the collection that adhere to the described comparator can be found.

A search tree is the–in this document–implicit concept of a set of interlinked `tree:Node`s publishing a `tree:Collection`. It will adhere to a certain growth or tree balancing strategy. In one tree, completeness MUST be guaranteed, unless indicated otherwise (as is possible in LDES using a retention policy).

§ 3. Initialization

A TREE-based client MUST be initiated either by using the IRI of the `tree:Collection`, or by using a URL that is the IRI of the root `tree:Node` or will redirect to it. The client MUST start by dereferencing the IRI resulting in a set of [rdf-concepts] triples or quads:

1. Detecting and handling a `tree:Collection`: If exactly one `<collectionIRI> tree:view ?o` triple pattern can be matched, its object identifies the root `tree:Node`. If the current page IRI is not equal to that node's IRI, the client MUST dereference the IRI of the root node and restart the initialization process from there. This behavior ensures backward compatibility and aligns with legacy practices. More advanced discovery mechanisms (e.g., supporting multiple views or selection logic) are out of scope for this specification and will be addressed by the report on Discovery and Context Information.
2. Detecting and handling a root `tree:Node`: In this case, the final IRI after all HTTP redirects MUST correspond to the object of a `tree:view` triple linking it to a `tree:Collection`. This allows the client to verify that it has reached a valid entry point into the TREE structure.

Once the root `tree:Node` is retrieved and the `tree:Collection` is known, the client can proceed to extracting members, traversing the relations, or using search forms.

> NOTE: The report on Discovery and Context Information is currently a work in progress for which an on-going draft is available.

§ 4. The member extraction algorithm

The member extraction algorithm allows a data publisher to define their members in different ways:

1. As in the examples above: all quads with the object of the `tree:member` quads as a subject (and recursively the quads of their blank nodes) are by default included (see also [CBD]), except when they would explicitly not be included in case 3, when the shape would be closed.
2. Out of band / in band:
    - when no quads of a member have been found, the member will be dereferenced. This allows to publish the member on a separate page.
    - part of the member can be maintained elsewhere when a shape is defined (see 3)
3. By defining a more complex shape with `tree:shape`, also nested entities can be included in the member
4. By putting the triples in a named graph of the object of `tree:member`, all these triples will be matched.

Depending on the goals of the client, it MAY implement the member extraction algorithm to fetch all triples about the entity as intended by the server. The method used within TREE is combination of Concise Bounded Descriptions [CBD], named graphs and the topology of a shape (deducted from the `tree:shape`). The full algorithm is specified in the shape topologies report.

On top of the Member Extraction Algorithm, a client MAY implement the Profile Algorithm.

§ 5. Traversing the search tree

After dereferencing a `tree:Node`, a client MUST extract all (zero or more) `tree:Relation` descriptions from the page. This can be done by searching for `<> tree:relation ?R` triples.

A client MUST follow the object of the relation's `?R tree:node ?object` triple, unless the client is able to prune the branch reachable from that node (see further).

A client MAY also extract the `tree:Relation`'s `tree:remainingItems` if it exists. If it does, it will be an integer indicating the remaining items to be found after dereferencing the node.

When traversing, a client SHOULD detect faulty search trees by keeping a list of already visited pages.

When dereferencing the object of a `tree:node` triple, the client MUST follow redirects.

> NOTE: Allowing redirects allows servers to rebalance their search trees over time.

A client can assume completeness of members intended by the search tree when it derefenced all node links.

§ 6. Pruning branches

In search trees, a `tree:Relation` will likely be typed using one of its subclasses:

- For partial string matching, `tree:PrefixRelation`, `tree:SubstringRelation`, and `tree:SuffixRelation`.
- For comparing various datatypes, `tree:GreaterThanRelation`, `tree:GreaterThanOrEqualToRelation`, `tree:LessThanRelation`, `tree:LessThanOrEqualToRelation`, `tree:EqualToRelation`, and `tree:NotEqualToRelation`.
- For geospatial trees, `tree:GeospatiallyContainsRelation`.

A client decides, based on their own tasks, what type of relations are important to implement. Each relation is a comparator function that helps deciding whether or not the subtree reachable from the `tree:node` link can be pruned. A relation can be interpreted as a boolean equation as follow:

1. **Left-hand side**: The set of terms identified by the predicate `tree:path` from members that are targets of the relations.
2. **Operator**: The type of relation, identified by its `rdf:type`, such as `tree:GreaterThanRelation` or `tree:SuffixRelation`.
3. **Right-hand side**: The reference value identified by `tree:value`, used for comparison with the values associated with the set of terms defined in point 1.

Thus, for every member materialized at the resource identified by the IRI of a `tree:Node`, each term of the member specified by `tree:path` (point 1) MUST satisfy the comparison defined by the `rdf:type` of the relation (point 2) and the reference value given in `tree:value` (point 3).

The client MUST combine all relations to the same `tree:Node` using a logical AND.

The members that the client is able to find in a subtree will be complete relative to the position in the search tree.

EXAMPLE 3 ¶

```
<> tree:relation [
        a tree:GreaterThanRelation ; # the type of the relation deciding the operator
        tree:node ex:Node2 ; # for the left-hand: all members from here
        tree:path dct:created ; # for the left-hand: the path pointing at the term(s) in
        tree:value "2024-12-16T12:00:00Z"^^xsd:dateTime # the right-hand
    ],[
        a tree:SubstringRelation ;
        tree:node ex:Node2 ;
        tree:path dct:title ;
        tree:value "osa"
    ] .
```

In the example above the subtree reachable from `ex:Node2` will contain all remaining members that are both created later in time than the given timestamp *and* will have the provided substring in the title. The client can choose to prune all links to other nodes if this is the only thing it is interested in. Alternatively, the client can choose prune the subtree reachable from `ex:Node2` if it is specifically not looking for members with the given substring, *or* when it is not interested in members created later in time than the given timestamp. Alternatively, it can also score the relation based on the likelihood of returning useful results and created a priority queue that is processed until a top K of results have been found. Mind that when the client is specifically not interested in members created later than the given creation time, but does not understand the SubstringRelation, the client can still prune the relation.

While each type of relation can decide on their own properties, relations will often use the `tree:path` to indicate the path from the member to the object on which the `tree:Relation` applies. For the different ways to express or handle a `tree:path`, we refer to 2.3.1 in the shacl specification. All possible combinations of e.g., `sh:alternativePath` or

`sh:inversePath` in the SHACL spec can be used. The resulting values of the evaluation of the `tree:path`, are the values that must be compared to the `tree:value` object. When multiple results from the path are found, they need to be interpreted as a logical OR: at least one of these values will fulfill the comparator.

A client, in case it wants to process relations that use the `tree:path` property, MUST implement a matching algorithm to check whether the relation is relevant. E.g., a `tree:path` on `(rdfs:label [sh:alternativePath rdfs:comment ] )` will be useful when the client is tasked to filter on `rdfs:comment`.

> NOTE: A server is allowed to refer the `tree:path` to a property that is not materialized in the current response. For the client, if it also needs those triples, we assume in this spec that the client has another way of retrieving those, or already retrieved them from another source.

6.1. Comparing strings, IRIs and time literals

When using comparator relations such as `tree:GreaterThanRelation` and `tree:PrefixRelation` the values MUST be compared as defined in the SPARQL algebra functions.

See the `tree:Relation` definitions in the vocabulary for the specific references to those functions. The TREE specification enforces stricter requirements than SPARQL.

- **String comparisons** MUST be performed using Unicode canonical equivalence. Ordering MUST follow case-sensitive Unicode code point order.
- **IRI comparisons** MUST be treated as simple literal comparisons, consistent with the rules defined for the ORDER BY operation in the SPARQL specification.
- **`xsd:dateTime` literals without a timezone comparisons** MUST be performed by interpreting the `xsd:dateTime` literal as a period of time, with a min inclusive bound of `-14:00` and max exclusive bound of `+14:00`.

> NOTE: It is highly recommended that server developers and data producers provide a timezone in their `xsd:dateTime` literals for time comparison.

6.2. Comparing geospatial features

The `tree:GeospatiallyContainsRelation` is a relation following the definition of geof:sfContains. The client MUST consider the relation when it overlaps with the region of interest defined by the `tree:value` clause.

When using `tree:GeospatiallyContainsRelation`, the `tree:path` MUST refer to a literal containing a WKT string, such as `geosparql:asWKT`.

7. Search forms

Searching through a TREE will allow you to immediately jump to the right `tree:Node` in a subtree. TREE relies on the Hydra search specification for its search forms. It does however extend Hydra with specific search properties (`hydra:IriTemplate`) for different types of search forms, and searches starting from a specific `tree:Node`, to which the search form is linked with `tree:search`. The behaviour of the search form fully depends on the specific property, for which TREE introduces a couple of specific properties:

7.1. Geospatial XYZ tiles search form

Three properties allow to specify a geospatial XYZ tiles template (also known as slippy maps).

1. `tree:longitudeTile` describes the X value
2. `tree:latitudeTile` describes the Y value
3. `tree:zoom` describes the zoom level

All properties expect positive integers.

EXAMPLE 4

```
<https://tiles.openplanner.team/#LatestCollection> a tree:Collection ;
    dcterms:title "A prototype tree:Collection for Linked OpenStreetMap's roads"@en .
<https://tiles.openplanner.team/planet/> a tree:Node ;

    tree:search [
        a hydra:IriTemplate ;
        hydra:template "https://tiles.openplanner.team/planet/20201103-095900/{z}/{x}/{y
        hydra:variableRepresentation hydra:BasicRepresentation ;
        hydra:mapping [
            a hydra:IriTemplateMapping ;
            hydra:variable "x";
            hydra:property tree:longitudeTile;
            hydra:required true
        ],[
            a hydra:IriTemplateMapping ;
            hydra:variable "y";
            hydra:property tree:latitudeTile;
            hydra:required true
        ],[
            a hydra:IriTemplateMapping ;
            hydra:variable "z";
            hydra:property tree:zoom;
            hydra:required true
        ]
    ] .
```

This search form describes a specific search form that uses a quad tree. The zoom level describes the depth, the longitudeTile and latitudeTile describe the x and y index of the pagination. (e.g., on zoom level 0, there's 1 tile, on zoom level 1, there are 4 tiles, etc.).

§ 7.2. Searching through a list of objects ordered by time

Same as the previous example but with the predicate `tree:timeQuery` expecting an `xsd:dateTime`. This time however, when the page itself does not exist, a redirect is doing to happen to the page containing the timestamp. A `tree:path` can indicate the time predicate which is intended.

EXAMPLE 5

```
<https://example.org/#Collection> a tree:Collection ;
    dcterms:title "An example collection with a time search view"@en ;
    tree:view <https://example.org/Node1> .

<https://example.org/Node1> a tree:Node ;
    tree:search [
        a hydra:IriTemplate ;
        hydra:template "https://example.org/{generatedAt}" ;
        hydra:variableRepresentation hydra:BasicRepresentation ;
        hydra:mapping [
            a hydra:IriTemplateMapping ;
            hydra:variable "generatedAt";
            tree:path prov:generatedAtTime;
            hydra:property tree:timeQuery;
            hydra:required true
        ]
    ] .
```

8. Vocabulary

Namespace: `https://w3id.org/tree#`

Prefixes:

```
@prefix tree: <https://w3id.org/tree#>.
@prefix hydra: <http://www.w3.org/ns/hydra/core#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

8.1. Classes

8.1.1. tree:Collection

A collection has members that MAY adhere to a certain shape.

8.1.2. tree:Node

A `tree:Node` is a node that MAY contain links to other dereferenceable resources that lead to a full overview of a `tree:Collection`.

8.1.3. tree:Relation

An entity that describes a relation between two `tree:Nodes`.

The `tree:Relation` has specific sub-classes that implement a more specific type between the values. These types are described in the ontology (all classes are `rdf:subClassOf tree:Relation`). `tree:Relation` can be express in term of SPARQL algebra functions:

- String, Date or Number comparison:
  - `tree:PrefixRelation` — All elements in the related node have this prefix. MUST conform to the STRSTARTS SPARQL function.
  - `tree:SubstringRelation` — All elements in the related node have this substring. MUST conform to the SUBSTR SPARQL function.
  - `tree:SuffixRelation` — All members of this related node end with this suffix. MUST conform to the STRENDS SPARQL function.
  - `tree:GreaterThanRelation` — the related Node's members are greater than the value. MUST conform to the SPARQL Operator Mapping.
  - `tree:GreaterThanOrEqualToRelation` — similar to ↑
  - `tree:LessThanRelation`
  - `tree:LessThanOrEqualToRelation`
  - `tree:EqualToRelation`
  - `tree:NotEqualToRelation`
- Geo-spatial comparison (requires the node values to be WKT-strings):
  - `tree:GeospatiallyContainsRelation` — MUST conform to geof:sfContains.

For more detailed behavior related to specific types, see the section Traversing the search tree.

8.1.4. tree:ConditionalImport

A class to import a file or a stream based on a `tree:path` of properties. This way it can import the necessary data for complying to the SHACL shape, or evaluating a relation type.

8.2. Properties

8.2.1. tree:relation

Links a node to a relation

Domain: `tree:Node`

Range: `tree:Relation`

8.2.2. tree:remainingItems

Remaining number of items of this node, the items in its children included.

Domain: `tree:Relation`

Range: `xsd:integer`

8.2.3. tree:node

The URL to be derefenced when this relation cannot be pruned.

Domain: `tree:Relation`

Range: `tree:Node`

8.2.4. tree:value

The contextual value of this node: may contain e.g., a WKT-string with the bound of a rectangle, may contain a string, an integer, or even link to another resource where clear comparison rules apply.

Domain: `tree:Relation`

8.2.5. tree:path

A property path, as defined by SHACL, that indicates what resource the `tree:value` affects.

See [](#relations)

Domain: `tree:Relation`

8.2.6. tree:view

Links the collection to the current `tree:Node`.

Domain: `tree:Collection`

Range: `tree:Node`

8.2.7. tree:search

Links a `tree:Node` to a `hydra:IriTemplate`. The search form will search the remaining items of the node.

Domain: `tree:Node`

Range: `hydra:IriTemplate`

8.2.8. tree:shape

The SHACL shape the members of the collection adhere to.

Domain: `tree:Collection`

Range: `sh:NodeShape`

8.2.9. tree:member

Links to the collection's items that are the `sh:targetNode`s of the SHACL shape defined with `tree:shape`.

Domain: `tree:Collection`

8.2.10. tree:import

Imports a document containing triples needed for complying to the SHACL shape, or for evaluating the relation.

8.2.11. tree:conditionalImport

Imports a document only when the client is interesting in a specific `tree:path`.

8.2.12. tree:zoom

A search form parameter: the zoom level of the tile cfr. OSM convention.

As defined by Slippy Map Tilenames in OpenStreetMap

8.2.13. tree:longitudeTile

A search form parameter: the X tile number from longitude cfr. OSM convention.

As defined by Slippy Map Tilenames in OpenStreetMap

8.2.14. tree:latitudeTile

A search form parameter: the Y tile number from latitude cfr. OSM convention.

As defined by Slippy Map Tilenames in OpenStreetMap

8.2.15. tree:timeQuery

A search form parameter: accompagnied by a `tree:path`, it indicates the property on which a time search can be done

8.2.16. tree:viewDescription

Links together any HTTP response with a view description on which things like retention policies, contact information of a server, etc. can be found.

Domain: `tree:Node`

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

> EXAMPLE 6 ¶
>
> This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

> Note, this is an informative note.

§ References

§ Normative References

[CBD]
Patrick Stickler, Nokia. *CBD - Concise Bounded Description*. 3 June 2005. W3C Member Submission. URL: https://www.w3.org/Submission/CBD/

[RDF-CONCEPTS]
Graham Klyne; Jeremy Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. URL: https://w3c.github.io/rdf-concepts/spec/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[SHACL]
Holger Knublauch; Dimitris Kontokostas. *Shapes Constraint Language (SHACL)*. URL: https://w3c.github.io/data-shapes/shacl/