How-to configure Sparnatural in SHACL

date: 2024-09
Sparnatural version: 9.5.0

Introduction

Welcome to this guide on how to configure Sparnatural !

The Sparnatural SHACL configuration reference documentation lists the available annotations and axioms available to configure Sparnatural. In this documentation you will learn how to use these annotations concretely in an Excel spreadsheet to define the entities, properties, widgets and datasources in order to make your Sparnatural explorer as appealing as possible for your users.

Conventions

URIs are indicated like this.

Headers in the spreadsheet are indicated like this.

 **Important** : this is an important note, pay attention !

 **Advanced note**: this is explaining something advanced. Don't worry if you don't understand all the details at first.

 **Tip**: this is a useful and practical tip.

Prerequisites

1. Make sure you have followed the introductory "Hello Sparnatural" guide to setup your environment to point Sparnatural to your triplestore and adjust the browser security settings.
2. You must have a local spreadsheet editor, like Microsoft Excel, or use Google spreadsheet
3. Although not absolutely required, it is good if you have a basic understanding of SHACL specifications[1]. To go further with SHACL, you will also discover in annex of

[1] You can read the official SHACL spec, and here are a few suggestions of introductory materials to get acquainted with SHACL :

The free book « Validating RDF data » : https://book.validatingrdf.com/

This masterclass :

- https://github.com/veleda/shacl-masterclass
- in particular the slides at https://github.com/veleda/shacl-masterclass/tree/main/slides/KGC%202023

This serie of posts :

- https://www.linkedin.com/pulse/ontology-modeling-shacl-getting-started-holger-knublauch-iwlrf
- https://www.linkedin.com/pulse/ontology-modeling-shacl-qualified-cardinality-holger-knublauch-zp8hf/
- https://www.linkedin.com/pulse/ontology-modeling-shacl-sparql-based-constraints-holger-knublauch-qeisf/
- https://www.linkedin.com/pulse/ontology-modeling-shacl-defining-forms-instance-data-holger-knublauch-ann5f/

this manual how you can generate a documentation based on your Sparnatural SHACL configuration

4. For configuring your own datasource queries, you need to be proficient with SPARQL. This is described in annex.

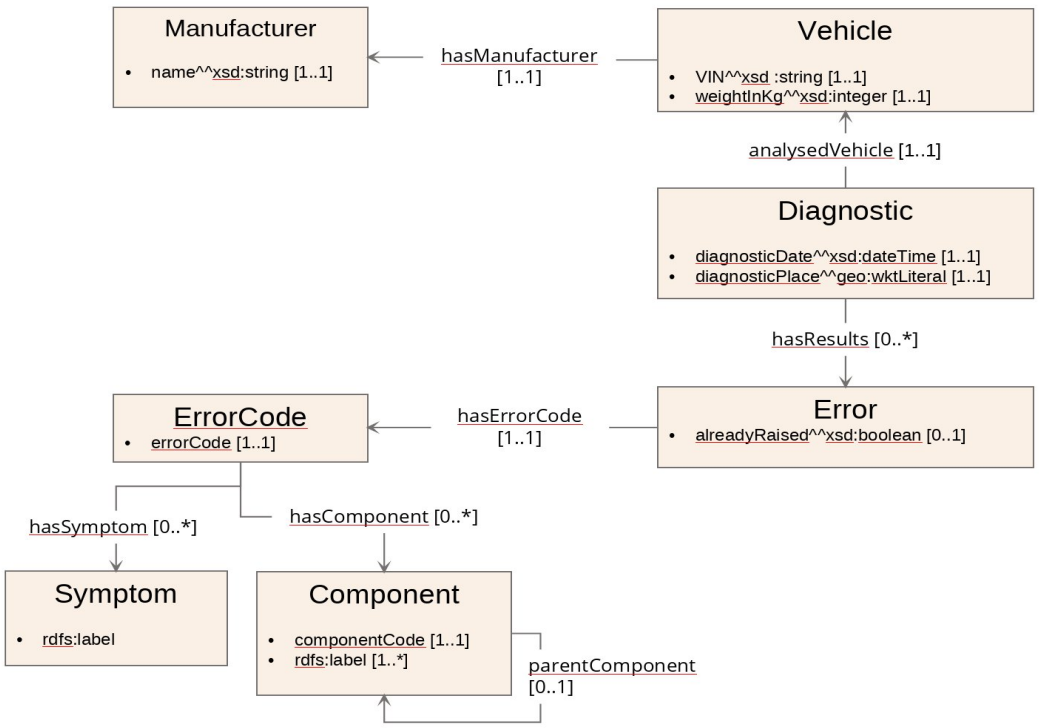
Documentation files

This guide comes with a set of files that you should have ready. Click on the links to download them:

1. **car.ttl** : a sample OWL ontology describing car diagnostics.
2. **car_instances.ttl** : a few manually crafted instances of the sample ontology. Although not strictly required, you should load these instances into your triplestore if you want to follow along and test the example configuration against the dataset.
3. **sparnatural-car-configuration-shacl.xlsx** : the example Sparnatural Excel config file
4. **sparnatural-car-configuration.ttl** : the result of converting the Excel config file with the Excel-2-RDF converter (explanations below). This is the actual Sparnatural configuration file to pass in the “src” attribute of the sparnatural HTML element, if you want to test it to see the final result (but this is not required to follow this documentation).

Structure of the example ontology

For the purpose of this documentation we will use an example ontology, defined in the file **car.ttl**, and described in the following diagram:



This is a simplistic representation of “On-board diagnostic” systems of cars : Vehicles, identified by their Vehicle Identification Number (VIN) have a Manufacturer; Diagnosis (“Diagnostics”) are made on given vehicles at a certain date and a certain place, and can yield Errors. An Error has a code, and a flag indicating if the error was already detected on the same vehicle. Error codes are associated with Symptoms (“Engine Misfire” or "Transmission Slipping") and Components (“Engine”, “Transmission”, “Brakes”). Components are hierarchically organized.
The ontology uses the prefix “odb” associated with the URI http://example.com/ontology/odb#.

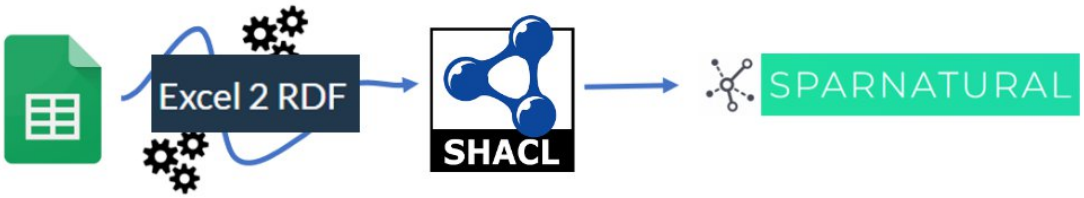
Disclaimer : this “car” ontology sample is a fictitious one, which has only been created for the purpose of testing maximum Sparnatural different functionalities. This ontology is not suitable for a real car diagnostic industrial context !

Setup the configuration spreadsheet

How-to use the spreadsheet and the Excel-2-RDF converter

Sparnatural can be configured by a SHACL specification, and the “Hello Sparnatural” guide explains how to use a simple Excel spreadsheet to start creating a SHACL config for Sparnatural.

The configuration spreadsheet can be edited in a local file or in an online (Google) spreadsheet. Both options are described below. Whether local or online, the conversion of the spreadsheet into SHACL requires a conversion using the Excel-2-RDF converter before it can be read by Sparnatural :



The code of the Excel-2-RDF converter is open-sourced in the xls2rdf Github repository. The Excel-2-RDF converter is available in different packagings:

1. an online REST service
2. an online form where you can upload your file
3. a command-line converter with its documentation
4. a Java library file to be integrated into your application

All these “packagings” behave the same way for the conversion of the spreadsheet in RDF. For the purpose of following this documentation, we suggest either using an online Google spreadsheet and rely on the online conversion service, or simply use a local file and upload it through the online form, and save the resulting SHACL file.

The detailed behaviour of the Excel-to-RDF converter as to how the Excel file is interpreted is out of scope of this guide, and is documented in the online converter service.

Setup for a Google spreadsheet

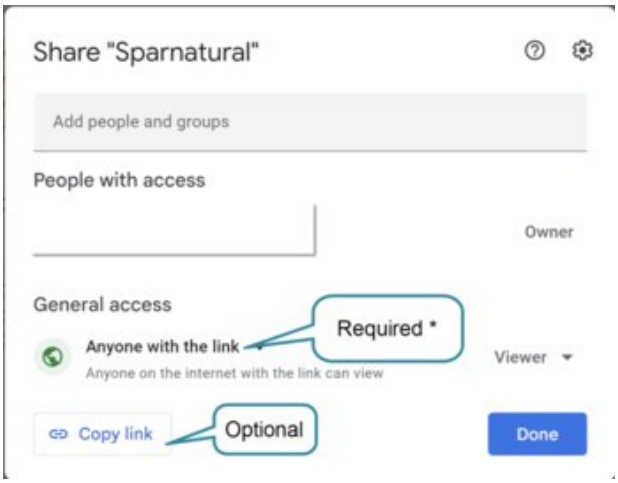
Using a Google spreadsheet has the following advantages:

1. The configuration is “live” : while in the test phase, you can edit your spreadsheet, refresh your Sparnatural HTML page, and it will be updated automatically.
2. Multiple persons can collaborate on the same configuration.

To initialize your configuration spreadsheet:

1. Make a copy of the configuration template
2. Your spreadsheet needs to be publicly visible. You need to share it with the *"Anyone with the link = Viewer"* option. To do this, select the option **Share**.

In the next window, click the **"General access"** button. Select the *"Anyone with the link"* option and press the "Done" button.



After you close the window, copy the URL of the spreadsheet in your browser's address bar.

3. Copy this URL in the cell B2 of the configuration file. Make sure the URL does not end with “/edit#gid=xxxxxxx”, remove this part of the URL manually. The URL should look like *https://docs.google.com/spreadsheets/d/xxxxxxxxxx*”
4. Save the content of cell B3 (in red) : this is the configuration URL that you can pass to the “src” attribute of the <spar-natural> HTML element. You see it starts with https://xls2rdf.sparna.fr : this is the online Excel-2-RDF conversion service that takes the Google spreadsheet URL as a parameter. Each time your Sparnatural page will load, it

will call this URL of the converter, which will in turn trigger the conversion of the Google spreadsheet. The page is connected “live” to the spreadsheet.

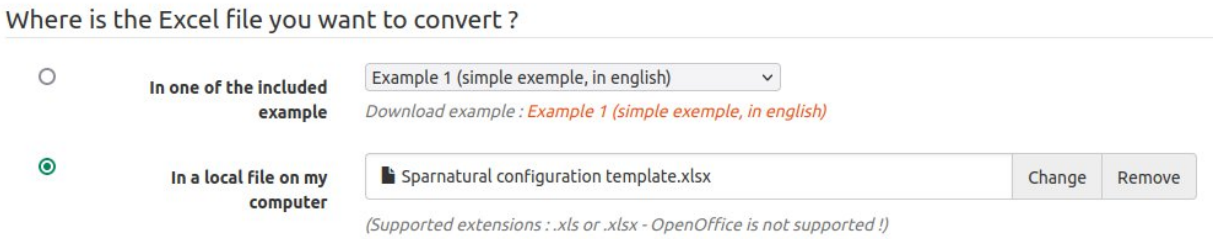
> **Important** : once your configuration is ready, do NOT leave Sparnatural pointing to the live spreadsheet, otherwise your page will depend on the availability of the on-line converter. Instead, save the result of the conversion to a local file “sparnatural-config.ttl”, and adjust the “src” attribute of the <spar-natural> HTML element to point to the local file.

Setup for a local spreadsheet

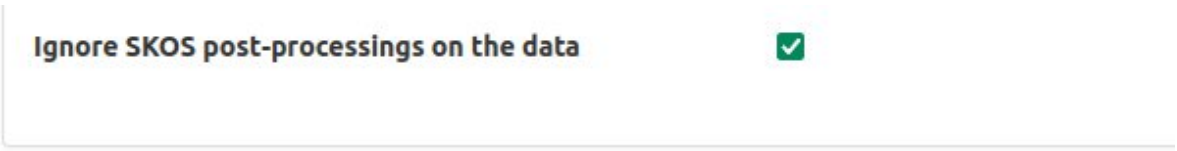
Relying on Google services might not be applicable in every context. It is also possible to design the configuration in a local spreadsheet, and convert it to a SHACL file. The configuration is not live in that case, and you will have to reconvert the file every time you make a change in it.

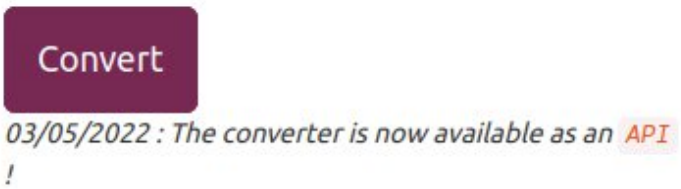
To start a fresh configuration template:

1. Download the configuration spreadsheet template.
2. Edit the content as necessary
3. Go to the online converter at https://skos-play.sparna.fr/play/convert
4. Upload the file in the field “in a local file on my com-puter”:



5. Check the box “Ignore SKOS post-processings on the data”:





6. Click on Convert.
7. Save the resulting file in the same folder as your Sparnatural page.
8. Adjust the “src” attribute of the <spar-natural> HTML element to point to this local file.

Reconvert the file the same way every time you make a change in it.

Declare properties and entities

In this documentation we will work with a local spreadsheet. Download the spreadsheet configuration_template and save it in a local file. You will be working on this local file.

Important : throughout this documentation, we are referring to the columns of the spreadsheet by their header name, e.g. sh:order or sh:targetClass. The header is the green line in bold:

8	*prefix declared in the prefixes tab.*	*This is an integer, e.g. "1", "2", etc.*	*license to access the full set of icons.*	*This should **always** be sh:NodeShape.*	*declared in the "prefixes" tab.*	*English label displayed in*	
9	**URI**	**sh:order^^xsd:integer**	**volipi:iconName**	**rdf:type(separator=",")**	**sh:targetClass**	**rdfs:**	

Each column header corresponds to one configuration property as detailed in the SHACL configuration reference page. The header line does not need to be at a fixed line; it is automagically detected, so don't worry if you add or delete lines before this one.

Important : Beware of hidden columns ! Make sure all the necessary columns are visible in your spreadsheet. Most of the screenshots in this documentation are taken with some hidden columns to save some space.

Adjust the configuration URI and the prefixes

You first need to adjust the URI of your configuration, as well as enter the prefixes used in your knowledge graph. The prefixes will be used in the rest of the configuration.

Configuration IRI

Make sure you are on the "Entities" tab of the configuration template, and edit the content of cell B1. This cell needs to contain the URI of your configuration. It is not very important, unless you plan to share your configuration later. It is typically set to something like "https://data.mydomain.com/sparnatural-config" or to a URL where Sparnatural will be deployed, like "https://mydomain.com/sparnatural-page/sparnatural-config".

Metadata cleanup

Cells B2 and B3 are only useful when working with online Google spreadsheets, so that the configuration is converted and refreshed everytime the Sparnatural page is reloaded. We don't need that in a local file, so simply delete the content of cells B2 and B3. Keep them if you work with a Google spreadsheet.

Prefixes

You need to add additional prefixes from your configuration. Some prefixes are already declared in the "Prefixes" tab. Leave them as they are there, and add additional prefixes if ne-

cessary, by adding new lines. The column A always needs to contain the keyword PREFIX, column B is the prefix name, and column C is the complete URI associated with the prefix.

Important : Note also that a special prefix, “this” is declared in the “Entities” tab. It is set automatically from the Configuration IRI in cell B1. Leave it as it is. It is a special prefix that you will use to declare your entities and properties.

Example

Following the above, in our example configuration we set the configuration IRI to https://data.mydomain.com/ontologies/sparnatural-config, delete the content of cells B2 and B3 in the “Entities” tab, and in the “Prefixes” tab, add our prefix “odb” on line 8, corresponding to the URI http://example.com/ontology/odb#

	A	B	C
1	**Shapes IRI**	https://data.mydomain.com/ontologies/sparnatural-config	
2	**dct:source**		
3	**dct:format**		
4	**rdf:type**	owl:Ontology	
5	**PREFIX**	this	https://data.mydomain.com/ontologies/sparnatural-config/

	A	B	C	D	E	F
1	*You can add more prefixes like the ones below. Default prefixes are already known: foaf, schema, owl, rdfs, etc.* *See full list at https://xls2rdf.sparna.fr/rest/doc.html#default-prefixes-known-in-the-converter*					
2	**PREFIX**	core	http://data.sparna.fr/ontologies/sparnatural-config-core#			
3	**PREFIX**	dash	http://datashapes.org/dash#			
4	**PREFIX**	datasources	http://data.sparna.fr/ontologies/sparnatural-config-datasources#			
5	**PREFIX**	geo	http://www.opengis.net/ont/geosparql#			
6	**PREFIX**	volipi	http://data.sparna.fr/ontologies/volipi#			
7	**PREFIX**	dbpedia	http://dbpedia.org/ontology/			
8	**PREFIX**	odb	http://example.com/ontology/odb#			

Declare simple entities

You need to declare the entities that will be listed in the selector for the subject as well as for the object of each query criteria. Don't hesitate to read the guidelines in the green line above the body of the table to get additional information about the content of each columns. This are the basic columns you need to fill in:

- always use the "this" prefix in the "URI" column to give a meaningful identifier to your entity. You will refer to this identifier later when configuring properties.
- in the sh:targetClass column, enter the URI of the corresponding class from your knowledge graph ontology.
- in column sh:order^^xsd:integer set the display order of the entity to sort the items in Sparnatural's interface. The value must be an integer.
- the volipi:iconName column is where you can copy-paste the code of a Font Awesome free icon you will choose on the website (e.g. "fa-solid fa-car").
- in the rdf:type column set `sh:NodeShape` as the value for every entity ; this is a fixed value that never changes.
- In the sh:nodeKind column, always set `sh:IRI` (except when dealing with literal values, see below).
- then add the label of the entity, in the rdfs:label@xx column (this is the label that will appear in the user interface).

> **Advanced note**: You can change the language of the label by editing the header row. By default the template enables labels in English (rdfs:label@en), and French (rdfs:label@fr). You can adjust the language code after the "@" sign. All the labels in a given column will be tagged with this language. Make sure the language you use matches the "`lang`" parameter of Sparnatural in your web page. More on this in the section about multilingual configuration.

- if you need some, you can also add tooltips in the sh:description@en column. This is not mandatory. Depending on the use-case, the tooltip may provide more contextual information to the user than only the definition from the ontology (e.g. *"Select this entry if you want to search on xxx or yyyy"*).
    - Similar to labels, you can adjust the language code of the tooltips by editing the language code after the "@" symbol in the header line.

> **Tip**: HTML markup is supported in tooltips.

Tip: By using the labels combined with the order, you can group your Entities in a meaningful way, for example by setting a label that contains a hierarchy, such as “Actor > Person” and “Actor > Organization”, and setting those 2 classes next to each other with their order.

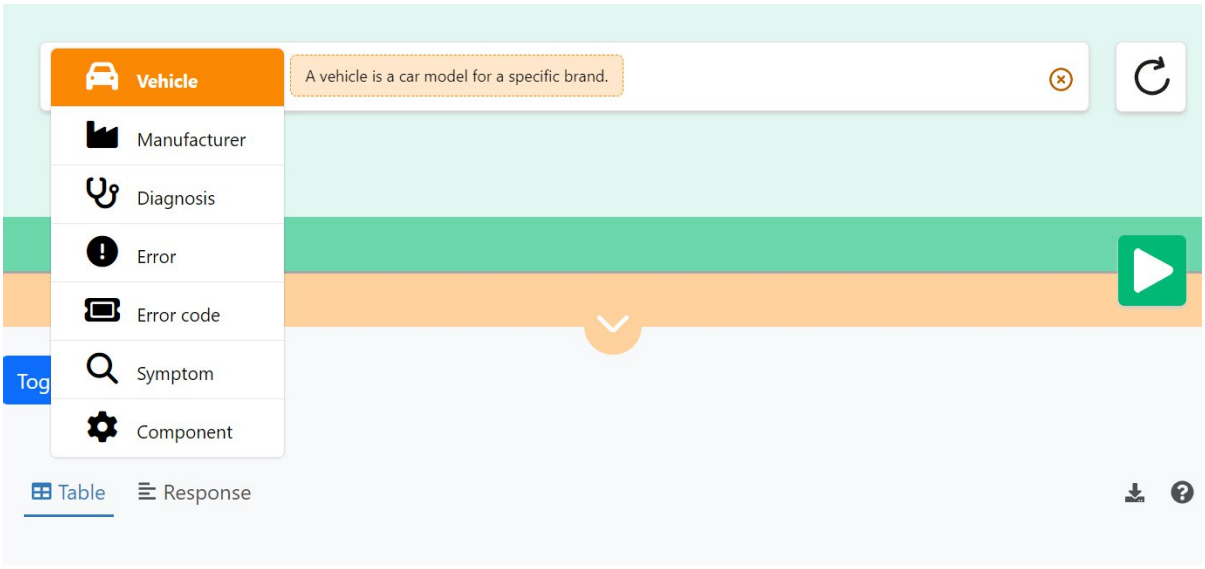
Example

Here in the example we have chosen to list all the existing Entities of the model (you could choose to have only some classes of your model, and not all). We took the same URIs as the ones in the data model and added order, icons, labels and tooltips :

URI of the entities. This column will use the "this" prefix declared in the prefixes tab.	*The sort order of the entity in the class dropdown list. This is an integer, e.g. "1", "2", etc.*	*The Fontawesome icon code for the class, e.g. "fa-duotone fa-user". Search for icon codes at https://fontawesome.com/. Fontawesome provides a limited number of icons for free, and you can buy a license to access the full set of icons.*	*This should **always** be sh:NodeShape.*	*the class in the OWL ontology to which the entity in the configuration corresponds. This column will use the prefix of your ontology declared in the "prefixes" tab.*	*Always set sh:IRI unless the NodeShape you declare is actually used to represent literal nodes, in which case use sh:Literal*	*English label that will be displayed in Sparnatural.*	*The English tooltip for the entity.*
URI	**sh:order^^xsd:integer**	**volipi:iconName**	**rdf:type(separator=",")**	**sh:targetClass**	**sh:nodeKind**	**rdfs:label@en**	**sh:description@en**
this:Vehicle	1	fa-solid fa-car	sh:NodeShape	odb:Vehicle	sh:IRI	Vehicle	A vehicle is a car model for a specific brand.
this:Manufacturer	2	fa-solid fa-industry	sh:NodeShape	odb:Manufacturer	sh:IRI	Manufacturer	A car manufacturer is a company whose main activity is the design, construction and marketing of cars.
this:Diagnostic	3	fa-solid fa-stethoscope	sh:NodeShape	odb:Diagnostic	sh:IRI	Diagnosis	A diagnosis identifies a possible problem on your vehicle. You can request a diagnosis when you suspect a breakdown or malfunction. Using an auto diagnosis kit, the mechanic identifies the problem with your vehicle.
this:Error	4	fa-solid fa-circle-exclamation	sh:NodeShape	odb:Error	sh:IRI	Error	An error is an element that comes up during a diagnosis, which indicates that the vehicle on which the analysis was carried out is encountering a problem.
this:ErrorCode	5	fa-solid fa-ticket	sh:NodeShape	odb:ErrorCode	sh:IRI	Error code	An error code is a set of numbers following a letter corresponding to a problem detected on your vehicle. The letter gives an indication of the family of the defect.
this:Symptom	6	fa-solid fa-arrows-to-eye	sh:NodeShape	odb:Symptom	sh:IRI	Symptom	A symptom is a phenomenon, perceptible or observable character linked to a state, a problem that it allows to detect, of which it is the sign.
this:Component	7	fa-solid fa-gear	sh:NodeShape	odb:Component	sh:IRI	Component	A class representing a component of a vehicle.
this:TrueFalse	8	fa-solid fa-yin-yang	sh:NodeShape		sh:Literal	True / False	
this:Search	9	fa-solid fa-search	sh:NodeShape		sh:Literal	Search...	Use this to search on labels and description

Note also that, at the end of the list, we have a few lines that don’t correspond to classes from the ontology, but are entities that correspond to literal values : `this:TrueFalse` or `this:Search`. More on this later.

We decided that “Vehicle” was an important entry point and set its order to 1. Following this, we can see it appears first in the query builder :



Note how the tooltip displays the definition from the configuration.

Declare simple properties

Now you need to declare the properties that link together the entities from your configuration. For this, move to the “Properties” tab of the spreadsheet.

Tip: We suggest you organize the “Properties” table by sections, each section corresponding to the specification of the properties attached to one given entity in your configuration. Make a colored line for each section, with the name of the entity as the title. In general you are free to arrange the spreadsheet as you want and use any formatting/color option you want. Lines that do no contain a URI in column A will be ignored.

In this tab you need to enter:

- URI column : The URI of the configuration property. This should not be confused with the URI of the property itself. We suggest to use the following syntax for the identifier:
  - the « this » prefix from your configuration ;
  - then concatenate the entity local name from column ^sh:property ;
  - add an underscore for better readability ;
  - and finally the property local name from your ontology from the sh:path column

  Which gives us for example : this:Vehicle_VIN.
- the sh:path column is for the actual URI of the property from your ontology, using your ontology namespace, for example odb:VIN.

- the ^sh:property column contains the entity URI to which the property is assigned (the « subject » of an RDF predicate). It is a reference to one of the Entities URI from the Entities tab, using the « this » prefix.
- in the sh:name@en column set the label of the property to be shown in the interface.
    - adjust the language code of the labels by editing the language code after the "@" symbol in the header line.
- if needed you can add a tooltip in the sh:description@en column ;
    - adjust the language code of the tooltips by editing the language code after the "@" symbol in the header line.
- the sh:node column contains the identifier of a Sparnatural entity to which the property points to (the "object" of an RDF predicate). This is the identifier of an entity from the first tab;

Advanced note: Sparnatural can also read sh:class SHACL constraints that point to a class that is itself the sh:targetClass of an entity in the config

- the dash:searchWidget column is used to configure the way the values can be selected in the query builder (see "widget" section below) : when you start designing your configuration we suggest using core:ListProperty to obtain simple populated lists using the data ; you can then refine this to other more appropriate values after.

Advanced note: it is possible that a single property has more than one entity as its domain. You can specify more than one entity identifier in the ^sh:property column, by separating them with a comma.

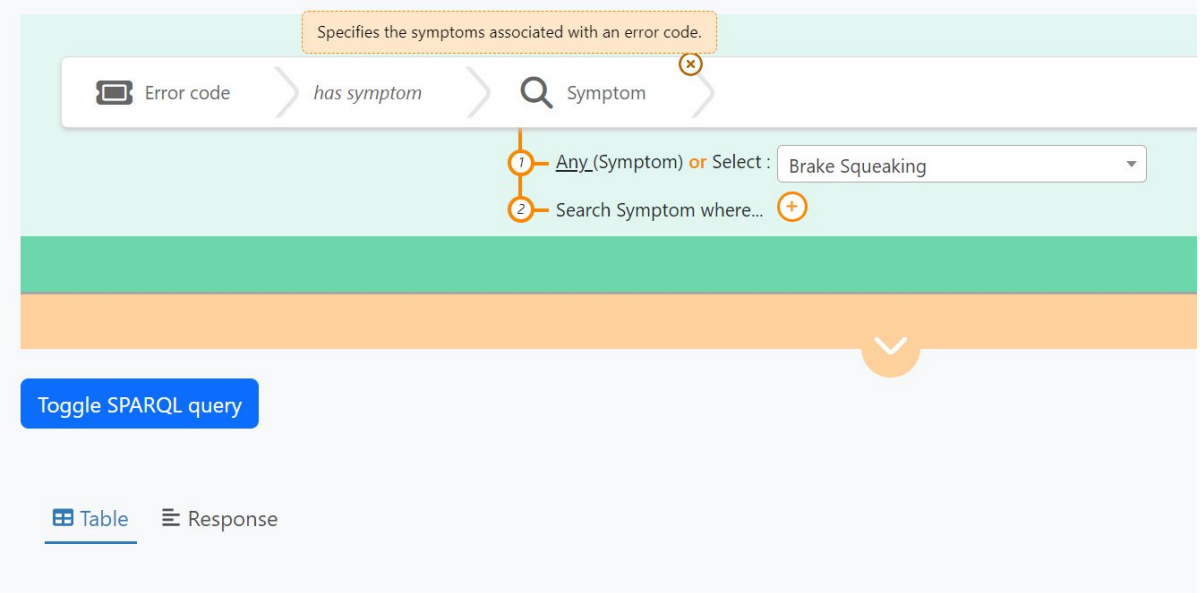
Advanced note: min and max cardinalities (sh:minCount and sh:maxCount) are currently not used by Sparnatural. However they are an important part of any SHACL specification, and they could by used by Sparnatural in the future. Hence they are included in the configuration template columns.

Example

Note how the table is organized with one section per entity; note also how each property refers to the entity to which it is attached in the ^sh:property column (in each "section" the ^sh:property is always the same), and the entity to which it refers to in the sh:node column, for most of them.

URI	sh:path	^sh:property(separator=",")	sh:name@en	sh:description@en	sh:datatype	sh:node	dash:searchWidget
Vehicle							
this:Vehicle_VIN	odb:VIN	this:Vehicle	has VIN	Specifies the Vehicle Identification Number (VIN) of the vehicle.	xsd:string		core:AutocompleteProperty
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	has manufacturer	Specifies the manufacturer of the vehicle.		this:Manufacturer	core:ListProperty
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	this:Vehicle	has diagnosis	The property is the inverse of odb:analysedVehicle.		this:Diagnostic	core:NonSelectableProperty
this:Vehicle_weightInKg	odb:weightInKg	this:Vehicle	weight in kg		xsd:integer		core:NumberProperty
Manufacturer							
this:Manufacturer_name	odb:name	this:Manufacturer	has name	Specifies the name of the manufacturer.	xsd:string		core:NonSelectableProperty
Diagnosis							
this:Diagnostic_diagnosticDate	odb:diagnosticDate	this:Diagnostic	has diagnosis date	Defines the date on which the diagnosis occurs.	xsd:date		core:TimeProperty-Date
this:Diagnostic_analysedVehicle	odb:analysedVehicle	this:Diagnostic	analysed vehicle	Specifies that the vehicle has been analyzed, to identify a potential problem.		this:Vehicle	core:AutocompleteProperty
this:Diagnostic_hasResults	odb:hasResults	this:Diagnostic	has results	Specifies the results, from the analysis.		this:Error	core:NonSelectableProperty
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	this:Diagnostic	has diagnosis place	Defines the place where the diagnosis occurs.	geo:wktLiteral		core:MapProperty
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	this:Diagnostic	returns code	The property is a shortcut between Diagnosis and Error Code.		this:ErrorCode	core:ListProperty
Error							
this:Error_alreadyRaised	odb:alreadyRaised	this:Error	already raised	Attribute indicating whether an error has already been detected previously.	xsd:boolean	this:TrueFalse	core:BooleanProperty
this:Error_hasErrorCode	odb:hasErrorCode	this:Error	has error code	Specifies the error code relating to an error reported during a diagnostic.		this:ErrorCode	core:ListProperty
ErrorCode							
this:ErrorCode_errorCode	odb:errorCode	this:ErrorCode			xsd:string		core:NonSelectableProperty
this:ErrorCode_hasSymptom	odb:hasSymptom	this:ErrorCode	has symptom	Specifies the symptoms associated with an error code.		this:Symptom	core:ListProperty
this:ErrorCode_hasComponent	odb:hasComponent	this:ErrorCode	has component (list)	Specifies the components impacted by an error code.		this:Component	core:ListProperty
this:ErrorCode_hasComponent_tree	odb:hasComponent	this:ErrorCode	has component (tree)	Specifies the components impacted by an error code.		this:Component	core:TreeProperty

As a result we can see - when index.html is refreshed - the object properties *in italic* appear in the interface, between the Entities items :



The tooltip of the property is displayed if it was added before in the configuration file.

We see a dropdown list appears when the range of the query (i.e. the “object” entity of the assertion) is chosen. As explained before, the way the selected values are to be displayed depends on the type (dash:searchWidget) of the property, also referred to as the “widget” of the property.

Configure value selection widgets

Each property needs to be associated with a value selection widget (although Sparnatural has some default behavior is none is indicated, we recommend to always set it explicitely). The value selection is how the user will enter a search value for this criteria.

To set a value selection widget, enter one of the following predefined values in the dash:searchWidget column.

Widget type (dash:searchWidget)	Description
core:ListProperty	dropdown list widget, useful for list of values < 500
core:AutocompleteProperty	autocomplete search field, useful when the list of values is larger
core:TreeProperty	tree browsing widget, useful with some tree-shaped values, typically SKOS hierarchies, part-of hierarchies, etc;
core:MapProperty	map selection widget (GeoSPARQL queries)
core:SearchProperty, core:StringEqualsProperty, core:GraphDBSearchProperty	string search widget, searched as regex or as exact string
core:TimeProperty-Date, core:TimeProperty-Year	date range widget (date or year precision)
core:BooleanProperty	boolean widget (true/false, yes/no values…)
core:NonSelectableProperty	no value selection (useful for 'intermediate' Entities whose values don't need to be displayed)
core:NumberProperty	Numeric range widget

All of them are already fully documented in the reference documentation for Sparnatural widgets.

The choice of the widget is driven by how we want the user to select a value, and how many different values are available (e.g. lists are good only when the values are relatively small, typically less than 500 distinct values).

Tip: lists and autocomplete can work for both object properties (having IRIs as their values) and datatype properties (having literals as their values).

Example

Note how the properties in our configuration use different kinds of widgets:

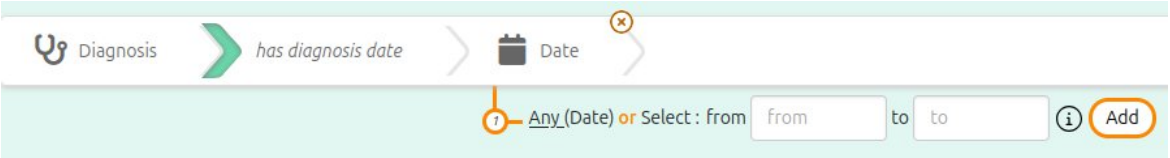
URI	sh:path	sh:name@en	dash:searchWidget
Vehicle			
this:Vehicle_VIN	odb:VIN	has VIN	core:AutocompleteProperty
this:Vehicle_hasManufacturer	odb:hasManufacturer	has manufacturer	core:ListProperty
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	has diagnosis	core:NonSelectableProperty
this:Vehicle_weightInKg	odb:weightInKg	weight in kg	core:NumberProperty
Manufacturer			
this:Manufacturer_name	odb:name	has name	core:NonSelectableProperty
Diagnosis			
this:Diagnostic_diagnosticDate	odb:diagnosticDate	has diagnosis date	core:TimeProperty-Date
this:Diagnostic_analysedVehicle	odb:analysedVehicle	analysed vehicle	core:AutocompleteProperty
this:Diagnostic_hasResults	odb:hasResults	has results	core:NonSelectableProperty
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	has diagnosis place	core:MapProperty
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	returns code	core:ListProperty
Error			
this:Error_alreadyRaised	odb:alreadyRaised	already raised	core:BooleanProperty
this:Error_hasErrorCode	odb:hasErrorCode	has error code	core:ListProperty
ErrorCode			
this:ErrorCode_errorCode	odb:errorCode		core:NonSelectableProperty
this:ErrorCode_hasSymptom	odb:hasSymptom	has symptom	core:ListProperty
this:ErrorCode_hasComponent	odb:hasComponent	has component (list)	core:ListProperty
this:ErrorCode_hasComponent_tree	odb:hasComponent	has component (tree)	core:TreeProperty
Symptom			
this:Symptom_label	rdfs:label	label	core:SearchProperty
Component			
this:Component_componentCode	odb:componentCode	has component code	core:SearchProperty
this:Component_label	rdfs:label	label	core:SearchProperty
this:Component_label_or_code	[sh:alternativePath (odb:componentCode rdfs:label)]	label or code	core:SearchProperty

On Vehicle, the `odb:VIN` property is set as an autocomplete. Being a long technical identifier, having an autocomplete will help user selecting a correct value.

On Vehicle, the `odb:manufacturer` property uses a `core:ListProperty` because there is a limited list of possible car manufacturers, so using a list is convenient.

On Manufacturer, we have set the `odb:name` property as `core:NonSelectableProperty`, because we assume the user will never have to search or select a value for the name of a Manufacturer.

On Diagnosis, `odb:diagnosticDate` uses a `core:TimeProperty-Date` widget as the values in the graph have an `xsd:date` datatype. This will result in a date range selection to be displayed in the UI :



Disable optional or negative queries on some properties

According to the SPARQL syntax, Sparnatural offers also a way to express optional or negative assertions, corresponding in SPARQL to OPTIONAL or negative "FILTER NOT EXISTS" query patterns.

This is enabled by default for every property. But you can choose to disable both options for each individual property in the Properties tab of the spreadsheet. If you set "false" as the value of the column core:enableOptional^^xsd:boolean or core:enableNegation^^xsd:boolean, the corresponding optional or negative option will be disabled for this property.

This is related to the minimum cardinality of the property : is the property is mandatory with a sh:minCount value to 1, then it is pointless for a user to ask for situations when the property is not present, or to retrieve the property in an optional way; and it could have an impact on performance. So it is better to disable it in some situations.

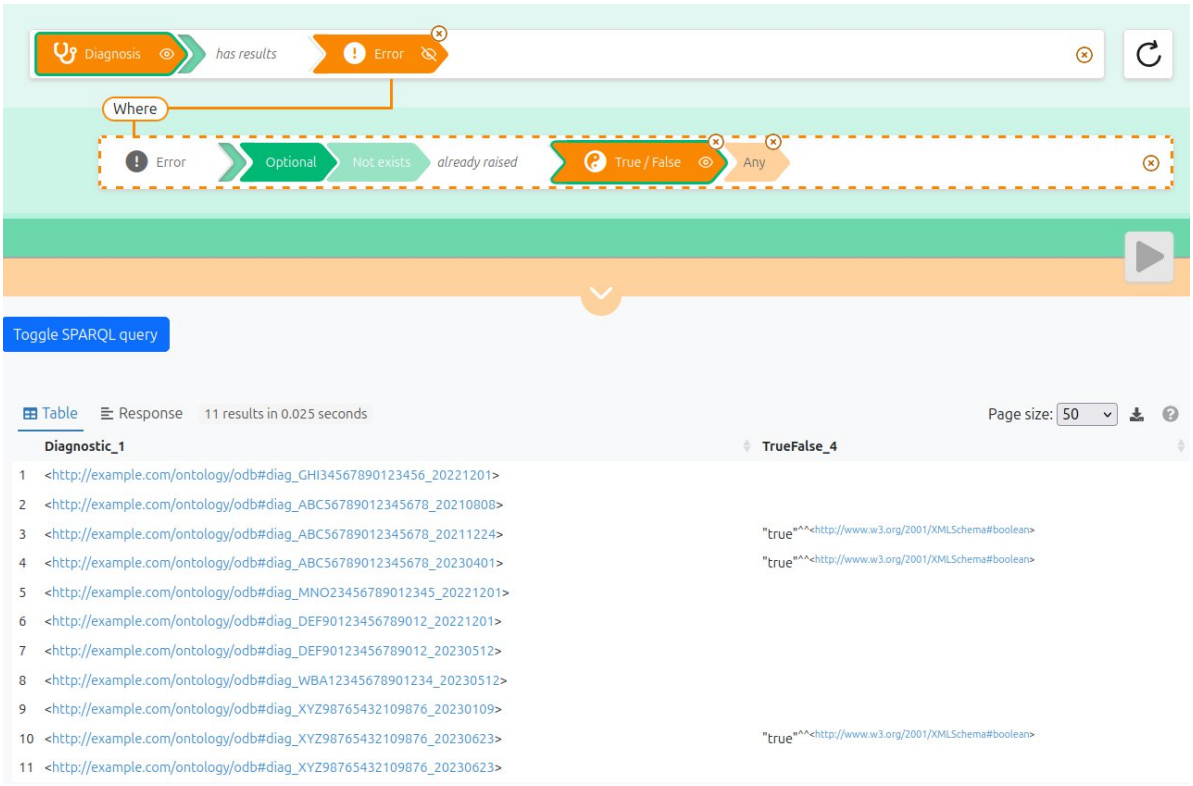
Example

Here we can see in the latest two columns from the screenshot that the optional and negative parameters have been set to false (“FAUX”) for every property where the minimum cardinality (sh:minCount) is 1, indicating the property is actually always present in the knowledge graph.

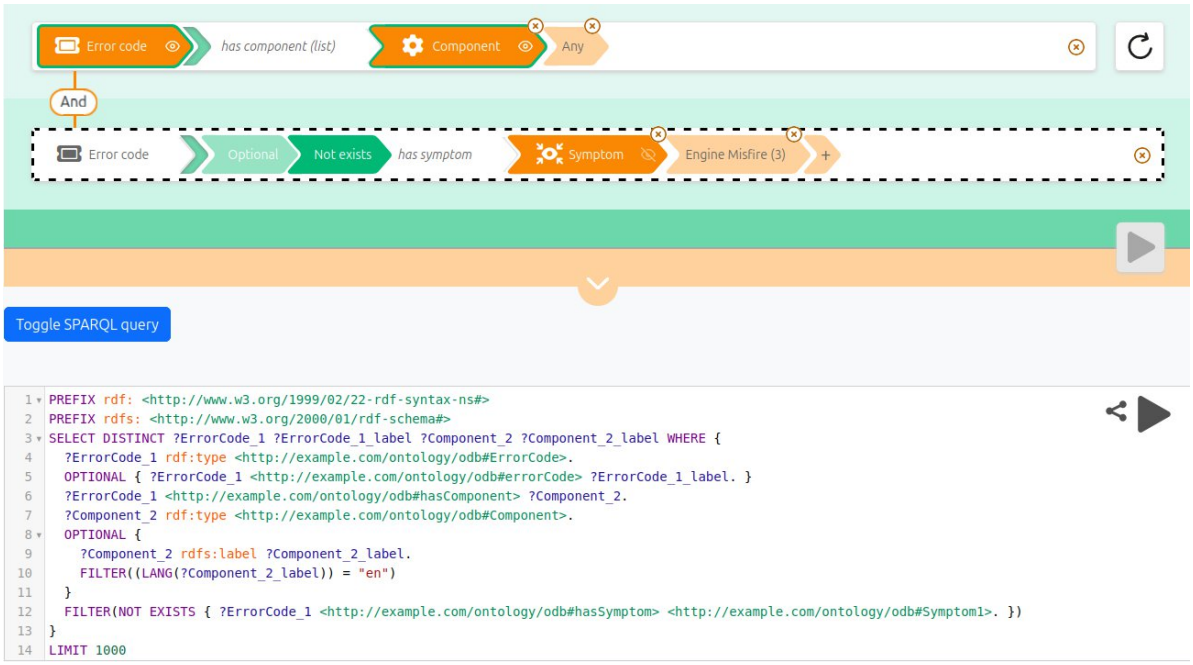
Note how no value is set for the odb:alreadyRaised property, because our knowledge graph sets this property to “true” when the error was already raised on the same vehicle, but does not set the property otherwise (it is not explicitly set to false).

URI	sh:path	^sh:property(separator=",")	sh:minCount^ ^xsd:integer	sh:maxCount^ ^xsd:integer	sh:node	core:enableOptional^ ^xsd:boolean	core:enableNegation^ ^xsd:boolean
Vehicle							
this:Vehicle_VIN	odb:VIN	this:Vehicle	1	1		FAUX	FAUX
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	1	1	this:Manufacturer	FAUX	FAUX
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	this:Vehicle			this:Diagnostic		
this:Vehicle_weightInKg	odb:weightInKg	this:Vehicle	1	1		FAUX	FAUX
Manufacturer							
this:Manufacturer_name	odb:name	this:Manufacturer	1	1		FAUX	FAUX
Diagnosis							
this:Diagnostic_diagnosticDate	odb:diagnosticDate	this:Diagnostic	1	1		FAUX	FAUX
this:Diagnostic_analysedVehicle	odb:analysedVehicle	this:Diagnostic	1	1	this:Vehicle	FAUX	FAUX
this:Diagnostic_hasResults	odb:hasResults	this:Diagnostic			this:Error	FAUX	FAUX
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	this:Diagnostic	1	1		FAUX	FAUX
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	this:Diagnostic			this:ErrorCode	FAUX	FAUX
Error							
this:Error_alreadyRaised	odb:alreadyRaised	this:Error		1	this:TrueFalse		
this:Error_hasErrorCode	odb:hasErrorCode	this:Error	1	1	this:ErrorCode	FAUX	FAUX
ErrorCode							
this:ErrorCode_errorCode	odb:errorCode	this:ErrorCode	1	1		FAUX	FAUX
this:ErrorCode_hasSymptom	odb:hasSymptom	this:ErrorCode			this:Symptom		
this:ErrorCode_hasComponent	odb:hasComponent	this:ErrorCode			this:Component		

The following screenshot shows an optional query pattern on the “already raised” property which is optional (cardinality [0..1]). Let’s imagine we’d like to display all the results following this property no matter *if actually there are some* (or not). This enables to obtain a list of results even in case when the value isn’t there :



This one shows a negative pattern where we want to search for every component related to an error code that does not have “Engine Misfire” as a symptom :



However, note how the “Vehicle has manufacturer” property does not show the optional and not exists options, because they have been disabled in the configuration:

Hide properties or entities

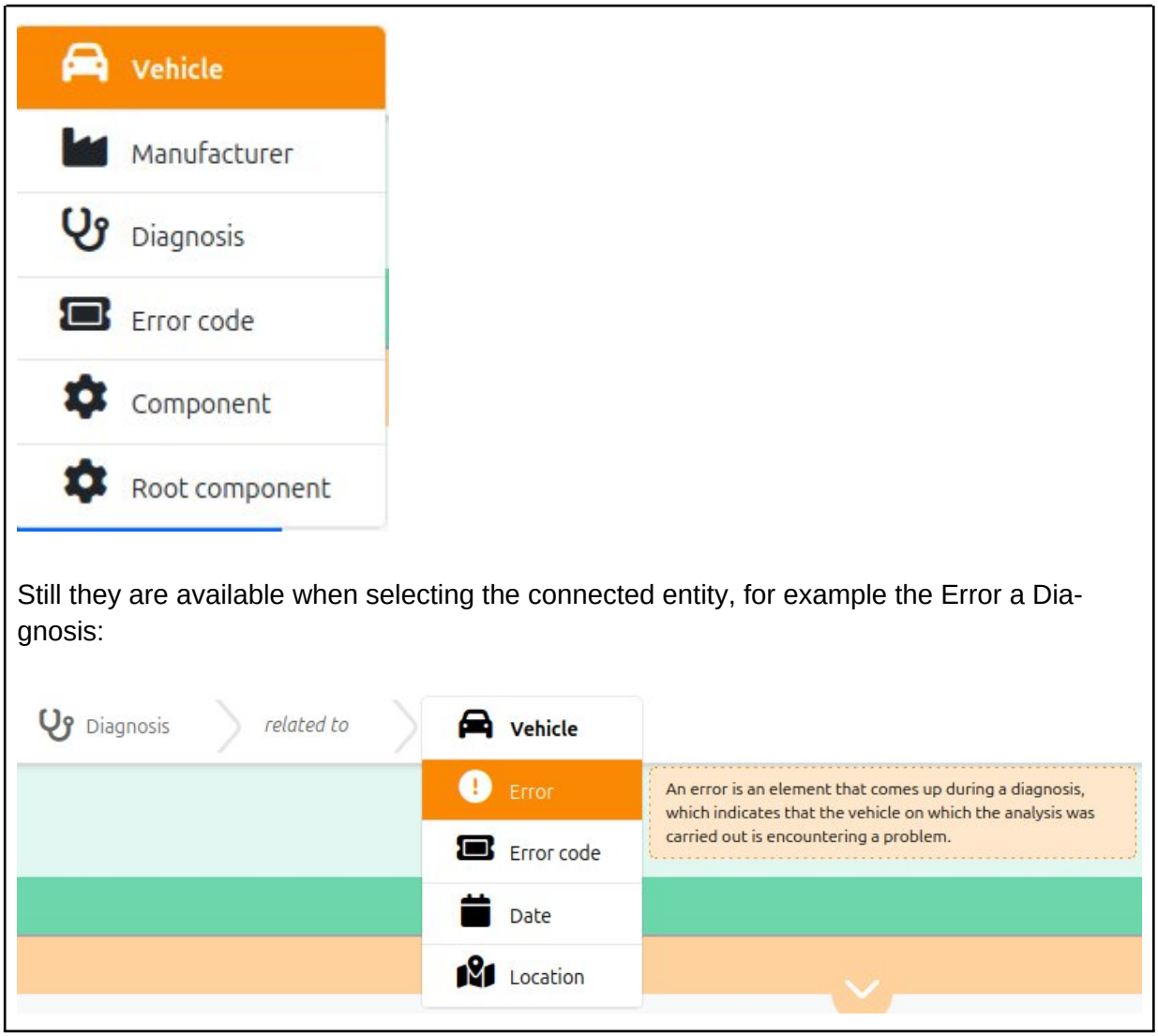
You may want, from an initial SHACL specification, to hide some properties or some entities. To do so, flag them with « true » as the value of the sh:deactivated column. A deactivated property shape will not be displayed in the UI, and a deactivated node shape will be hidden from the first list of entities shown. It will still be available as the object of the criteria if it is used as range of certain properties.

Example

In our sample ontology Error and Symptom are flagged as deactivated:

9	URI	sh:nodeKind	rdfs:label@en	sh:deactivated^^xsd:boolean
13	this:Error	sh:IRI	Error	VRAI
14	this:ErrorCode	sh:IRI	Error code	
15	this:Symptom	sh:IRI	Symptom	VRAI
			Component	

Hence, those classes don't appear in the initial list:



Datasources : populate lists and autocomplete fields

Indicate the default label property for each entity

For each entity, you can “flag” one (and only one !) of its properties as the default label property of this entity. Typical default label properties are `rdfs:label`, `foaf:name`, `skos:prefLabel`, etc. Setting this flag is useful for 2 purposes:

1. Displaying the label in Sparnatural drop-down lists, and in autocomplete search fields. This is what is covered in the rest of this section.
1. Automatically fetching a label for the entity when it is selected by the user (using the “eye” icon in the orange arrow), so that the query result table uses it to display the entity, instead of the URI. This will be covered in another section of this documentation.

To do this, you set the value `dash:LabelRole` in the `dash:propertyRole` column[2].

[2] For more information on the DASH property roles vocabulary see https://datashapes.org/propertyroles.html

Example

VIN are vehicles identifiers. As such, and without other human-readable labelling property or vehicle, the odb:VIN property is marked as the default label property for Vehicle.
Similarly, on Manufacturer, the odb:name property (being the sole property of this entity !) is marked as the default label property of Manufacturers.

(here with a few hidden columns for readability).

URI	sh:path	^sh:property(separator=",")	sh:name@en	dash:searchWidget	dash:propertyRole
Vehicle					
this:Vehicle_VIN	odb:VIN	this:Vehicle	has VIN	core:AutocompleteProperty	dash:LabelRole
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	has manufacturer	core:ListProperty	
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	this:Vehicle	has diagnosis	core:NonSelectableProperty	
this:Vehicle_weightInKg	odb:weightInKg	this:Vehicle	weight in kg	core:NumberProperty	
Manufacturer					
this:Manufacturer_name	odb:name	this:Manufacturer	has name	core:NonSelectableProperty	dash:LabelRole
Diagnosis					
this:Diagnostic_diagnosticDate	odb:diagnosticDate	this:Diagnostic	has diagnosis date	core:TimeProperty-Date	
this:Diagnostic_analysedVehicle	odb:analysedVehicle	this:Diagnostic	analysed vehicle	core:AutocompleteProperty	
this:Diagnostic_hasResults	odb:hasResults	this:Diagnostic	has results	core:NonSelectableProperty	
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	this:Diagnostic	has diagnosis place	core:MapProperty	
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	this:Diagnostic	returns code	core:ListProperty	

Note that Diagnosis do not have any properties flagged with dash:LabelRole, because none of its properties qualify as a good human-readable label.

Tip: sometimes the default label property for an entity is available to the user as a property that can be searched on. For example Persons might have “name” as their default label property, and you want the user to search on person names with an autocomplete widget. But sometimes you want the default label property to be hidden in the query builder, and you simply need it to be fetched in the result table. In that case, proceed exactly as normal, except that you can mark the corresponding property with sh:deactivated so that it does not appear in the UI. More on this below.

Default datasource behavior

By default, properties using a ListProperty widget will leverage the dash:propertyRole flag to provide the following default behavior : the list will contain the value of the property flagged as “dash:LabelRole” of the entity that is indicated in the “sh:node” column. The list is sorted alphabetically.

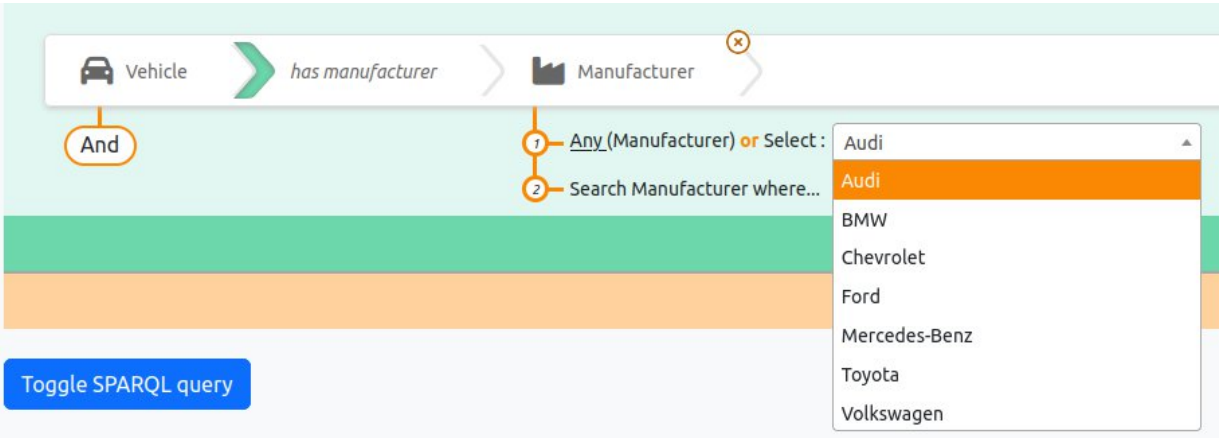
Example

In our configuration, the odb:hasManufacturer property on Vehicle has been set to a List-

Property, because the list of possible car manufacturers is small. Manufacturers have the odb:name property, that has been flagged as their default label property : (here with a few hidden columns for readability)

URI	sh:path	^sh:property(separator=",")	sh:node	dash:searchWidget	dash:propertyRole
Vehicle					
this:Vehicle_VIN	odb:VIN	this:Vehicle		core:AutocompleteProperty	dash:LabelRole
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	this:Manufacturer	core:ListProperty	
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	this:Vehicle	this:Diagnostic	core:NonSelectableProperty	
this:Vehicle_weightInKg	odb:weightInKg	this:Vehicle		core:NumberProperty	
Manufacturer					
this:Manufacturer_name	odb:name	this:Manufacturer		core:NonSelectableProperty	dash:LabelRole
Diagnosis					
this:Diagnostic_diagnosticDate	odb:diagnosticDate	this:Diagnostic		core:TimeProperty-Date	
this:Diagnostic_analysedVehicle	odb:analysedVehicle	this:Diagnostic	this:Vehicle	core:AutocompleteProperty	t

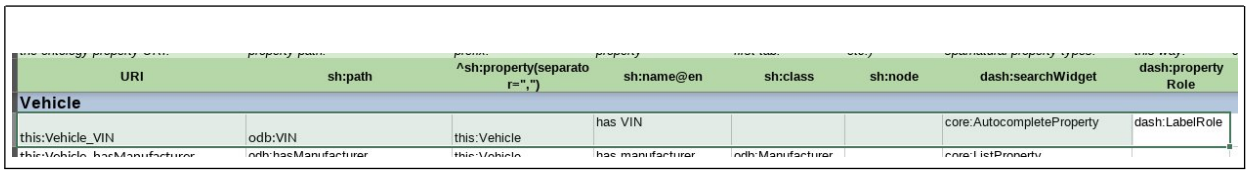
In the query builder, the dropdown list for selecting car manufacturers will be populated with the odb:name property of the manufacturers, sorted alphabetically :



By default, properties using an AutocompleteWidget will leverage the dash:propertyRole flag to provide the following default behavior : the search field will search on the content of the property flagged as “dash:LabelRole” of the entity that is indicated in the “sh:node” column. The search is done at any position in the character string.

Example

There is no such property using an AutocompleteProperty and leveraging a dash:LabelRole flag in our configuration. The property odb:VIN on Vehicle is indeed using an Autocomplete-Property, but it is a literal value, and this will be explained later. It is itself marked as the dash:LabelRole for Vehicle, but it is purely accidental. Remember that the dash:LabelRole flag is sought on the entities indicated on sh:node, and the odb:VIN property does not have any sh:node indicated :



URI	sh:path	^sh:property(separator=",")	sh:name@en	sh:class	sh:node	dash:searchWidget	dash:propertyRole
Vehicle							
this:Vehicle_VIN	odb:VIN	this:Vehicle	has VIN			core:AutocompleteProperty	dash:LabelRole
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	has manufacturer	odb:Manufacturer		core:ListProperty	

There is no default behaviour for TreeProperty.

In most of the cases, the default behaviour is sufficient to deal with common use-cases. In more advanced situations, other means of configuring the datasources of lists and autocomplete fields are possible, and documented below.

Use predefined datasources

You may want to override the default list or autocomplete behaviour. For this you use a “datasource” to populate them in a different way. For example, we may want the label displayed in the dropdown list to contain the number of occurrences, or to concatenate two properties; or we may want an autocomplete search field to search only on the beginning of the string, or use a custom full-text search operator (e.g. bif:contains for Virtuoso).

For that purpose you can use the datasources:datasource column of the Properties tab. The datasource of a dropdown list populates the list, the datasource of an autocomplete property feeds the autocomplete proposals. TreeProperty also requires two datasources; the configuration of tree datasources is covered in the advanced configuration section.

Sparnatural comes with off-the-shelves datasources, in tab “sparnatural-config-core” of the spreadsheet. Here you can find a list of preconfigured datasources corresponding to different widget types for lists, autocomplete (search) and tree.

List of possible widget types	List of preconfigured datasources	List of preconfigured queries
core:AutocompleteProperty	datasources:list_dctermstitle_alpha	datasources:query_list_label_alpha
core:ListProperty	datasources:list_dctermstitle_count	datasources:query_list_label_count
core:TimeProperty-Date	datasources:list_dctermstitle_alpha_with_count	datasources:query_list_label_alpha_with_count
core:TimeProperty-Year	datasources:list_foafname_alpha	datasources:query_list_label_with_range_alpha
core:SearchProperty	datasources:list_foafname_count	datasources:query_list_label_with_range_alpha_with_count
core:GraphDBSearchProperty	datasources:list_foafname_alpha_with_count	datasources:query_list_label_with_range_count
core:NonSelectableProperty	datasources:list_rdfslabel_alpha	datasources:query_list_URI_alpha
core:LiteralListProperty	datasources:list_rdfslabel_count	datasources:query_list_URI_count
core:BooleanProperty	datasources:list_rdfslabel_alpha_with_count	datasources:query_list_URI_or_literal_alpha
core:StringEqualsProperty	datasources:list_schemaname_alpha	datasources:query_list_URI_or_literal_alpha_with_count
core:TreeProperty	datasources:list_schemaname_count	datasources:query_list_URI_or_literal_count
	datasources:list_schemaname_alpha_with_count	datasources:query_literal_list_alpha
	datasources:list_skospreflabel_alpha	datasources:query_literal_list_alpha_with_count
	datasources:list_skospreflabel_count	datasources:query_literal_list_count
	datasources:list_skospreflabel_alpha_with_count	datasources:query_search_label_bifcontains
	datasources:list_URI_alpha	datasources:query_search_label_contains
	datasources:list_URI_count	datasources:query_search_label_strstarts
	datasources:list_URI_or_literal_alpha	datasources:query_search_literal_contains
	datasources:list_URI_or_literal_alpha_with_count	datasources:query_search_literal_strstarts
	datasources:list_URI_or_literal_count	datasources:query_search_URI_contains
	datasources:literal_list_alpha	datasources:query_tree_children
	datasources:literal_list_alpha_with_count	datasources:query_tree_children_with_count
	datasources:literal_list_count	datasources:query_tree_root_noparent
	datasources:search_dctermstitle_bifcontains	datasources:query_tree_root_noparent_with_count
	datasources:search_dctermstitle_contains	datasources:query_tree_root_domain
	datasources:search_dctermstitle_strstarts	
	datasources:search_foafname_bifcontains	
	datasources:search_foafname_contains	
	datasources:search_foafname_strstarts	
	datasources:search_rdfslabel_bifcontains	
	datasources:search_rdfslabel_contains	
	datasources:search_rdfslabel_strstarts	
	datasources:search_schemaname_bifcontains	
	datasources:search_schemaname_contains	

The predefined datasources are documented in the datasource documentation of Sparnatural, but we give some simple indications to select the adequate one for your use-case:

- datasources beginning by “list” are for ListProperty, while datasources beginning by “search” are for AutocompleteProperty.
- The identifier of the property indicates which property Sparnatural uses to display the entry or search on it : rdfs:label, foaf:name, dcterms:title, schema:name, skos:prefLabel
- List datasources come in 3 variants : “alpha” is pure alphabetical, count is sorted by descending number of occurrences, “alpha_with_count” is alphabetical but displays the number of occurrences in parenthesis.
- Search datasources come in 3 variants : “strstarts” looks for the string at the beginning of the property, “contains” looks for the string anywhere in the property, “bifcontains” is specific to Virtuoso and will look for the string anywhere in the property but as a complete word/token.

If your configuration doesn’t use the dash:LabelRole flag described above, a typical frequent choice to populate a list is the datasource “datasource:list_rdfslabel_alpha” which will populate a list with the rdfs:label of the values, sorted alphabetically.

Advanced note: if you look at the SPARQL queries (e.g. by navigating to the URI of one “query_list_xxxx” datasource), you will notice that the default provided queries do not use the range class as a criteria in the query, mostly for performance reasons. They assume that a given property always refers to a single type of entity. If you have a property that can refer to

multiple classes as range, then you need to use one of the provided query that includes "with_range" in its name (e.g. datasources:query_list_label_with_range_alpha), and inject the property name in it (see following section)

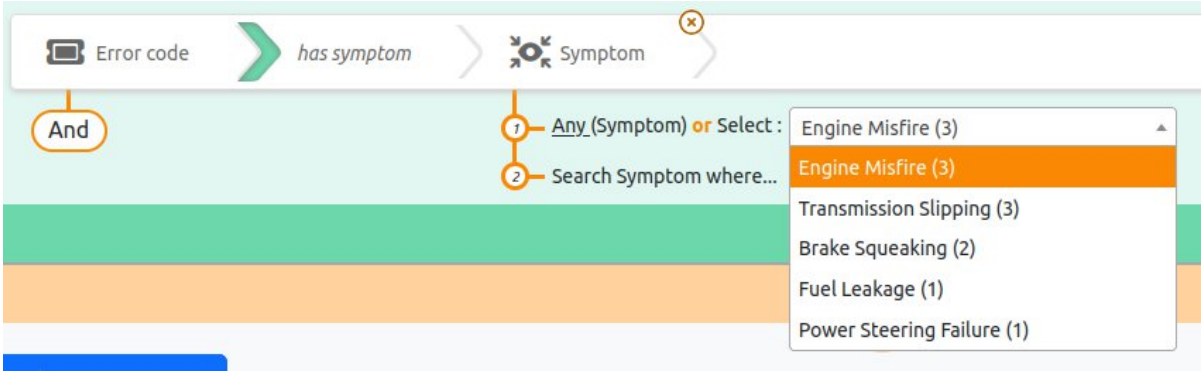
Advanced note: if you don't specify any datasource, and there is no dash:LabelRole flag on the target entity for the property, Sparnatural will default to datasources:list_URI_or_literal_alpha for lists and to datasources:search_URI_contains or datasources:search_literal_contains (depending if the range class is marked as a literal or not, see below). You will most probably never leave these defaults and always specify a datasource.

Example

On ErrorCode, the property odb:hasSymptom is configured to be a ListProperty. We would like that this list contains the label (rdfs:label) of the symptom, with the number of occurrences of that symptom for all error codes. This is provided by one of the predefined datasources, datasources:list_rdfslabel_count, which we set in the datasources:datasource column for this property (here shown with a few hidden columns for readability)

URI	sh:path	^sh:property(separator=",")	sh:node	dash:searchWidget	datasources:datasource
ErrorCode					
this:ErrorCode_errorCode	odb:errorCode	this:ErrorCode		core:NonSelectableProperty	
this:ErrorCode_hasSymptom	odb:hasSymptom	this:ErrorCode	this:Symptom	core:ListProperty	datasources:list_rdfslabel_count
this:ErrorCode_hasComponent	odb:hasComponent	this:ErrorCode	this:Component	core:ListProperty	this:list_componentCode_alpha
this:ErrorCode_hasComponent_tree	odb:hasComponent	this:ErrorCode	this:Component	core:TreeProperty	
Symptom					
this:Symptom_label	rdfs:label	this:Symptom		core:SearchProperty	
Component					
this:Component_componentCode	odb:componentCode	this:Component		core:SearchProperty	

The dropdown list will contain the list of Symptoms with the count in parenthesis, ordered by decreasing number of occurrences :



Use a predefined query with your own property

When your data model uses a property to label entities other than one of the 5 for which pre-configured datasources exist, you can create your custom one, based on one of the pre-defined query (alpha, count or alpha_with_count for lists, or query_search_label_contains or strstarts for autocomplete), in which your property will be “injected”.

To do so, go to “Datasources” tab of your spreadsheet and write down the URI of the new datasource you want to create in column A, using the “this:” namespace, using a name as explicit as possible. Then:

- in rdf:type column, always set the value datasources:SparqlDatasource
- In the datasources:queryTemplate column, pick one of the query from the sparnatural-config-core tab you will copy-paste in the corresponding column. The queries identifiers start with “datasources:query_list...” or “datasources:query_search...”
- In the datasources:labelProperty column, enter the URI of the label property in your data, either as a complete URI (surrounded by “<” “>”) or as a prefixed one. Your custom datasource is created, and can refer to its URI from the “Properties” tab in the “datasources:datasource” column.

The predefined queries for list properties are:

- datasources:query_list_label_alpha / count / alpha_with_count: lists the values in the dropdown, either
    - using the label with a simple alphabetical sort (“alpha” variant)
    - using the label with the count of occurrences in parenthesis, ordered by inverse number of occurrences (“count” variant)
    - or using an alphabetical sort and keeping the number of occurrences in parenthesis (“alpha_with_count” variant)

  These queries do NOT use the range class criteria.
- datasources:query_list_label_with_range_alpha / count / alpha_with_count: same as previous, but using the range class criteria in the query.
- datasources:query_list_URI_alpha / count / alpha_with_count: fill the dropdown with the URI of values. Literals are ignored.
- datasources:query_list_URI_or_literal_alpha / count / alpha_with_count: fill the dropdown with either the URI or the literal value. This is the default behaviour for a List-Property widget pointing to an entity without a default label property.
- datasources:query_literal_list_alpha / count / alpha_with_count: fill the dropdown with the literal value of the property. URIs are ignored.

The predefined queries for an autocomplete properties are:

- datasources:query_search_label_bifcontains: uses Virtuoso specific bif:contains operator to implement the autocomplete proposals, on a specific labelling property of the range entity.
- datasources:query_search_label_contains: uses contains() method – will search anywhere in the string, case insensitive, on a specific labelling property of the range entity.
- datasources:query_search_label_strstarts: uses strstarts() method – will search only at the beginning of the string, case insensitive, on a specific labelling property of the range entity.

- datasources:query_search_literal_contains: does not search on the label of the range entity, but rather directly on the literal value of the property. Useful when creating auto-complete on a literal value. Uses contains() to search anywhere in the literal.
- datasources:query_search_literal_strstarts: same as previous, with strstarts() – searches only at the beginning of the literal.
- datasources:query_search_URI_contains: same as before, but for resources : searches on the URI of the value.

Example

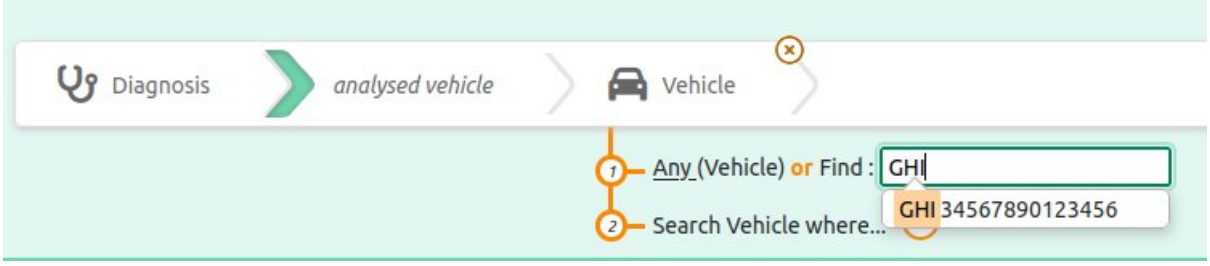
On the Diagnosis class, we would like to fine-tune the way Vehicles are searched in the "analyzed vehicle" property. By default, search is performed anywhere in the identifier, but to limit noise, we would like it to search only on the beginning of it. For this, we create a custom datasource "this:search_VIN_strstarts", using the predefined query datasources:query_search_label_strstarts, and we indicate the label property odb:VIN in column datasources:labelProperty :

URI of the datasource in the configuration. This is the value that should be referenced from the "datasources:datasource" column in the properties tab	*This must **always** be datasources:SparqlDatasource*	*A reference to the query template that this datasource relies on. use only when the query is reused for the same label property. Use EITHER the queryString column, OR the queryTemplate column, but not both.*	*Only if you used datasources:queryTemplate, the label property to inject into the query template. This must be a valid complete URI, including "<...>".*
URI	**rdf:type**	**datasources:queryTemplate**	**datasources:labelProperty**
this:search_VIN_strstarts	datasources:SparqlDatasource	datasources:query_search_label_strstarts	odb:VIN

We then refer to this new datasource identifier in the datasources:datasource column of the odb:analyzedVehicle property:

URI	sh:path	^sh:property(separator=",")	sh:node	dash:searchWidget	datasources:datasource
Diagnosis					
this:Diagnostic_diagnosticDate	odb:diagnosticDate	this:Diagnostic		core:TimeProperty-Date	
this:Diagnostic_analysedVehicle	odb:analysedVehicle	this:Diagnostic	this:Vehicle	core:AutocompleteProperty	this:search_VIN_strstarts
this:Diagnostic_hasResults	odb:hasResults	this:Diagnostic	this:Error	core:NonSelectableProperty	
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	this:Diagnostic		core:MapProperty	
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	this:Diagnostic	this:ErrorCode	core:ListProperty	

The search is then implemented by looking at the beginning of the identifier only:



Create datasources from custom SPARQL

You can also provide your own custom SPARQL query to populate a list or an autocomplete field. This is covered in the advanced configuration section and requires knowledge of SPARQL query language.

Declare literal entities

Explicit literal entities

Sparnatural UI always display a subject entity, a property/link, and an object entity. The subject entity is necessarily a resource (URI or blank node), but the object entity may correspond to a resource or a literal, such as “Date”, “Identifier”, “Label”, corresponding to underlying datatypes such as xsd:date, xsd:string or rdf:langString. You can declare these entities that correspond to literal values.

SHACL allows to create shapes that correspond to literal nodes in the RDF, and this is what Sparnatural uses in this case to declare entities that actually correspond to literal values and note resources.

To declare such an entity, fill in a new line in the Entities tab as described previously, with the following differences:

1. Do not enter anything in the sh:targetClass column (since those entities actually don’t correspond to resources).
2. In the sh:nodeKind column, set the value sh:Literal. This indicates this entity is a literal[3].
3. You refer to that entity as the “range” of a property in the Properties tab in the usual sh:node column.
4. You will never use those entities as the domain of other properties (in the ^sh:property column), as they correspond to literal values, and literal cannot be the subject of triples in RDF.

Tip: Either you can declare a single entity for all literal values, such as “this:Attribute”, so that all literal properties are “grouped” under a generic “Attribute” entry, or you can choose to decompose by datatype, such as “Text”, “Date”, “Boolean”, or you can even decompose by properties, with one literal entity per literal property (e.g. “Coverage” class corresponding to “coverage” property), which imply some kind of duplication. The strategy to use depends on how you would like things to be presented to your users.

The consequence of declaring an entity with sh:nodeKind = sh:Literal and no sh:targetClass is that the generated SPARQL query will never contain an rdf:type criteria for such objects, since they are literal values. Also, those entities won’t appear in the initial classes menu as they will never be used as the domain of other properties (only as range).

Example

Our model include an odb:alreadyRaised flag which is a boolean, on class odb:Error. This

[3] For more information on SHACL node kind values, refer to the node kind paragraph of the SHACL specification.

is reflected in Sparnatural configuration with the corresponding property configured as a BooleanProperty. The property uses the sh:node column to refer to the entity this:True-False :

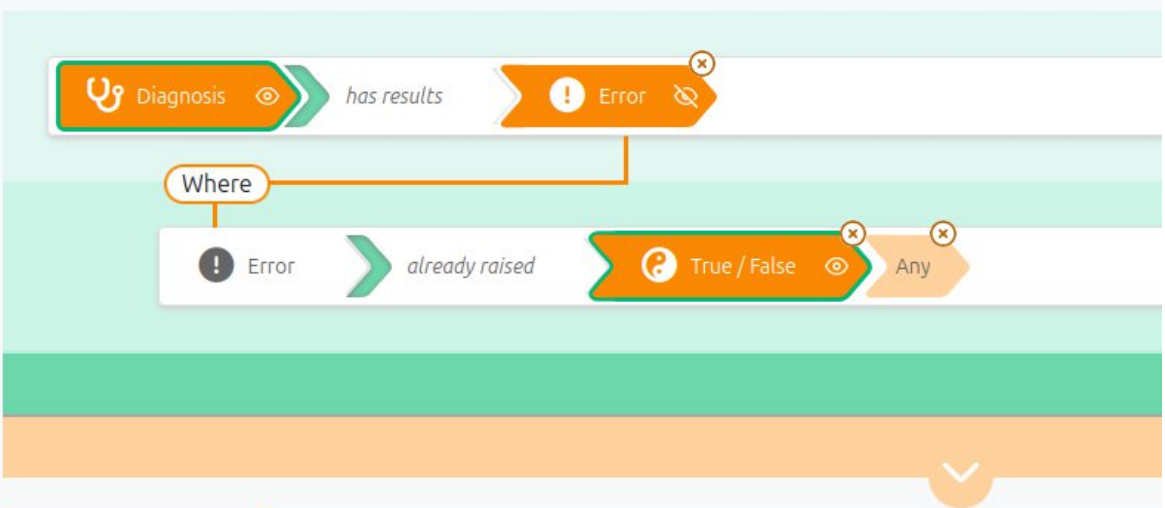
URI	sh:path	^sh:property(separator=",")	sh:node	dash:searchWidget
Error				
this:Error_alreadyRaised	odb:alreadyRaised	this:Error	this:TrueFalse	core:BooleanProperty
this:Error_hasErrorCode	odb:hasErrorCode	this:Error	this:ErrorCode	core:ListProperty

The entity this:TrueFalse corresponds to a literal in the entity tab:

URI	sh:order^^xsd:integer	volipi:iconName	rdf:type(separator=",")	sh:targetClass	sh:nodeKind	rdfs:label@en	rdfs:label@fr
this:TrueFalse	8	fa-solid fa-yin-yang	sh:NodeShape		sh:Literal	True / False	Vrai / Faux

(no target class, and sh:Literal in sh:nodeKind column).

We can test the corresponding behaviour in the query builder by clicking on the “eye” button to select the “True / False” entity related to Error by the property “already raised”:




```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?Diagnostic_1 ?TrueFalse_4 WHERE {
  ?Diagnostic_1 rdf:type <http://example.com/ontology/odb#Diagnostic>;
    <http://example.com/ontology/odb#hasResults> ?Error_2.
  ?Error_2 rdf:type <http://example.com/ontology/odb#Error>;
    <http://example.com/ontology/odb#alreadyRaised> ?TrueFalse_4.
}
LIMIT 1000
```

Note how the query does *not* include an rdf:type criteria for ?TrueFalse_4 variable.

Default literal entities

Sparnatural has some default behaviour for properties that correspond to literal values but that do not use an explicit literal entity as their range. For example a property with an explicit value for sh:datatype, such as xsd:string or xsd:integer. This allows Sparnatural to work with plain SHACL specifications (not customized for Sparnatural UI).

The default entities that Sparnatural has for literal values are:

- "Text" for properties that have datatype xsd:string or rdf:langString
- "Number" for properties that have datatype xsd:integer, xsd:decimal, xsd:long, xsd:int, xsd:short, etc.
- "Date" for properties that have datatype xsd:date, xsd:dateTime, xsd:gYear
- "Location" for properties that have datatype geo:wktLiteral
- "Other" for properties with a datatype not in one of the other category.

This is good since it allow Sparnatural to work with a plain SHACL specification, but when designing a Sparnatural UI, we don't recommend to rely on these default properties but rather to always design your own entities.

Example

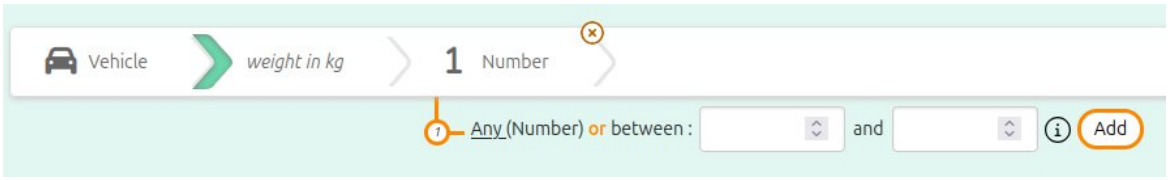
Note how Vehicles are associated to 2 literal properties in our model : their VIN (odb:VIN) and their weight in kilograms (odb:weightInKg). These 2 properties are not explicitly associated to an entity in the sh:node column:

URI	sh:path	^sh:property(separator=",")	sh:nodeKind	sh:datatype	sh:node	dash:searchWidget
Vehicle						
this:Vehicle_VIN	odb:VIN	this:Vehicle	sh:Literal	xsd:string		core:AutocompleteProperty
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	sh:IRI		this:Manufacturer	core:ListProperty
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	this:Vehicle	sh:IRI		this:Diagnostic	core:NonSelectableProperty
this:Vehicle_weightInKg	odb:weightInKg	this:Vehicle	sh:Literal	xsd:integer		core:NumberProperty

Note the default behaviour in the UI:

And then when selecting "Number" we see the "weight in kb" property with a number range

widget:



Map properties to the underlying knowledge graph

So far we introduced entities in the configuration corresponding exactly to one class, and properties in the configuration corresponding exactly to one predicate. But configuring Sparnatural using SHACL allows to provide your users with a slightly different view of the underlying graph structure. Typically you might want to show them a simplified view of the more elaborate structure in the graph with entities corresponding to a selection of resources and properties corresponding to a path in the graph. To do this, the property shapes are associated to SHACL property paths, and node shapes are defined with a query as their target.

This is standard SHACL (it is not specific to Sparnatural) and this is done in the sh:path column for properties, and in the sh:target + sh:select columns for entities.

Follow the “recipes” below that will guide you on how to fill in these columns for different situations.

Query a sequence of properties (using a shortcut)

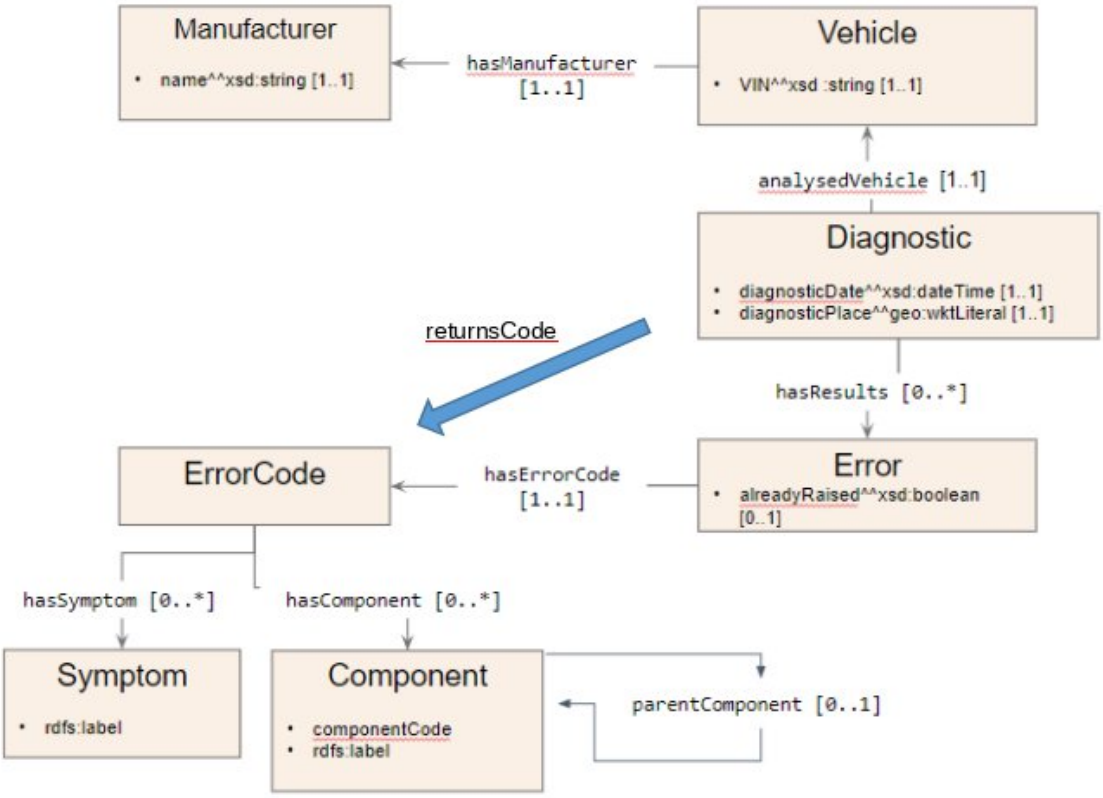
The most frequent use-case for simplifying the user view is when two entities in your data model are connected through one (or more) intermediate entity that you would like to hide in Sparnatural. For example: *“Persons live in City, and City is part of Country”*. Suppose what you would like to show to your users in the query builder is simply “*Persons live in Country”*, hiding the “City” entity.

You will do this with a “*SHACL sequence path*”, which is written with parenthesis “**( )”**, containing the list of properties to follow in sequence separated by a whitespace. In our simple example this would be something like “(ex:lives_in ex:is_part_of)”. This means: *“follow the lives_in property, then follow the is_part_of property”*. This will be translated into the SPARQL property path ex:lives_in/ex:is_part_of.

Note that you can traverse more than two properties by adding more property identifiers inside the parenthesis, in order, separated by a whitespace.

Example

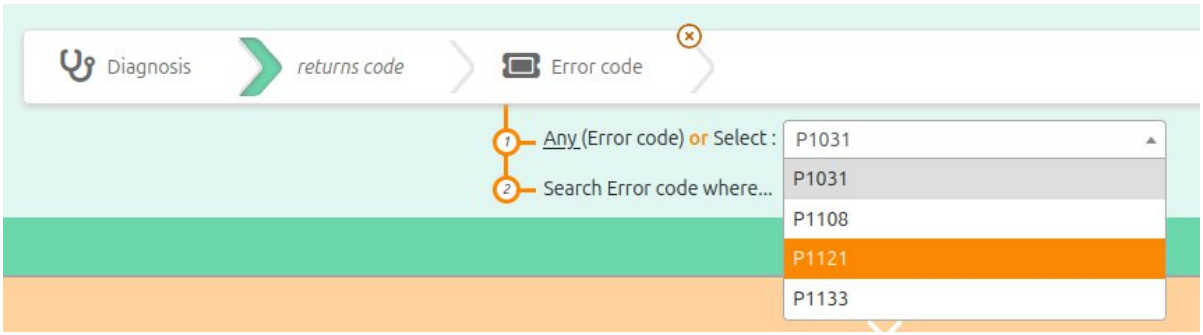
Let's figure out, starting with the "Diagnostic" class of cars ontology,that you would like to go straightly to the Error Code, going through the "Error" item that doesn't interest you that much :



This shortcut is to be created as a new property in the configuration file, while specifying a sequence path in the sh:path column of the spreadsheet, namely (odb:hasResults odb:hasErrorCode)

URI	sh:path	^sh:property(separator=",")	sh:node
Manufacturer			
this:Manufacturer_name	odb:name	this:Manufacturer	
Diagnosis			
this:Diagnostic_diagnosticDate	odb:diagnosticDate	this:Diagnostic	
this:Diagnostic_analysedVehicle	odb:analysedVehicle	this:Diagnostic	this:Vehicle
this:Diagnostic_hasResults	odb:hasResults	this:Diagnostic	this:Error
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	this:Diagnostic	
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	this:Diagnostic	this:ErrorCode

In the UI, the Error Code appears to be directly linked to the Diagnosis, so that we can directly obtain the list of Error Codes corresponding to a given Diagnosis :



(in that case, we chose to also provide the user to traverse explicitely from Diagnosis to Error to Error Code, but sometimes you want to completely hide some entities)

Query inverse properties

Another frequent use-case where the user view differs from the underlying graph structure is when you want to provide the user with an inverse relationship that does not exist in the data. For example if you have *"City is part of Country"* in your graph, you may want to provide the user with the ability to navigate with *"Country contains City"*.

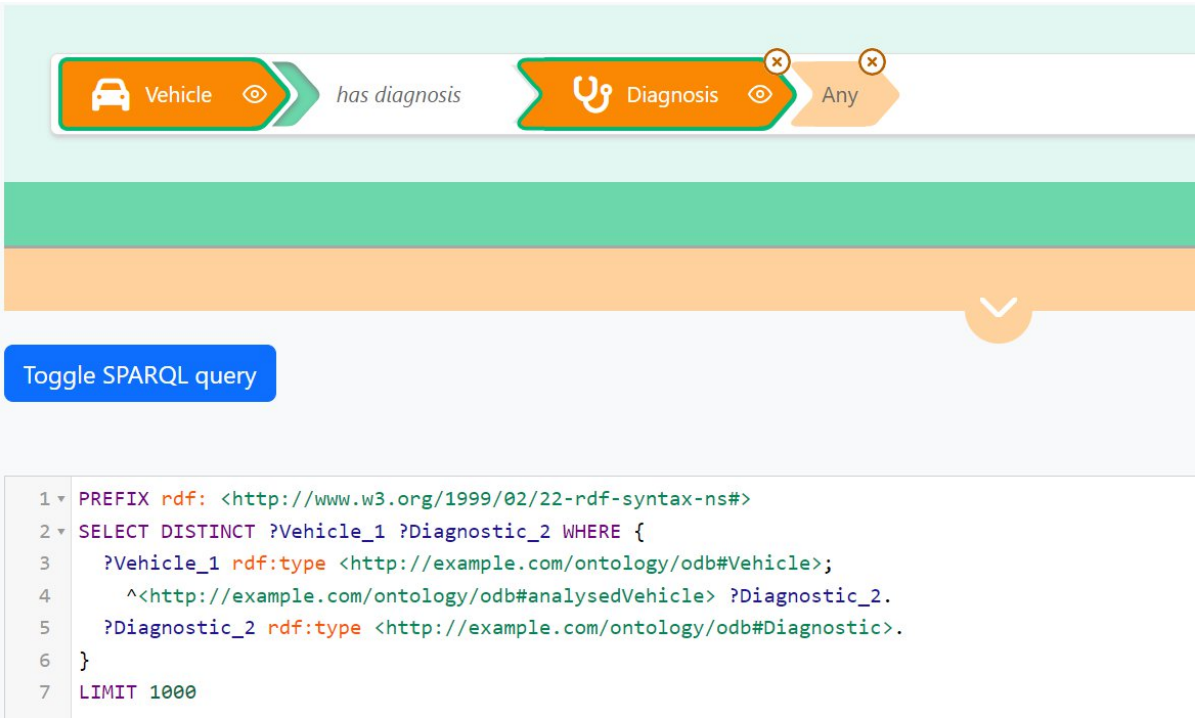
You will do this with an *"SHACL inverse path"*, which is a pair of brackets "**[ ]"**, containing the property sh:inversePath, followed by the identifier of the property to traverse in the inverse direction. In our example this would be "[sh:inversePath ex:is_part_of]". This means *"follow the is_part_of property in the inverse direction"*. This will be translated into the SPARQL property path ^ex:is_part_of .

Example

In cars ontology, starting from the Vehicle, searching for a Diagnosis isn't possible if we refer to the diagram : the property odb:analysedVehicle goes from Diagnostic —to—> Vehicle indeed. Here we create the "this:hasDiagnosis" property, that goes from Vehicle —to—> Diagnostic, and corresponds to the sh:path [sh:inversePath odb:analysedVehicle] (inverse of odb:analysedVehicle).

URI	sh:path	^sh:property(separator=",")	sh:node	
Vehicle				
this:Vehicle_VIN	odb:VIN	this:Vehicle		
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	this:Manufacturer	
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	this:Vehicle	this:Diagnostic	
this:Vehicle_weightInKg	odb:weightInKg	this:Vehicle		

The property now appears in the query builder note the caret "^" in the SPARQL query) :



Query multiple properties in a single criteria

This is to be used if you would like the user to query more than one property at a time. This can be useful if you would like to provide a search field (core:SearchProperty) that will search in label + description. This can also be used if two classes are connected by more than one possible property and you want to search all of them, as *"Person is friend with Person"* and *"Person is a colleague of Person"*; you may want to provide your user with *"Person knows Person"*, and "knows" would search for both "is friend with" and "is colleague of".

You will do this with an *"SHACL alternative path"*, which is a pair of brackets "**[ ]"**, containing the property `sh:alternativePath`, followed by a parenthesis **( )** containing the list of property identifiers to join. In our example this would be "[sh:alternativePath (ex:is_friend_of ex:is_colleague_of)]". This means *"follow either the is_friend_of or is_colleague_of properties"*. This will be translated into the SPARQL property path ex:is_friend_of|ex :is_colleague_of.

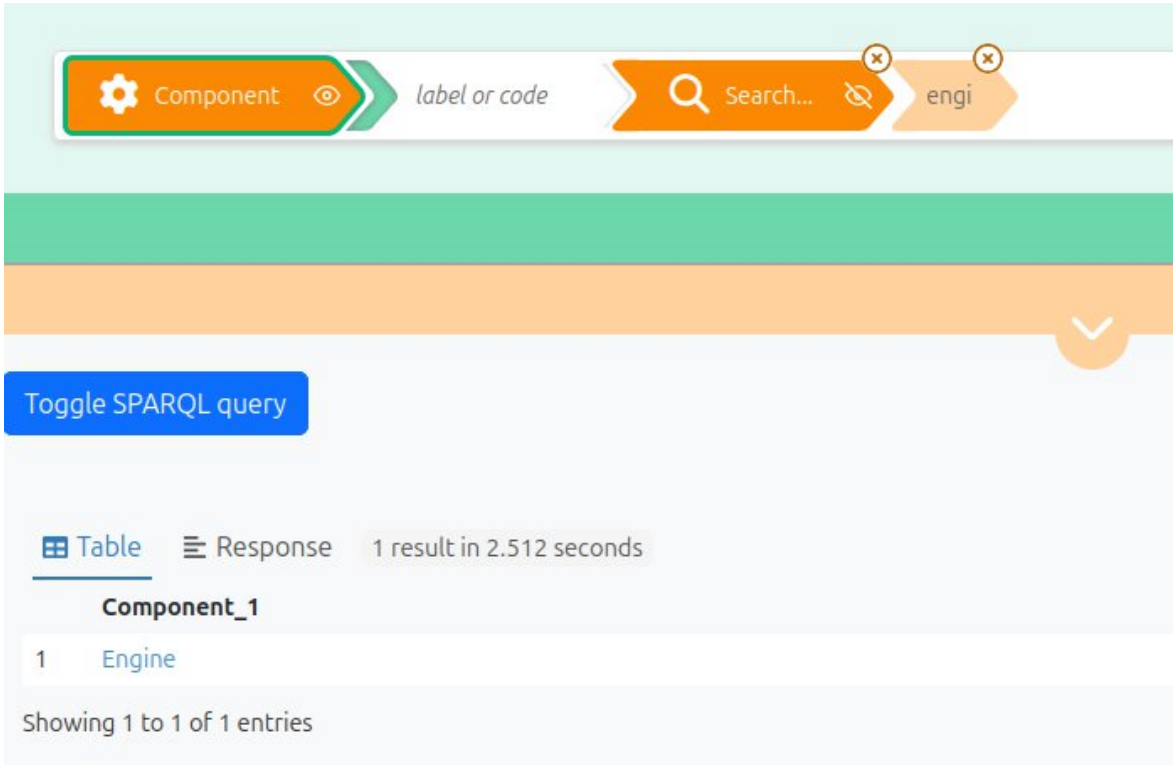
Example

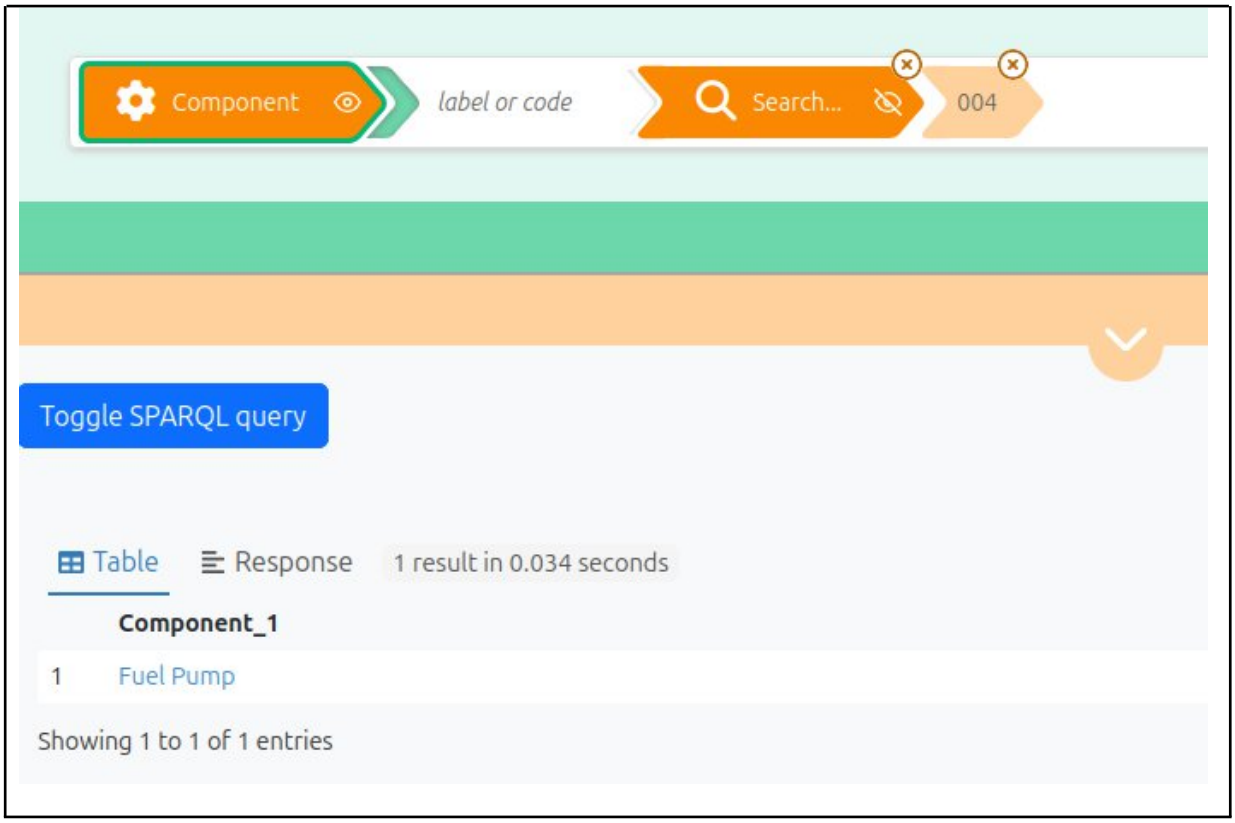
To illustrate this on the Component entity, we decided to query both label and component code in one unique field: you can see the new property this:Component_label_or_code has

been created therefore with the special SHACL property path [sh:alternativePath (odb:componentCode rdfs:label)] to combine both properties in a single one :

URI	sh:path	^sh:property(separator=",")	sh:node	dash:searchWidget
Component				
this:Component_componentCode	odb:componentCode	this:Component		core:SearchProperty
this:Component_label	rdfs:label	this:Component		core:SearchProperty
this:Component_label_or_code	[sh:alternativePath (odb:componentCode rdfs:label)]	this:Component	this:Search	core:SearchProperty

We can see in the two following screenshot that a search for either a label (“engi”) or a code (“004”) of component will work and yield a result:





Query a property recursively

This is a less frequent use-case and it is used in combination with a tree property (`core:TreeProperty`). This is useful when you would like the user to query recursively and transparently into a complete "branch" of entities related with a hierarchical link (typically `skos:broader` or `dcterms:isPartOf`).
Most of the time, when you provide a tree widget, the implicit expectation from the user is that when she selects a node in the tree, then the query would also search for all children of that node.
For example if you have *"Place is part of Place"* in your graph, with places organized as a tree, if the user searches for "Restaurant located in Paris", then she would expect to receive restaurants also located in places that are part of Paris, such as "17eme arrondissement".

You will do this with a combination of "sequence path" (SHACL property path using parenthesis), containing inside the parenthesis, in the second position of the sequence, a *"SHACL zero or more path"*, which is a bracket **[ ]** containing the identifier `sh:zeroOrMorePath`, followed by the URI of the property to traverse recursively. In our example this would be "(ex:is_located_in [sh:zeroOrMorePath ex:is_part_of])". This means: *"follow the is_located_in property, then follow the is_part_of property recursively (until you reach the selected node, which in our example would be Paris)"*; In other words *"select all restaurants with a is_located_in property that points to a place that is linked to Paris with any number of is_part_of properties"*.

Combine property paths

It is possible to combine property paths together, in other words to use in combination sequence paths “(p1 p2)”, inverse paths “[sh:inversePath p1]”, alternative paths “[sh:alternativePath p1]”, and zero-or-more paths “[sh:zeroOrMorePath p1]”. A typical use-case is to combine inverse path with a sequence path to traverse properties in the inverse direction in a sequence path.

Example

In our “Car” ontology we could imagine a direct link between a “Vehicle” and the “Error Code” that were diagnosed on this Vehicle, which would give the property path ([sh:inversePath odb:analysedVehicle] odb:hasResult odb:hasErrorCode)

Map classes to the underlying knowledge graph

Query a subset of a class

Imagine your knowledge graph contains very broad classes, such as “Document” but you would like to present more specific entities to your users, such as “Report”, “Article” or “News item”, based on a “type” property of the Document instances. You might also use SKOS Concepts, organized in different Concept Schemes, but you would like to present them as different entities than simply “Concept”.

You will do this by specifying a custom target for the corresponding entities, instead of using sh:targetClass. Declare an entity as usual, with a new line in the Entities tab, with the following differences:

1. Do not specify anything in the sh:targetClass column
2. Enter a URI identifier in the sh:target column, which will be the same URI as the entity, followed by “-target” (e.g. If the entity has the URI this:Person the sh:target will be this:Person-target).
3. Enter the SPARQL query defining the target of your entity in the sh:select column. The SPARQL query MUST follow these constraints:
    a. It must not use prefixes. All property and classes identifiers have to be written as full URIs using brackets <http://...>
    b. It must select and return the variable “**$this**” – yes, with a dollar sign, not a question mark (this is legal SPARQL).
    c. It must not select any other variables.
    d. All other variables in the query must start with a question mark (e.g. “?parent”)

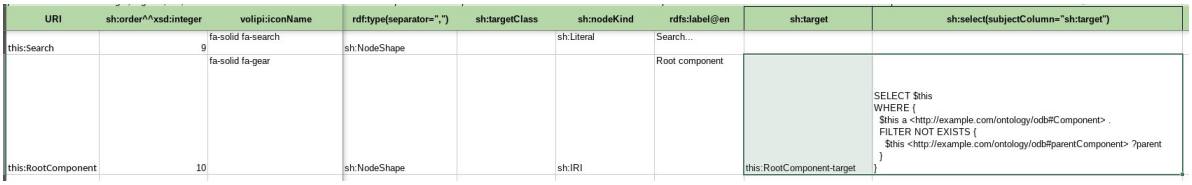
Example

We are adding an new entity in our sample configuration ontology to represent “root components”, that is components that are at the top of the component hierarchy. We do that because only root components can be associated with a criticity level, not other components.

We define root components as “components without a parent”, hence we associate them with the following SPARQL query, following the guidelines above (no prefixes, and using $this) :

```
SELECT $this
WHERE {
 $this a <http://example.com/ontology/odb#Component> .
 FILTER NOT EXISTS {
  $this <http://example.com/ontology/odb#parentComponent> ?parent
 }
}
```

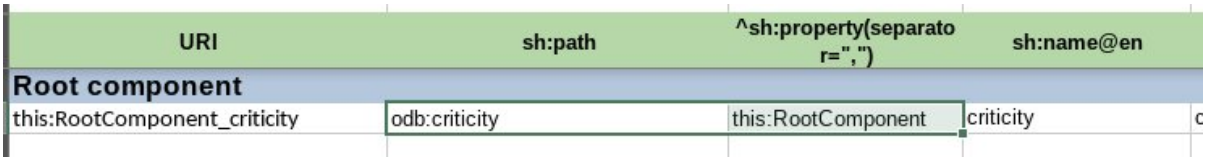
Which gives us :



URI	sh:order^^xsd:integer	volipi:iconName	rdf:type(separator=",")	sh:targetClass	sh:nodeKind	rdfs:label@en	sh:target	sh:select(subjectColumn="sh:target")
this:Search	9	fa-solid fa-search	sh:NodeShape		sh:Literal	Search...		

this:RootComponent	10	fa-solid fa-gear	sh:NodeShape		sh:IRI	Root component	this:RootComponent-target	SELECT $this
								WHERE {
								$this a <http://example.com/ontology/odb#Component> .
								FILTER NOT EXISTS {
								$this <http://example.com/ontology/odb#parentComponent> ?parent
								}
								}

We then use this entity as the domain of the odb:criticity property in the property tab, just like any other property:



URI	sh:path	^sh:property(separator=",")	sh:name@en	
Root component				
this:RootComponent_criticity	odb:criticity	this:RootComponent	criticity	c

And we can see in the UI that :

- Root component appears as an entity in the first list
- It has the criticity property attached
- The generated SPARQL query includes our SPARQL query to select only components that do not have any parent:

Query more than one class

This is a less frequent use-case. It can be useful if your graph has specific classes, but you want to show more generic entries to your users.
For example if you have the classes “Person” and “Company”, but you want to show to the user a single entry like “Actors”, encompassing both persons and companies.

As with the previous situation (“Querying a subset of a class”), you will do this by specifying an an entity associated to a custom target in sh:target + sh:select, instead of using sh:target-Class. The target is defined with a SPARQL query that will select instances of multiple classes instead of a single one.

Entities without any mapping / targets

There are situation where you need to declare an entity in your configuration, that is indeed an IRI, but for which there are no actual instances in your graph. This happens for example if you have alignments or links to external IRIs, such as Wikidata IRI, but your graph does not contain any rdf:type statement on those IRI. You don’t want an rdf:type criteria to be inserted in the final SPARQL query.

In that case, the solution is simple : do not specify any target for your entity, neither using sh:targetClass, neither using sh:target. No rdf:type query criteria will be inserted for this kind of entity in the final SPARQL query string.

Create a multilingual configuration

Multilingual labels and tooltips

You can create a multilingual configuration if your website enable your users to select their preferred language. Sparnatural is multilingual by nature and can display the labels and tooltips from its configuration in multiple languages, if they are provided in the configuration. The "<spar-natural>" HTML element contains a "lang" attribute that indicates which language should be used to select the labels and tooltips to display. That attribute can be adjusted by a control in the HTML page (out of scope of Sparnatural and of this documentation), typically a language-selection dropdown.

If you want to provide your users with a multilingual configuration you have to add additional columns in your configuration files:

- In the "Entities" tab:
  - add more "rdfs:label@xx" columns and adjust the language tag in the header to populate the labels of classes in different languages
  - add more "sh:description@xx" columns and adjust the language tag in the header to populate the tooltips of classes in different languages
- In the "Properties" tab, duplicate the same columns "sh:name@xx" and "sh:description@xx" for the labels and tooltips of the properties.

Advanced note: Sparnatural is also configured with a "defaultLang" parameter. This default language is the language in which the knowledge graph is supposed to always have a label for all Entities. This is meant to deal with situations where some Entities do have a label in the user preferred language, and others don't, but will have a label in the default language. The default label can be returned to display a label to the user. Read the HTML attributes reference documentation for more information.

Example

Entities and properties labels and tooltips can be translated in as many languages as wished just by adding the translations in an "@xx" column for each : here the entities tab, translated in French, `rdfs:label@fr` and `sh:description@fr` :

URI	rdfs:label@en	rdfs:label@fr	sh:description@en	sh:description@fr
this:Vehicle	Vehicle	Véhicule	A vehicle is a car model for a specific brand.	Un véhicule est un modèle de voiture pour une marque spécifique.
this:Manufacturer	Manufacturer	Constructeur	A car manufacturer is a company whose main activity is the design, construction and marketing of cars.	Un constructeur automobile est une entrepris qui a pour activité principale la conception, la construction et la commercialisation de voitures.
this:Diagnostic	Diagnosis	Diagnostic	A diagnosis identifies a possible problem on your vehicle. You can request a diagnosis when you suspect a breakdown or malfunction. Using an auto diagnosis kit, the mechanic identifies the problem with your vehicle.	Un diagnostic permet d'identifier un éventuel problème sur votre véhicule. Vous pouvez demander un diagnostic quand vous suspectez une panne ou un dysfonctionnement. À l'aide d'une valise de diagnostic auto, le mécanicien identifie le problème subi par votre véhicule.
this:Error	Error	Erreur	An error is an element that comes up during a diagnosis, which indicates that the vehicle on which the analysis was carried out is encountering a problem.	Une erreur est un élément qui remonte lors d'un diagnostic, qui indique que le véhicule sur lequel on a fait l'analyse rencontre un problème.
this:ErrorCode	Error code	Code d'erreur	An error code is a set of numbers following a letter corresponding to a problem detected on your vehicle. The letter gives an indication of the family of the defect.	Un code erreur, est un ensemble de chiffres suivant une lettre correspondant à un problème détecté sur votre véhicule. La lettre donne une indication sur la famille du défaut.
this:Symptom	Symptom	Symptôme	A symptom is a phenomenon, perceptible or observable character linked to a state, a problem that it allows to detect, of which it is the sign.	Un symptôme est un Phénomène, caractère perceptible ou observable lié à un état, un problème qu'il permet de déceler, dont il est l signe.
	Component	Composant	A class representing a component of a	Une classe représentant un composant d'un

here the properties tab, with `sh:name@fr` and `sh:description@fr` :

URI	sh:path	sh:name@en	sh:name@fr	sh:description@en	sh:description@fr
Vehicle					
this:Vehicle_VIN	odb:VIN	has VIN	a pour VIN	Specifies the Vehicle Identification Number (VIN) of the vehicle.	Spécifie le numéro d'identification du véhicule (VIN).
this:Vehicle_hasManufacturer	odb:hasManufacturer	has manufacturer	a pour constructe	Specifies the manufacturer of the vehicle.	Spécifie le constructeur d'un véhicule.
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	has diagnosis	a pour diagnostic	The property is the inverse of odb:analysedVehicle.	Propriété inverse de odb:analysedVehicle.
this:Vehicle_weightInKg	odb:weightInKg	weight in kg	poids en kg		
Manufacturer					
this:Manufacturer_name	odb:name	has name	nom	Specifies the name of the manufacturer.	Spécifie le nom du constructeur.
Diagnosis					
this:Diagnostic_diagnosticDate	odb:diagnosticDate	has diagnosis date	date du diagnosti	Defines the date on which the diagnosis occurs.	Définit la date à laquelle le diagnostic a eu lieu.
this:Diagnostic_analysedVehicle	odb:analysedVehicle	analysed vehicle	véhicule analysé	Specifies that the vehicle has been analyzed, to identify a potential problem.	Spécifie que le véhicule a été analysé, pour identifier un potentiel problème.
this:Diagnostic_hasResults	odb:hasResults	has results	a pour résultat	Specifies the results, from the analysis.	Spécifie les résultats issus de l'analyse.
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	has diagnosis place	lieu du diagnostic	Defines the place where the diagnosis occurs.	Définit le lieu où le diagnostic a été effectué.
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	returns code	renvoie le code	The property is a shortcut between Diagnosis and Error Code.	Cette propriété est un raccourci entre Diagnostic et Code d'erreur.

This makes it possible to have a Sparnatural interface in French, by adjusting the “lang” attribute of the <spar-natural> element in the HTML page to “fr”:

Multilingual default label properties

An earlier section introduced the `dash:LabelRole` flag to flag one entity as the default label for an entity. By default, when fetching the value of the default label property, Sparnatural will not apply any language filter; so multiple values will be retrieved in case the label property holds

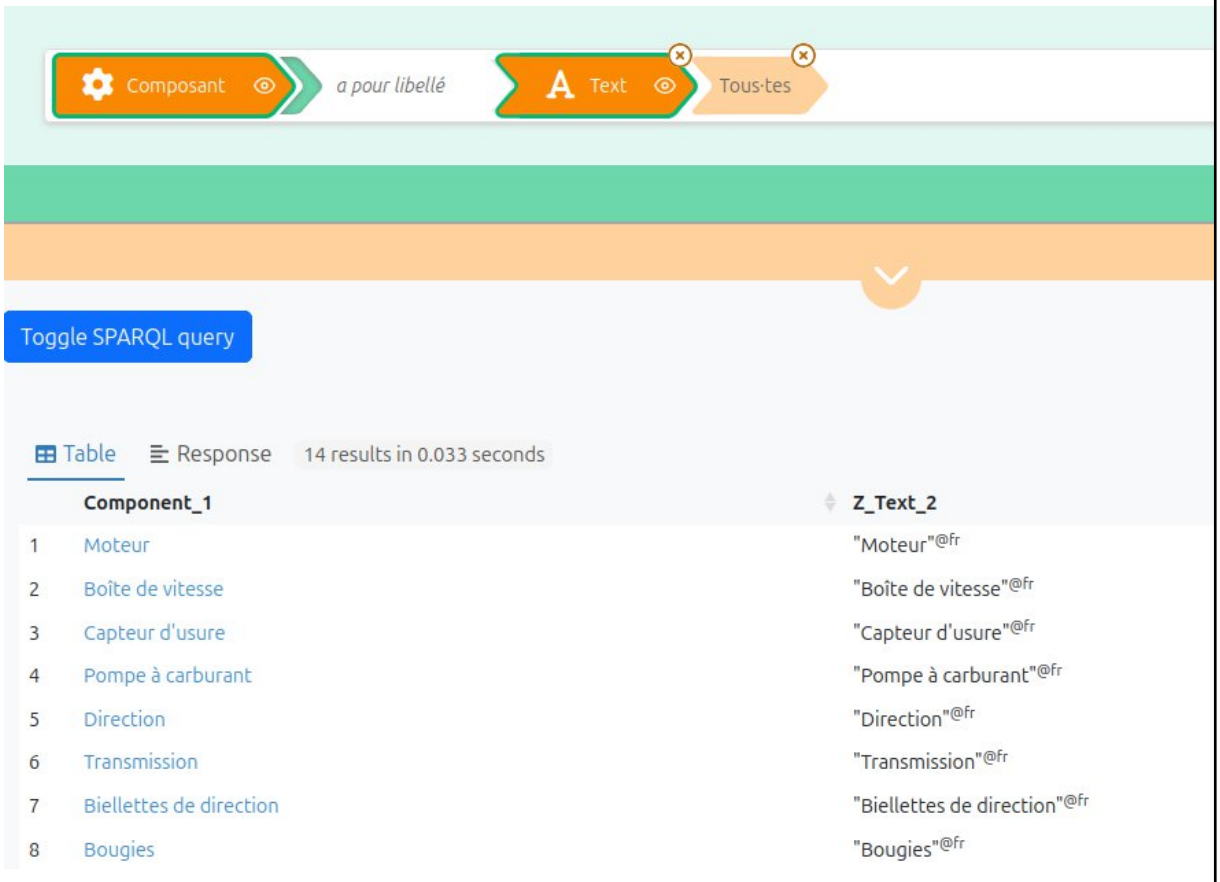
multilingual values. In order to instruct Sparnatural to retrieve the default label property only in the current user language, its datatype must be set to rdf:langString and not xsd:string.

Example

In the example data of the cars ontology, labels of components are multilingual, e.g. "Engine"@en and "Moteur"@fr. In order to indicate to Sparnatural that only the label in the current user language should be retrieved, we indicated that the datatype is rdf:langString column:

URI	sh:path	sh:name@en	sh:nodeKind	sh:datatype	dash:property Role
Component					
this:Component_componentCode	odb:componentCode	has component code	sh:Literal	xsd:string	
this:Component_label	rdfs:label	label	sh:Literal	rdf:langString	dash:LabelRole
this:Component_label_or_code	[sh:alternativePath (odb:componentCode rdfs:label)]	label or code	sh:Literal		

We can see that only French labels are retrieved in the result table, when Sparnatural is set to French:



Create a hierarchical configuration

Sparnatural since version 10 supports hierarchies of entities. You may need to show your users not a flat list of entities but a hierarchy of entities organised in a tree. Users are able to navigate up or down the tree to select the entity that is the subject or the object of the criteria that they build.

There are two different ways to express a hierarchical information between entities : “ontological hierarchy” and “contextual hierarchy”, which we explain below.

Ontological hierarchy

An “ontological hierarchy” is a hierarchy between the classes of your ontology. This hierarchical information is true in “every possible world” in which your ontology is used, and of course the corresponding parents and children classes are declared in your ontology.

This information is captured by rdfs:subClassOf triples between the classes of your ontology (themselves being the values of sh:targetClass in your Sparnatural configuration).

Tip: The configuration spreadsheet does include a column rdfs:subClassOf so you can create these ontological relationships directly in your configuration. However this relation is typically already in your OWL ontology. You can pass to Sparnatural both your SHACL specification *and* your OWL ontology in its “src” attribute by separating them with a whitespace. See the integration documentation for more details.

Contextual hierarchy

A “contextual hierarchy” is a hierarchy between the entities (node shapes) of your Sparnatural configuration. This hierarchical information is valid only in your configuration, and not in every possible world. It relates node shapes in your configuration, which, as it was described in a previous section, may not correspond one-one to classes in your ontology, if they are mapped to more complex SPARQL targets.

This hierarchical information is captured by the sh:node column in the entities tab.

Example

We introduced earlier the entity “root components”, that is components that are at the top of the component hierarchy. We associated this entity to a SPARQL query that defines which nodes it targets in the underlying knowledge graph. That selection criteria will be used in the finel SPARQL query being generated.

However we didn’t relate this entity to the “Component” entity, but it makes sense to do so : every “root component” is a “component” and can inherit from all properties attached at the component level.

We create this relation by specifying this:Component as the parent entity of this:RootCom-

ponent in the sh:node column.

URI of the entities. This column will use the "this" prefix declared in the prefixes tab.	*The sort order of the entity in the class dropdown list. This is an integer, e.g. "1", "2", etc.*	*This column is used to indicate the parent class on the class from the ontology. This is "ontological" hierarchy. Use this column only in advanced cases where there is some inheritance of the properties between broader classes and more generic ones.*	*This column is used to indicate the parent entity in the configuration. This is "contextual" hierarchy. Use this column only in advanced cases where there is some inheritance of the properties between broader classes and more generic ones.*
URI	**sh:order^^xsd:integer**	**rdfs:subClassOf(subjectColumn="sh:targetClass")**	**sh:node**
this:Component	7		
this:TrueFalse	8		
this:Search	9		
this:RootComponent	10		this:Component

And we can see in the UI that Root Component is placed under “Component” and inherits from all the properties attached to “Component”, such as the search on a label or a code:

Display labels in the result table

By default, when selecting the entities to be displayed in the result table (through SPARQL variables) Sparnatural will simply select URIs, and URIs will be displayed in the result table. This is not user-friendly, as users expect to see some kind of human-readable label in the result table. Sparnatural can be instructed to fetch an extra SPARQL variable containing the human-readable label of a selected entity, when there is one.

An earlier section of this documentation introduced the notion of a “default label property” for an entity. A default label property is a property marked with the value dash:LabelRole in the dash:propertyRole column. This is how you can tell Sparnatural to fetch this extra human-readable label.

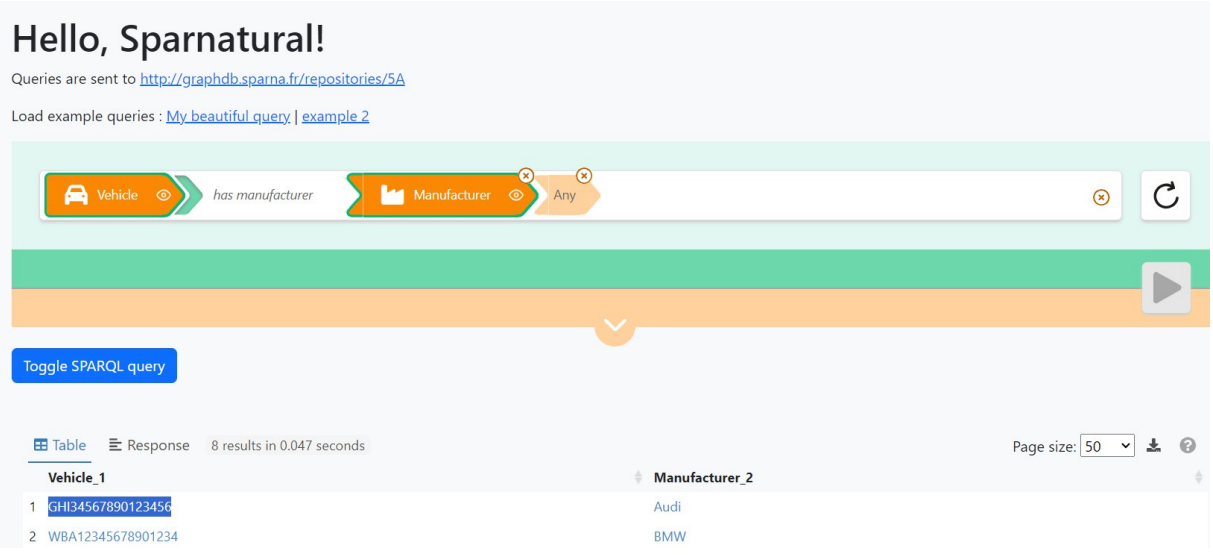
Example

VIN are vehicles identifiers. As such, and without other human-readable labelling property or vehicle, the `odb:VIN` property is marked as the default label property for Vehicle.
Similarly, on Manufacturer, the `odb:name` property (being the sole property of this entity !) is marked as the default label property of Manufacturers.

(here with a few hidden columns for readability).

URI	sh:path	^sh:property(separator=",")	sh:name@en	dash:searchWidget	dash:propertyRole
Vehicle					
this:Vehicle_VIN	odb:VIN	this:Vehicle	has VIN	core:AutocompleteProperty	dash:LabelRole
this:Vehicle_hasManufacturer	odb:hasManufacturer	this:Vehicle	has manufacturer	core:ListProperty	
this:Vehicle_hasDiagnosis	[sh:inversePath odb:analysedVehicle]	this:Vehicle	has diagnosis	core:NonSelectableProperty	
this:Vehicle_weightInKg	odb:weightInKg	this:Vehicle	weight in kg	core:NumberProperty	
Manufacturer					
this:Manufacturer_name	odb:name	this:Manufacturer	has name	core:NonSelectableProperty	dash:LabelRole
Diagnosis					
this:Diagnostic_diagnosticDate	odb:diagnosticDate	this:Diagnostic	has diagnosis date	core:TimeProperty-Date	
this:Diagnostic_analysedVehicle	odb:analysedVehicle	this:Diagnostic	analysed vehicle	core:AutocompleteProperty	
this:Diagnostic_hasResults	odb:hasResults	this:Diagnostic	has results	core:NonSelectableProperty	
this:Diagnostic_diagnosticPlace	odb:diagnosticPlace	this:Diagnostic	has diagnosis place	core:MapProperty	
this:Diagnostic_returnsCode	(odb:hasResults odb:hasErrorCode)	this:Diagnostic	returns code	core:ListProperty	

When triggering a query, the result is the following : we can see in the result table that the vehicle VIN number and the Manufacturer name are displayed for the corresponding selected entities.



Advanced note: What really happens is that the query result set contains the “Vehicle_1” (containing the URI) and “Vehicle_1_label” (containing the name) columns, and also the “Manufacturer_2” and “Manufacturer_2_label” columns. You can see this raw query result set if you click on the download button.

Then the query result table component actually merges them in a single URI+label column. This is done by the Sparnatural TableX YasGUI plugin. You should refer to the technical documentation for the integration of this plugin.

Advanced note: you can mark the default label property as optional, with core:enableOptional. Sparnatural will honour this by wrapping the default label property variable in an OPTIONAL clause. This will populate the xxxx_label in the query only when it is known (as opposed to not returning the row if the property is missing on an item).

Advanced configuration

Advanced configuration : create custom SPARQL datasources

Creating a custom datasource to populate a list property or an autocomplete property is possible by providing your custom SPARQL query. To do this you need to be proficient with SPARQL. This allows you to concatenate 2 properties as the label in the dropdown list or to customise the way autocomplete proposals are proposed.

To create your custom datasource, go to the “Datasources” tab of the configuration file, and:

- Add a line, with your datasource URI in column A, in the “this:” namespace
- in column rdf:type, set the value `datasources:SparqlDatasource`
- in column datasources:queryString, enter the SPARQL query, including all its prefixes.
- then you can refer to your datasource from the “datasources:datasource” column of the “Properties” tab.

The datasources documentation explains the rules you need to follow to create your own SPARQL datasource. Please refer to this documentation for details. To sum it up, your query:

- MUST return 2 variables `?uri` and `?label`
- can include special variables that will be passed by Sparnatural before the query is sent, such as:
    - `$domain` with the class selected at the beginning of the criteria
    - `$range` with the class selected at the end
    - `$property` with the property selected
    - `$lang` with current user language,
    - `$key` with the key being searched by the user in autocomplete fields,
    - etc.

  You don’t *have to* use all of them.

If you don’t see any results in your dropdown list populated with a custom query, refer to the next section to know how to debug the query.

Example

Here we propose to set a custom datasource for `odb:hasComponent` property. Let's imagine it would be created using a concatenation of component code + component label. To do so we first write the SPARQL query that will be sent to the system to get the info, then we can embed it in a new "this" datasource (tab "Datasources" of Sparnatural config sheet) :

URI	rdf:type	datasources:queryString

this:list_componentCode_alpha	datasources:SparqlDatasource	PREFIX odb: <http://example.com/ontology/odb#>
		SELECT DISTINCT ?uri ?label
		WHERE {
		?domain $type $domain .
		?domain $property ?uri .
		# Note how the range criteria is not used in this query
		FILTER(isIRI(?uri))
		?uri rdfs:label ?libelleComposant .
		FILTER(lang(?libelleComposant) = "" || lang(?libelleComposant) = $lang)
		?uri odb:componentCode ?codeComposant .
		# Concat component code + component label
		BIND(CONCAT(STR(?codeComposant)," - ",STR(?libelleComposant)) AS ?label)
		}
		ORDER BY UCASE(?label)
		LIMIT 500

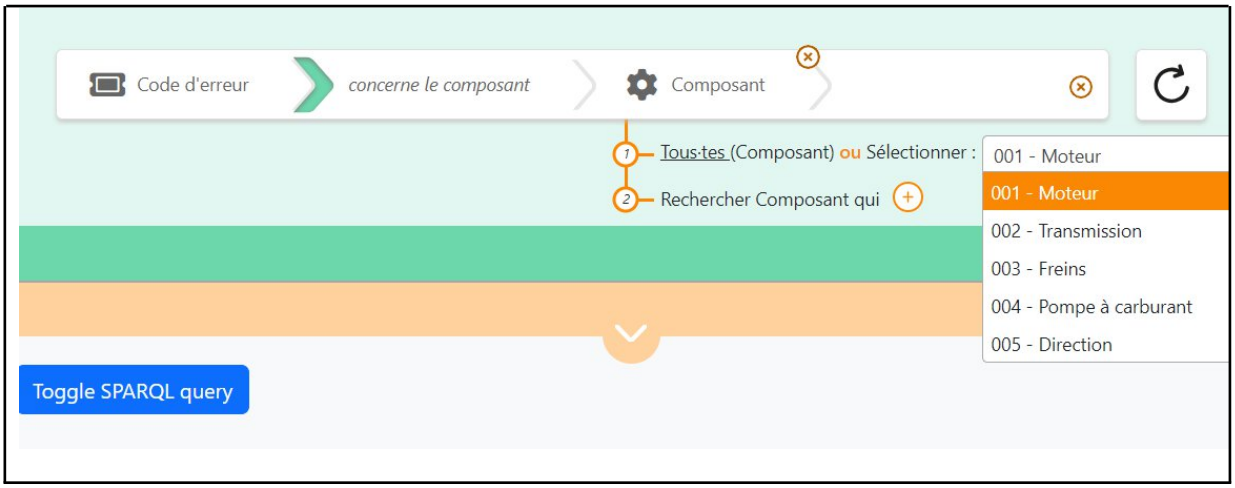
The details of the SPARQL query is beyond the scope of this documentation, please simply note that

a) it is using "magic variables" `$domain`, `$property`, `$lang` that are replaced at runtime by Sparnatural with the corresponding values in the criteria being built (see the Sparnatural datasource documentation)
b) the `BIND(CONCAT(...) AS ?label)` line that is doing the actual concatenation of the code with the name, which is returned in the result set.

Next step is to modify the property's datasource itself with the URI of the new datasource :

URI	sh:path	sh:name@en	datasources:datasource
ErrorCode			
this:ErrorCode_errorCode	odb:errorCode		
this:ErrorCode_hasSymptom	odb:hasSymptom	has symptom	datasources:list_rdfslabel_count
this:ErrorCode_hasComponent	odb:hasComponent	has component (list)	this:list_componentCode_alpha
this:ErrorCode_hasComponent_tree	odb:hasComponent	has component (tree)	

Then testing the query in the query builder to check that the query works well :

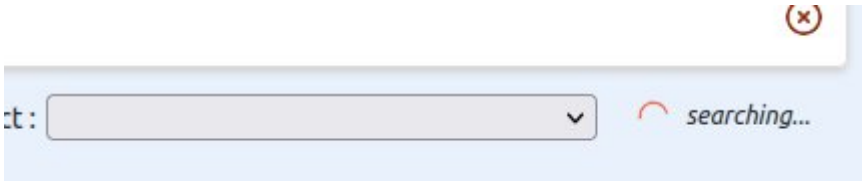


Advanced configuration : debug custom datasources

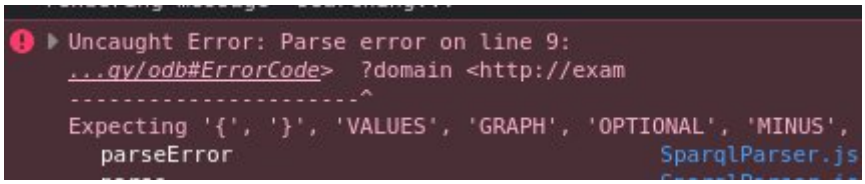
Most of the time a custom datasource query will not work the first time and a little debugging is necessary. There are three main reasons a custom datasource is not working:

Case 1 : the SPARQL query is syntactically wrong

UI Symptom : the loader keeps running, the list is not populated.



Console Symptom : Check in your console to see if there is a SPARQL parsing error message, like so:

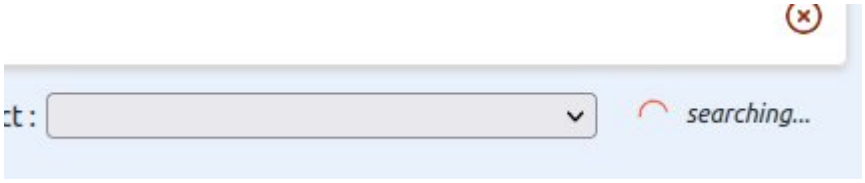


(in our case here, a missing dot in the SPARQL).

How to fix it : fix your SPARQL query, make sure you edit it in a tool with syntax checking.

Case 2 : The query to the endpoint failed (the server is unreachable, or there is a CORS issue, etc.)

UI Symptom : the loader keeps running, the list is not populated.



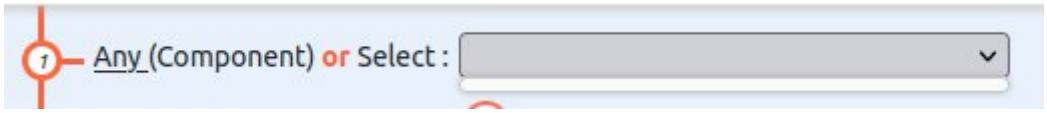
Console Symptom : you will see a network query failing in the network console:

(in our case here, we simulated a CORS issue).

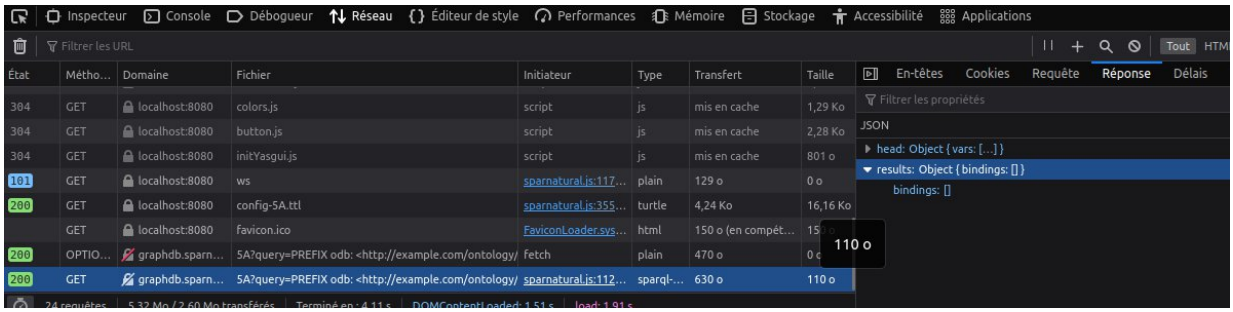
How to fix it : check more in detail why the network call failed. This could be for a security reason, a CORS reason, or another reason on the server that would return an HTTP 500 error.

Case 3 : The SPARQL query is syntactically correct and was successfully executed, but returned no results.

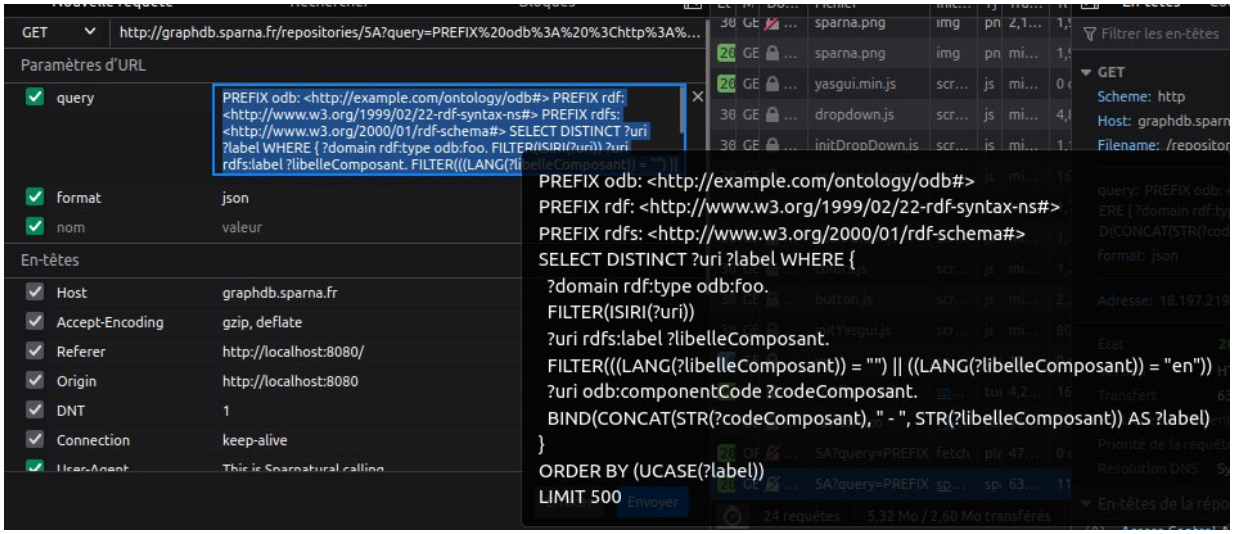
UI Symptom : the loader stops, the list is empty



Console Symptom : you will see the SPARQL HTTP request to populate the list was sent and was successful, but has returned no “bindings” in its response



How to fix it : You must understand why the query does not return the expected result. To do that you need to fetch it from the HTTP request in the console:



Copy the query, paste it in your triplestore SPARQL interface, and work on it to understand why it does not return the expected results.

Warning : remember that this is the final query being sent, after all “magic variables” have been replaced by Sparnatural with their final values. Please refer to the datasource documentation for explanations on these variables. When you understand why the query does not work, remember to replace all fixed variables back with their magic variable name (e.g. $domain, $lang, etc.)

Advanced configuration : setup tree widget datasource

A tree widget requires two datasources : one to get the root nodes of the tree, and one to get the children of a node that is unfolded. This is set with the datasources:treeRootsDatasource and datasources:treeChildrenDatasource columns respectively, in the “Properties” tab. These two columns are useful only when the property is a core:TreeProperty, you can ignore them otherwise. The datasource documentation gives the details of the existing default tree datasources and how to create a new tree widget datasource. Please refer to this documentation for details.

Example

In Sparnatural car configuration, the class odb:ErrorCode has a property odb:hasComponent, which refer to car components that re structured in a hierarchized manner. Therefore we can set this property as a core:TreeProperty with two custom tree datasources, one identified with this:tree_root_Component and one identified with this:tree_children_Component, which serve respectively to fetch the roots and the children of a node.

4	URI	rdf:type	datasources:queryString

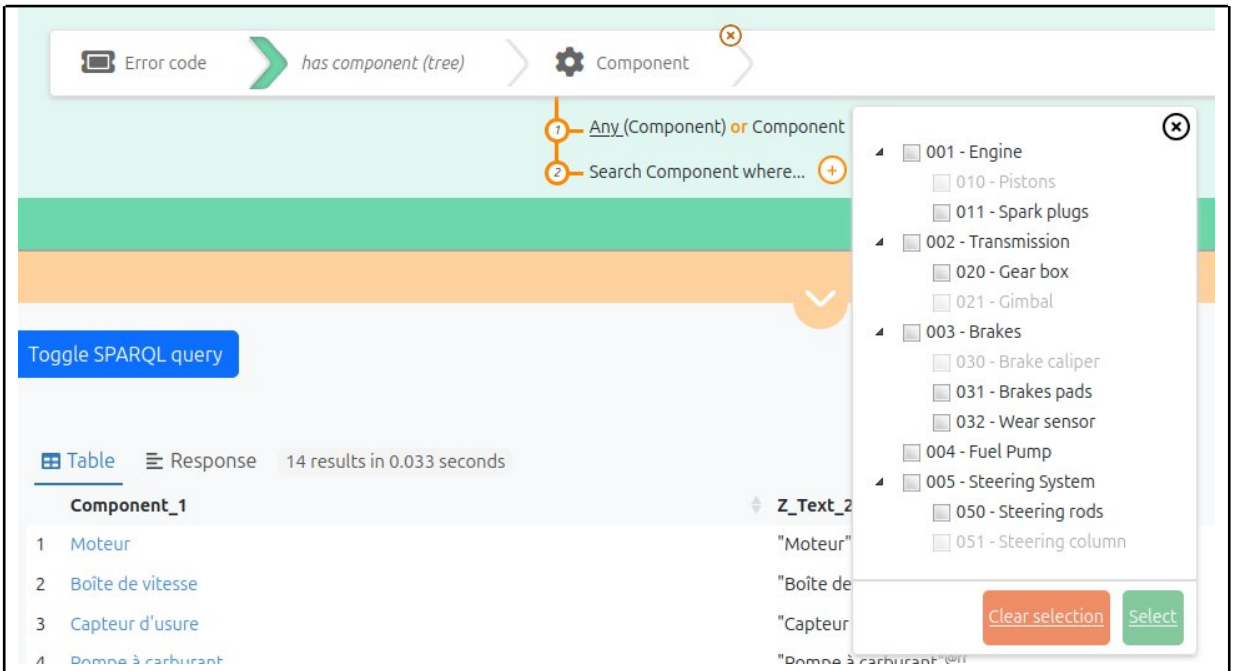
10	this:tree_root_Component	datasources:SparqlDatasource	PREFIX odb: <http://example.com/ontology/odb#>
			PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#label>
			SELECT ?uri ?label ?hasChildren (COUNT(?x) AS ?count) WHERE {
			?uri a odb:Component .
			# Keep only roots, that do not have any parent
			FILTER NOT EXISTS {
			?uri odb:parentComponent ?parent .
			}
			?uri rdfs:label ?libelleComposant .
			FILTER(lang(?libelleComposant) = "" || lang(?libelleComposant) = $lang)
			?uri odb:componentCode ?codeComposant .
			# Concat component code + component label
			BIND(CONCAT(STR(?codeComposant)," - ",STR(?libelleComposant)) AS ?label)
			OPTIONAL { ?uri ^odb:parentComponent ?children }
			BIND(IF(bound(?children),true,false) AS ?hasChildren)
			OPTIONAL {
			?x a $domain .
			?x $property ?uri .
			}
			}
			GROUP BY ?uri ?label ?hasChildren
			ORDER BY ?label

11	this:tree_children_Component	datasources:SparqlDatasource	PREFIX odb: <http://example.com/ontology/odb#>
			PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#label>
			SELECT DISTINCT ?uri ?label ?hasChildren (COUNT(?x) AS ?count) WHERE {
			
			$node ^odb:parentComponent ?uri .
			
			?uri rdfs:label ?libelleComposant .
			FILTER(lang(?libelleComposant) = "" || lang(?libelleComposant) = $lang)
			?uri odb:componentCode ?codeComposant .
			# Concat component code + component label
			BIND(CONCAT(STR(?codeComposant)," - ",STR(?libelleComposant)) AS ?label)
			
			OPTIONAL { ?uri ^odb:parentComponent ?children }
			BIND(IF(bound(?children),true,false) AS ?hasChildren)
			
			OPTIONAL {
			?x a $domain .
			?x $property ?uri .
			}
			}
			GROUP BY ?uri ?label ?hasChildren
			ORDER BY ?label

Selecting the core:TreeProperty widget from properties tab, these two datasources are then referred to like so :

URI	sh:path	sh:name@en	datasources:datasource	datasources:treeRootsDatasource	datasources:treeChildrenDatasource
ErrorCode					
this:ErrorCode_errorCode	odb:errorCode				
this:ErrorCode_hasSymptom	odb:hasSymptom	has symptom	datasources:list_rdfslabel_count		
this:ErrorCode_hasComponent	odb:hasComponent	has component (list)	this:list_componentCode_alpha		
this:ErrorCode_hasComponent_tree	odb:hasComponent	has component (tree)		this:tree_root_Component	this:tree_children_Component

This way the corresponding tree is displayed in the query builder :



Note how

a) some items in the component tree are greyed out because no error codes affect them.
b) some items in the component tree cannot be unfolded as they have no children.

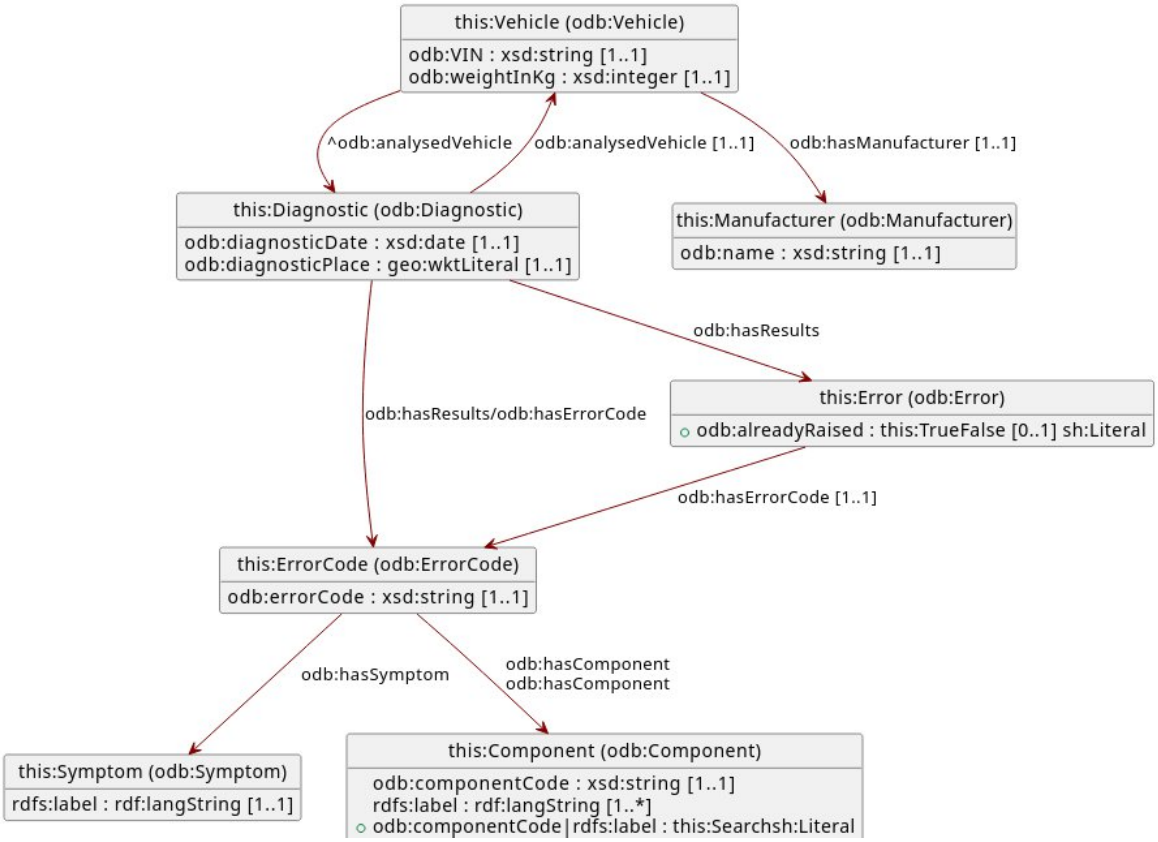
Those two informations (the fact that a node has children and the fact it is not referenced as a value) are computed by the SPARQL queries used as datasources, respectively in the `?hasChildren` variable and the `?count` variable.

Annex : View the Sparnatural configuration in SHACL Play

Since the Sparnatural configuration is based on SHACL with a few additional non-SHACL annotations, it is compatible with other SHACL tools. In particular, Sparna maintains SHACL Play! At https://shacl-play.sparna.fr . This tool, amongst other features, allows to generate a human-readable documentation from a SHACL specification.

For example if we upload the sample car configuration in SHACL in https://shacl-play.sparna.fr/play/doc, we can get a nice diagram :

Diagrams



And also a list of documentation tables of each of the entities in our model:

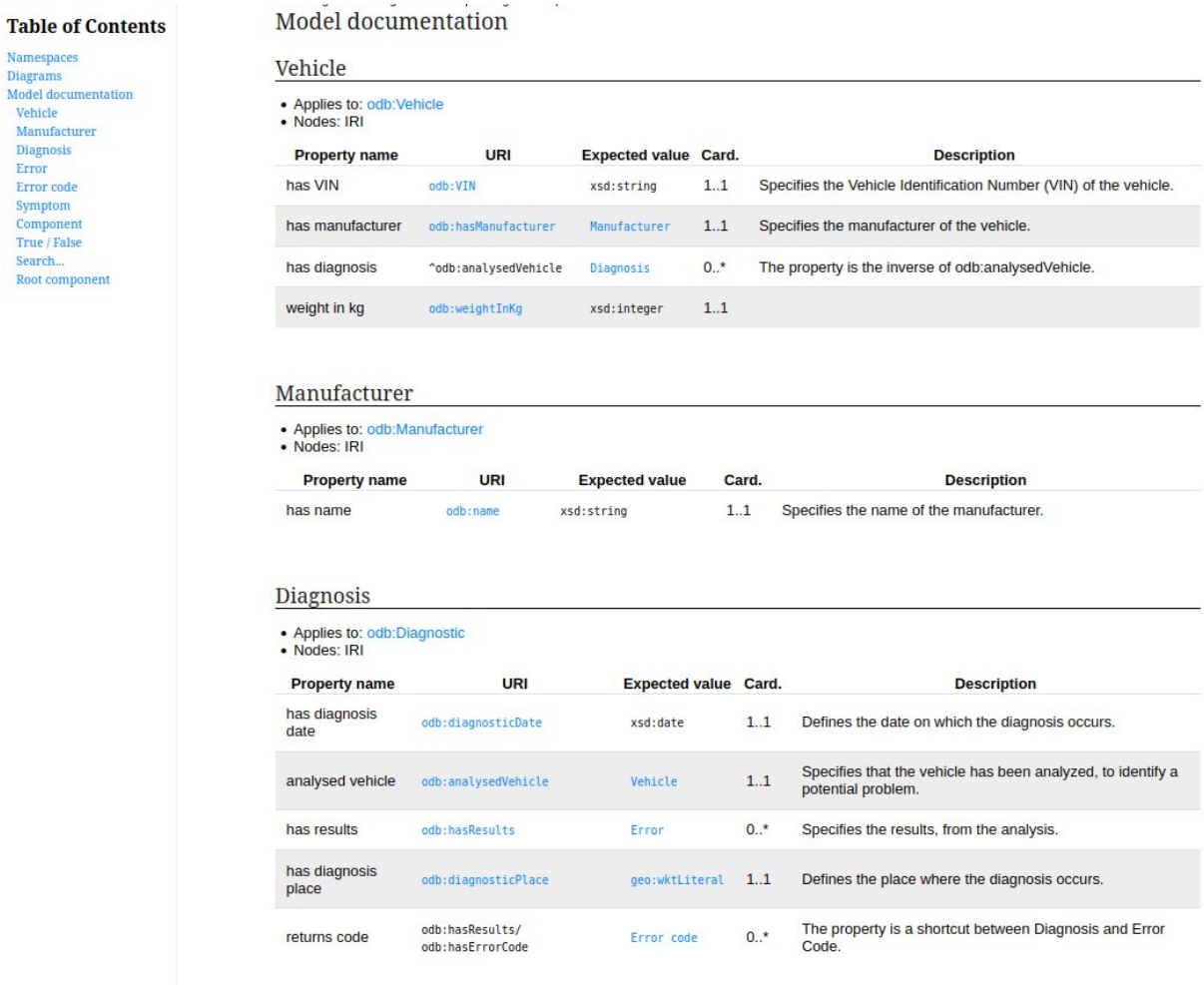


Table of Contents

Model documentation

Vehicle

- Applies to: odb:Vehicle
- Nodes: IRI

Property name	URI	Expected value	Card.	Description
has VIN	odb:VIN	xsd:string	1..1	Specifies the Vehicle Identification Number (VIN) of the vehicle.
has manufacturer	odb:hasManufacturer	Manufacturer	1..1	Specifies the manufacturer of the vehicle.
has diagnosis	^odb:analysedVehicle	Diagnosis	0..*	The property is the inverse of odb:analysedVehicle.
weight in kg	odb:weightInKg	xsd:integer	1..1	

Manufacturer

- Applies to: odb:Manufacturer
- Nodes: IRI

Property name	URI	Expected value	Card.	Description
has name	odb:name	xsd:string	1..1	Specifies the name of the manufacturer.

Diagnosis

- Applies to: odb:Diagnostic
- Nodes: IRI

Property name	URI	Expected value	Card.	Description
has diagnosis date	odb:diagnosticDate	xsd:date	1..1	Defines the date on which the diagnosis occurs.
analysed vehicle	odb:analysedVehicle	Vehicle	1..1	Specifies that the vehicle has been analyzed, to identify a potential problem.
has results	odb:hasResults	Error	0..*	Specifies the results, from the analysis.
has diagnosis place	odb:diagnosticPlace	geo:wktLiteral	1..1	Defines the place where the diagnosis occurs.
returns code	odb:hasResults/ odb:hasErrorCode	Error code	0..*	The property is a shortcut between Diagnosis and Error Code.

Annex : Generate SHACL automatically from an RDF Knowledge Graph

It is possible to generate automatically a SHACL specification from the analysis of an RDF knowledge graph. Such a specification can serve to bootstrap a Sparnatural configuration adjusted by hand, or can be fed directly to Sparnatural, since Sparnatural has default behaviours that allows it to work with a plain SHACL file (without Sparnatural-specific annotation) – of course, in that case, features like icons or ordering of entities will be missing.

Sparna maintains SHACL Play! At https://shacl-play.sparna.fr . This tool, amongst other features, allows to generate a SHACL specification profile from the analysis of an RDF Knowledge Graph. The SHACL generation form is freely available online but we suggest to run this kind of analysis using the command-line tool. The algorithm to derive the SHACL specification is precisely documented.

Sparnatural deployments such as https://www.nakala.fr/sparnatural/ are configured with a SHACL generated automatically.