THE KARP FRONTEND

INSTALLATION AND CUSTOMIZATION

Karp 1.0b1, 2015-12-04

DRAFT

SPRÅKBANKEN
UNIVERSITY OF GOTHENBURG

Contents

1. Installation

You can run the Karp frontend from any web server, but when you are working locally with Karp, it is recommended that you use the provided Grunt (www.gruntjs.com) script.

To run the frontend through the Grunt script, you will first need to install node.js (www.nodejs.org) on your computer, preferably through a package manager (https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager) or download an installer (http://nodejs.org/download/).

You can then use the node-js package mangager (npm) to install some other things you need.

Start off with the global dependencies (*Bower* and the *Grunt Command Line Interface*):

```
> npm install -g bower grunt-cli
```

After that you can install all local dependencies by standing in the base Karp directory and typing:

```
> npm install && bower install
```

And to start up Karp under node.js, stand in the Karp directory and type:

```
> grunt serve
```

Karp should now open up automatically in your web browser. If doesn't you can go to:

http://localhost:9000/

CoffeeScript files are automatically compiled to JavaScript as required, additionally causing the browser window to be reloaded to reflect the new changes.

2. Building Karp

Before you upload your Karp instance to your server, you should build it. Do this simply with:

```
> grunt build
```

You will find the build in the `dist/` directory. You can start the server with the built version of Karp using:

```
> grunt serve:dist
```

3. Configuration files and templates

Normally you need to make changes to the following files:

app/config.js
app/lexicons/<your_lexicon_id>/template.html

and if you want your lexicon to be editable, you also have to make changes to:

app/lexicons/<your_lexicon_id>/editor_config.js

app/config.js – the main configurations file

The main config file should define a JavaScript object called 'settings'

```
var settings = {};
```

The settings object can have the following members:

languages (object) *required*
definitions of language names for use with the GUI and with multi-language resources. The **name** should should correspond to a translation key and the **localName** should be the name in the respective language.

```
settings.languages = {};
settings.languages.eng = {
      name : "lang_english",
      localName : "English"
};
settings.languages.swe = {
      name : "lang_swedish",
      localName : "Svenska"
};
...
```

interfaceLanguages (array of strings) *required*
An array with all the languages to use with the GUI.

```
settings.interfaceLanguages = ["swe", "eng"];
```

defaultLanguage (string) *required*
The ISO code for the GUI language used by default.

```
settings.defaultLanguage = "swe";
```

backendBaseURL (string) *required*

The base URL (without any commands) to YOUR Karp Backend.

```
settings.backendBaseURL =
"https://ws.spraakbanken.gu.se/ws/karp/v1";
```

korpBackendURL (string)

The URL for your *Korp* backend, if you have one.

lexicons (object) *required*

Some basic settings for each lexicon you want to use. Each key should be a lexicon ID.

```
settings.lexicons = {};
settings.lexicons.saldo = {
      …
};
...
```

The lexicon objects may have the following members:

title (string) *required*
The human readable name of the lexicon. It will be translated if there is a translation key.

desc (string)
A short description of the lexicon. It will show up in the lexicon chooser.

labelField (string) *required*
The ID of a field definition to use for displaying labels in listings.

extraLabelField (string)
An optional extra ID of field definition to use for secondary information with labels in listings.

template (string) *required*
A path for the Angular-JS html template to use for showing the search results.
We suggest the path: `"lexicons/YOUR_LEXICON_ID/template.html"`

editorConfig (string)
A path to a config file for the editor. If this property exists, Karp will enable the editing capabilities for the lexicon.
We suggest the path: `"lexicons/YOUR_LEXICON_ID/editor_config.js"`

fields (object) *required*

Definitions for the different data fields. Each key should be the ID of the field (which normally is the same ID as is used in the backend).

```
settings.fields = {};
settings.fields.sense = {
      ...
};
…
```

NOTE: You don't need to make definitions for all fields of your data in the config.js file, but the fields you define will be searchable in the extended search tab and will also be listed in the *Statistics* tab. You will also be able to use them as labels for entries in various listings.

The field objects may have these members:

caption (string) *required*

The human readable name of the field. It will be localized if there is a translation key available.

```
...
caption : "sb_index_sense",
...
```

realIndex (string)

Normally the key of the field object must be the same as the backend field, but if you want to make an alias to a field (e.g. you could make a 'synset' item which is actually searching a 'sense' field) you will have to use 'realIndex' which must be the ID of the backend field. You will probably not need to set this property.

```
…
realIndex : "sense",
...
```

labelPath (string) *required*

The path to the field in the json entry object. It is needed for the statistics results view and for generating listings.
NOTE: If the corresponding backend field defines more than one json path, you should choose the one which makes the most sense to use as label for a result.

```
...
labelPath : ".Sense.senseid",
...
```

relations (array of strings) *required*

The relations which should be used as alternatives for the extended search.

Possible values:

"equals"	
"contains"	
"startswith"	
"endswith"	
"exists"	
"missing"	
"regexp"	
"lte"	(less than)
"gte"	(greater than)

```
...
relations : ["equals", "missing"],
...
```

resources (array of strings) *required*

The ID of each lexicon which uses this particular field.

```
...
resources : ["saldo", "swefn"],
...
```

formatWith (string)

An ID of a 'GUI handler' to use for the statistics rendering of the field. (See the section *GUI handlers*.)

```
...
formatWith : "stringHandler",
...
```

formatArgs (array of strings)

Any arguments you wish to supply to the GUI handler.
(See the section *GUI handlers*.) Defaults to an empty array.

```
...
formatArgs : ["example"],
...
```

set (string)
An ID for a set definition which will be used as alternatives in the extended search.
(See the section *Sets*.)

```
...
set : "posTags",
...
```

folders (object)
An optional folder hierarchy for the lexicon chooser.

```
settings.folders = {};
settings.folders.modernLexicons = {
    title : "modern",
    contents : ["saldo",  "swefn"]
};
settings.folders.olderLexicons = {
    ...
};
…
```

The **title** will be translated if there is a corresponding translation key. Any lexicons not in a folder will be added to the first level in the hierarchy.

sets (object)

All sets should be defined here. Sets are used in the extended search as well as in the editor, usually in the form of a dropdown menu form which the user can choose an item to search for or to add as trait to a lexicon entry.

type (string) *required*
Should be one of `"entries"`, `"closed"` and `"dynamic"` [dynamic is not yet implemented!].
The `"entries"` type needs a **query** and a **lexicon** on which to apply the query.

```
settings.sets.frames = {
     type : "entries",
     query : "extended||and|sense|exists",
     lexicon : "swefn"
};
```

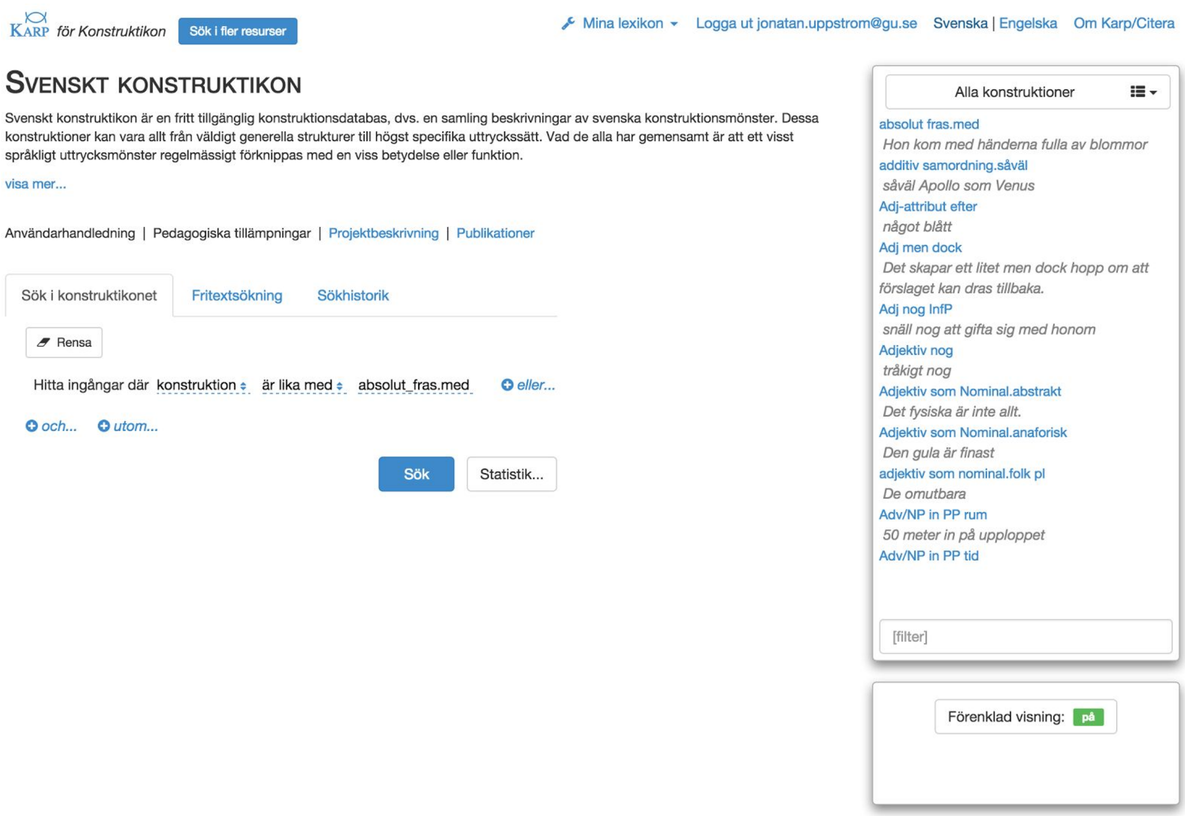
The actual set will be all entries found. The **labelField** in the lexicon definition will be used.

The `"closed"` type needs an **items** array, where each item has an **id** and a **t** member.
The **id** is what will actually be used as model data for constructing queryies and what will be saved in the lexicons when editing. The **t** is the translation key for what is presented in the GUI.

```
settings.sets.swefnDomains = {
    type : "closed",
    items : [
        {id: "Gen", t : "swefn_general"},
        {id: "Med", t : "swefn_medical"},
        {id: "Art", t : "swefn_arts"}
    ]
};
```

modes (object) *required*

It is possible to have different modes in the same Karp installation for different kinds of user groups. For example, Språkbanken has a special mode for user of the Swedish Constructicon:



SVENSKT KONSTRUKTIKON

Svenskt konstruktikon är en fritt tillgänglig konstruktionsdatabas, dvs. en samling beskrivningar av svenska konstruktionsmönster. Dessa konstruktioner kan vara allt från väldigt generella strukturer till högst specifika uttryckssätt. Vad de alla har gemensamt är att ett visst språkligt uttrycksmönster regelmässigt förknippas med en viss betydelse eller funktion.

visa mer...

Användarhandledning | Pedagogiska tillämpningar | Projektbeskrivning | Publikationer

Sök i konstruktikonet | Fritextsökning | Sökhistorik

Rensa

Hitta ingångar där konstruktion är lika med absolut_fras.med eller...

och... utom...

Sök Statistik...

Alla konstruktioner

absolut fras.med
Hon kom med händerna fulla av blommor
additiv samordning.såväl
såväl Apollo som Venus
Adj-attribut efter
något blått
Adj men dock
Det skapar ett litet men dock hopp om att förslaget kan dras tillbaka.
Adj nog InfP
snäll nog att gifta sig med honom
Adjektiv nog
tråkigt nog
Adjektiv som Nominal.abstrakt
Det fysiska är inte allt.
Adjektiv som Nominal.anaforisk
Den gula är finast
adjektiv som nominal.folk pl
De omutbara
Adv/NP in PP rum
50 meter in på upploppet
Adv/NP in PP tid

[filter]

Förenklad visning: på

The differences from the standard interface include:

- There is no lexicons chooser. Construkticon is the only lexicon to search in.
- There is a special header with relevant information.
- The extended search tab has a different label.
- The extended search and freetext search tabs have switched place.

- The sidebar is visible by default and its lists are predefined.
- The results are shown in simplified way by default and there is a button under the sidebar to turn the simplified mode on and off.

There should always be at least one mode, the **DEFAULT** mode:

```
settings.modes = {};
settings.modes.DEFAULT = {
      defaultExtended : "and|pos|equals|nn",
      listings : [
            {
                  set : "constructions",
                  buttonCaption : "const_in_konst"
            },
            {
                  set : "frames",
                  buttonCaption : "frames_in_swefn"
            }
      ]
};
```

The members of the mode objects are the following:

defaultExtended (string) *required*

The default query for the extended search.

listings (array of objects) *required?*

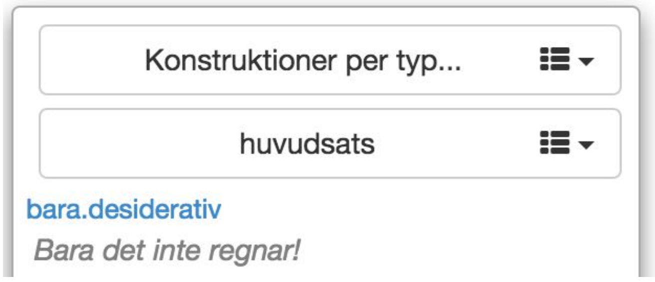
These are the predefined choices for the sidebar listings. There are two kinds of listings.

Direct listings

These listing object should have a **set** (the ID of a set defined in settings.sets) and a **buttonCaption** (which will be translated of there is an available translation key). The user will directly see a list of entries.

Indirect listings

These listings don't reference a set directly. They instead define a **field** property which will be used for generating a second dropdown menu where the user can select to list all entries where the particular field matches the choice in the second menu.

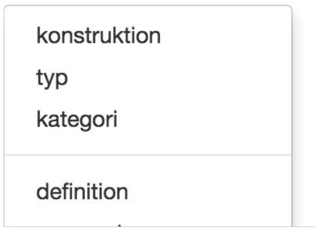


useKeyboard (boolean)
Controls whether or not to show the "helper keyboard" beside the textbox for the simple search. Defaults to **true**.

simpleSearchCaption (string)
The translation key for customizing the simple search tab.

extendedSearchCaption (string)
The translation key for customizing the extended search tab.

prioritizeFields (array of strings)
Use this to highlight the most important fields to search in the extended search.



```
prioritizeFields : ["construction", "type", "cat"]
```

sidebar (string)
Should the sidebar be visible? Possible values:

"NEVER"
The sidebar will not be used at all.
"MINIMIZED" (the default value)
The sidebar will be hidden but can be shown by clicking 'Listings' in the top menu on the screen.
"MAXIMIZED"
The sidebar will be shown but can be hidden by clicking on the arrow in the top left of the sidebar.
"ALWAYS"
The sidebar will always be shown.

simplifiedAndAdvanced (boolean)

If true, there will be a switch below the sidebar to choose between a simplified and an advanced mode. Defaults to **false**.

In the lexicon templates, `<tr karprow advanced="true" ng-model="hit._source.myField">` ... `</tr>` can be used. Such a row will be hidden either if the model is empty or if simplified mode is on. You can use this on other HTML tags as well, despite the name `karprow`.

switchSearchTabs (boolean)

Makes the extended search the default (and first) tab. Defaults to **false.**

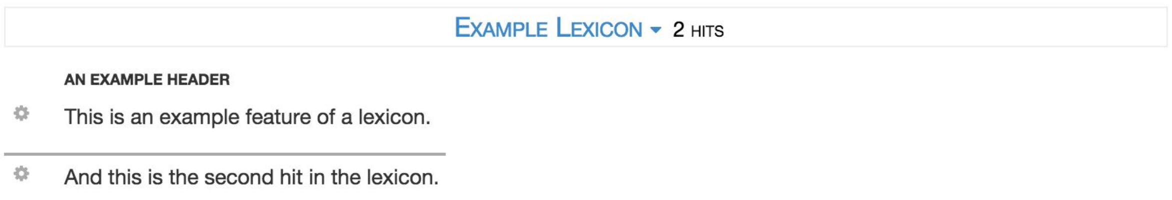
Lexicon templates

Each lexicon should have its own display template for the Karp results. The file should preferrably have this path: app/lexicons/<lexicon_name>/template.html

A minimal template could look like this (The **bold** parts of the example are parts that you probably want to change and extend in your own template.):

```
<table class="templateTable">
    <thead>
        <th></th>
        <th>{{'translationKeyForTheExampleHeader' | loc}}</th>
    </thead>
    <tbody>
        <tr ng-repeat="hit in hits">
            <td>
                <console ng-model="hit._id" resource="exampleLex"/>
            </td>
            <td>
                {{hit._source.exampleText}}
            </td>
        </tr>
    </tbody>
</table>
```

This will result in this view, with two entries found:



AN EXAMPLE HEADER

This is an example feature of a lexicon.

And this is the second hit in the lexicon.

If you are not familiar with AngularJS templates, https://docs.angularjs.org/guide/templates is a valuable source of information. Especially `ng-repeat`, `ng-if` and `ng-show` are important concepts to understand.

There are a few things you should note:

- The {{ ... }} notation can be used to output data from the actual result. The `ng-repeat` directive tells Karp to create one table row for each hit, where the model data for the hit is stored in `hit._source` .
- The `| loc` (Karp specific) filter should be used when you want to localize the string using a translation key.
- The `<console ng-model="hit._id" resource="exampleLex"/>` part is used for displaying the cog button with which the user (among other things) may choose to edit the entry. You should change `exampleLex` to your lexicon ID.
- You can use the `| ensurearray` filter (Karp specific, not shown above) to make sure that something is always an array (if it is not an array then it is turned into an array with only one item). This can be of help when you want to use `ng-repeat` to make lists of data that is not guaranteed to be an array.
  An example:
- You can use `<tr karprow ng-model="hit._source.myField">` ... `</tr>` to hide the row if the model is empty, so you can make sparse entries more compact. You can use this on other HTML tags as well, despite the name `karprow`.

```
<span ng-repeat="p in hit._source.pos | ensurearray">
      {{p}}
      <span ng-if="!$last">, </span>
</span>
```

4. Translating the user interface

In the `app/translations/` directory you will find the translation files. The names should be `locale-`*languageID*`.json` and `domain-`*languageID*`.json` , where *languageID* is one of the items in `settings.interfaceLanguages` in `config.js`. Make sure that the same language IDs are also present as keys in the `settings.languages` object, where the human readable language names are defined.

The `locale`[...]`.json` files contains the translation keys for the main user interface, while the `domain`[...]`.json` files contain keys specific for your domain, including lexicon names, labels in your templates, field names etc.

5. Using Karp as a lexicon editor

Lexicon config files

Each lexicon to be edited needs to have its own editor config file as well. The `settings.lexicons.`*`your_lexicon_id`*`.editorConfig` property in the main config.js file should point to this file. Normally it is named *editor_config.js* and is placed in the *lexicons/your_lexicon_id/* directory.

The skeleton for the file looks like this:

```
var structure = {};
var template = {};
registerKarpEditorConfig("the_lexicon_id", structure, template);
```

structure defines the structure of an entry in the lexicon, and how it should be shown in the user interface.

template is a javascript object that will be cloned and used as template when the user makes a *new* entry in the editor.

NOTE: The lexicon config files are loaded dynamically by Karp which can make them harder to debug than the general config.js file since compiling errors and exceptions will not show up red in your browser console. However, if Karp cannot load a file, it will tell you as much as it can about the problem and (if possible) return any error codes or exceptions in blue text in the browser console, like so:

```
WARNING. Could not load swefn editor config from lexicons/swefn/config.js:
SyntaxError: Unexpected string
```

If you still can't find the error, you can copy-paste portions of your file into the browser console to see where the browser complains.

You can also suffix (put it on the last line in the file) your files with:
`//@ sourceURL=a-name-of-your-choice.js`
This tells your browser to let you use its debugging tools.
In Chrome, you will have to look for the file in the '*(no domain)*' collection under *Sources*.

The structure object

The structure object is a hierarchical representation of an entry in the lexicon. Here is an example:

```
var structure = {
    "properties": {
        "lemma_german": {
            "label": "Baseform German",
            "multi": false,
            "dummy": true,
            "handler": "stringHandler",
            "collapsable": false
        },
        "pos_german": {
            "label": "Part of speech German",
            "multi": false,
            "dummy": true,
            "handler": "comboStringHandler",
            "set": "panaceaPos",
            "collapsable": false
        },
        "english": {
            "label": "English",
            "multi": true,
            "dummy": true,
            "collapsable": true,
            "properties": {
                "lemma_english": {
                    "label": "Baseform English",
                    "multi": false,
                    "dummy": true,
                    "handler": "stringHandler",
                    "collapsable": false
                },
                "pos_english": {
                    "label": "Part of speech English",
                    "multi": false,
                    "dummy": true,
                    "handler": "stringHandler",
                    "collapsable": false
                },
                "package_prob": {
                    "label": "Package probability",
                    "multi": false,
                    "dummy": true,
                    "handler": "numberFieldHandler",
                    "collapsable": false
                },
                "target_prob": {
                    "label": "Target probability",
                    "multi": false,
                    "dummy": true,
                    "handler": "numberFieldHandler",
                    "collapsable": false
                },
                "corpus_prob": {
                    "label": "Corpus Probability",
                    "multi": false,
                    "dummy": true,
                    "handler": "numberFieldHandler",
                    "collapsable": false
                }
            }
        }
    }
};
```

Each item can either be a branch (if it has its own `"properties"` property) or a leaf. The topmost level is always a branch. In the example above, there is one additional branch, `"english"` and some leaves; `"lemma_english"`, `"target_prob"`, `"corpus_prob"` etc. These are not magical Karp keywords but features of the particular lexicon.

However, each field should have properties that decide how the editor will present and work with the data:

label (string)
The human readable label for the field in the editor.

multi (boolean)
Should the field/branch be an array? If set to **true**, a plus icon will show up in the editor next to the item to add a new field/branch and it will be handled as an array internally.

dummy (boolean)
If set to **true**, a "dummy" item will always be shown in the editor if the field has not (yet) been filled. Defaults to **false**.

handler (string)
An ID of a *GUI handler* which should be used for editing the particular field. It is only relevant to leaves. A leaf can be a simple data item like a boolean or a string but it

can also be a complex json structure. It is always atomic in the way that its editing is handled as one unit - using a *GUI Handler*. (See the section *GUI Handlers.*)

args (unspecified array)
For supplying arguments to the GUI handler – making it possible to customize some aspects of editing. Defaults to an empty array.

set (string)
An ID of a set defined in the main config file. How the set is used depends on the GUI handler.

alwaysOneItem (boolean)
This property can be set to **true** when you have an array but you know you will always have only one item in it. It mimics the looks and behaviour of single item even though it's actually an array. You will for example not see any plus icons for adding new fields/branches. Defaults to **false**.

collapsable (boolean)
Can be used if you don't want the item to be collapsable with the disclosure triangle. Defaults to **true**.

indentation (boolean)
Normally a branch gets +1 indentation level. Use "indentation" : **false** to supress this behavior. Defaults to **true**.

note (string)
Use this to add a "post-it note" to the field with extra information to the user.

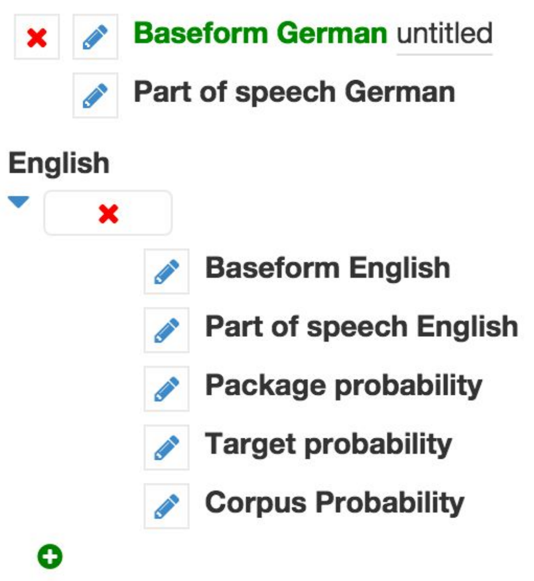
The template object

When the user creates a *new* entry, the template object gets cloned and becomes the new entry data.
Here is an example of a template object for the lexicon above:

```
var template = {
      "lemma_german": "untitled",
      "english": [{}]
};
```

We make `"english"` an array with one empty object so that we start of with one empty item in the editor.

Thus, when a new entry is created in the editor, it will then look something like this:



There are more unfilled fields than what is defined in the template, these are the so called *dummy* fields from the **structure** object.

NOTE: It is possible to use the special string "__AUTOFILL:USERNAME" in the template object to fill a field with the username of the current user (not shown in the example above).

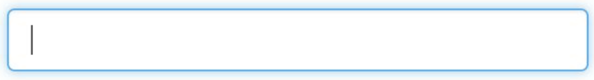
GUI Handlers

A GUI handler is responsible for:

In the editing mode:
- displaying the GUI when editing
- generating HTML for the non-edit state of the field
- handle the connection between the GUI and the actual model data

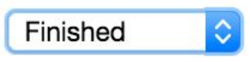
There are several predefined handlers to choose from:

"stringHandler"
Proides a simple text field for editing.

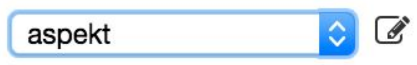


"numberFieldHandler"
A field that only accepts numbers.

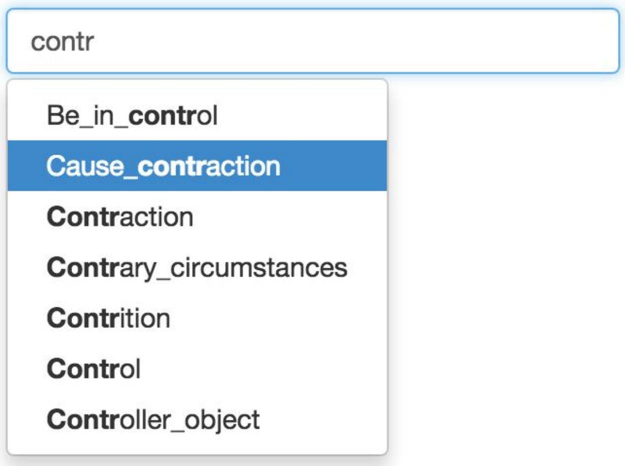
"choiceStringHandler"
Provides a dropdown list for choosing between alternatives in a set.



"comboStringHandler"
Combines the two above handlers by letting the user switch between using a dropdown list of predefined items in a set and using a text field for any additional input.



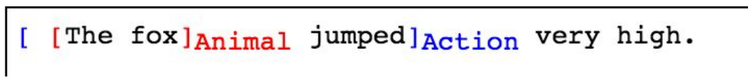
"autocompletionStringHandler"
Provides a text field which has an autocompletion feature for selecting items from a set. The user may still assign other values as well.



"checkboxHandler"
A simple checkbox which results in either true or false.

"markupHandler"

Let's the user edit marked up XML inside of the json structure, like so:



[[The fox]Animal jumped]Action very high.

The underlying data would look like this:

```
{
	"xml" : '<example><e n="0" name="Action"><text n="0">
	</text><e n="1" name="Animal">The fox</e><text n="2">
	jumped</text></e><text n="1"> very high.</text></example>'
}
```

This can be useful for annotating things like examples, idiomes etc.

The `n`-indexes can be useful if you want to convert the xml into json later on, but you might just as well ignore them if you are only interested in exporting for xml in the end.

The top level tag (in this case '<example>') is decided by supplying a string as the first argument to the handler:

```
...
"handler" : "markupHandler",
"args" : ["example"]
...
```

NOTE: The HTML generator capacity of the GUI handler may also be used from inside a lexicon template by using the directive **`<handler>`**, like this:

```
<handler id="markupHandler" ng-model="hit.example" args=""/>
```

Creating your own GUI handlers

If you feel that the predefined GUI handlers aren't suited for you needs, you can add your own. You should preferrably put them in the file `app/scripts/domain_gui_handlers.coffee` to lessen the risk of broken code in future versions of Karp. The header in this file provides the necessary information.