An API for the CorpusStudio web application

Erwin R. Komen

Start: *October 8, 2015*
This version: 1.6
Date: *October 24, 2017 (09:59)*

CONTENTS

1 Introduction

The “CorpusStudio” web application is the web-service based equivalent of the Windows program with the same name “CorpusStudio” (Komen 2009). The web application consists of three main components.

1) Main back-end: a web service `/crpstudio` that communicates with the ‘engine’ and the front-end
2) Engine back-end: a web service `/crpp` that executes CRPs (Corpus Research Projects) on a corpus (on the same server)
3) Front end: a HTML/JavaScript component that only interacts with the `/crpstudio` back-end

1.1 Short overview: principle components

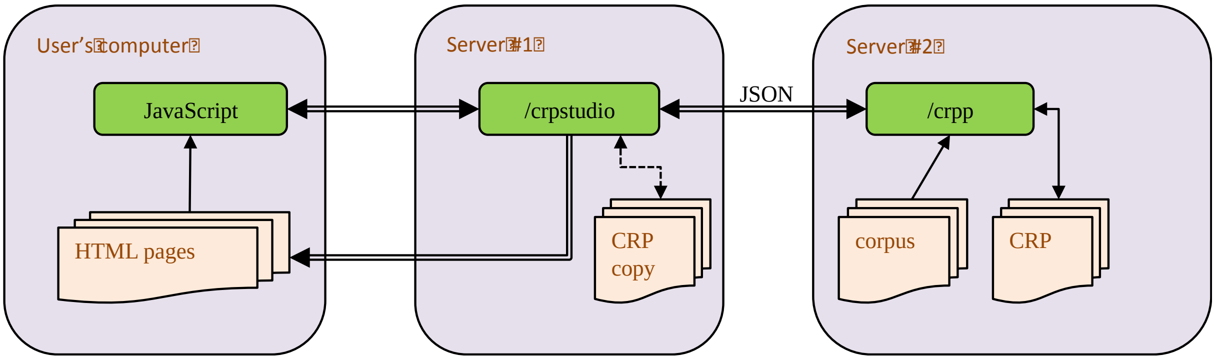

Figure 1 Main components of the CorpusStudio web application

Central in the architecture is the `/crpstudio` component (service #1), since this is the service to which the user first connects (by typing the URL of the service in a browser). Communication between the different modules can be divided into several parts:

1) **User to /crpstudio**: The Java code of `/crpstudio` takes in the user’s request to start a CorpusStudio session and reacts by creating and sending an *html* page. This page is part of the front-end, the user-interface. The html pages produced by `/crpstudio` are ‘reactive’: events like clicking a button or typing text on the web page result in calls to a JavaScript function—also part of the front-end. The JavaScript functions handle only part of the UI requests, holding only a limited amount of user-data internally.
2) **Javascript to /crpstudio**: The JavaScript functions (which are called as a reaction to the user clicking a web page or typing in a web form) only communicate with the `/crpstudio` service. They may need additional information about a CRP or database, which they do by issuing a POST request to `/crpstudio`. They may need to start an action (such as renaming a file, deleting a file and so on), and this too is done by issuing a POST request to `/crpstudio`. The requests may result (a) in a response in JavaScript (after which the *html* page is updated with the new information), or (b) in a new *html* page being created by `/crpstudio` which then appears on the browser of the user.
3) **Service /crpstudio to service /crpp**: The `/crpstudio` component handles many requests by communicating with the ‘engine’ service `/crpp`. A request to provide a list of CRPs belonging to the current user, for instance, is a `/crpp/crplist` command with a JSON object argument containing the user’s id. The `/crpp` service looks in the hosting (virtual or actual) computer, checks which CRPs belong to the user and returns a JSON list of them to `/crpstudio`. It is only the `/crpp` service that has access to the *xml* corpora that are searched, that holds the search results, the result databases and so forth.

The current implementation of `/crpstudio` holds copies of CRPs on its server when these are accessed by a user, but this may change in the future. The main goal of `/crpstudio` is to serve as a central place where user's requests enter, where commands to get information are issued and where responses are processed and then fed back to the user.

The overview of the web application set-up in Figure 1 shows the `/crpstudio` and the `/crpp` services as being separate. They may run on the same computer or on different computers, since they have separate places where they store information, and the communication between them is strictly through web service calls. The `/crpp` component is completely independent, so that it may serve as a back-end to other applications such as desktop or smartphone applications from any computer platform or language.

1.2 Short overview: interacting with the CRPP webservice

The `/crpp` can be used independently of the other components. The most important steps for this web service are these:

1) Set up the **Linux** computer that contains the (Java) web service (TODO: ref)
2) Check if `/crpp` is ready by a simple command without parameters.
3) Use a `/crpp/crpset` command to **upload** a CRP to the service (TODO: ref)
4) Issue a `/crpp/exe` command to start **executing** the CRP and extract the `jobid` from the initial response (**Fout! Verwijzingsbron niet gevonden.**)
5) Monitor the **progress** by issuing a `/crpp/statusxq` command every now and then (4.14).
6) Once the response (TODO: ref) has the status code "`completed`", a **table** overview of the results is included in the `/crpp/statusxq` response.
7) Download individual parts of the results using the `/crpp/update` command (TODO: ref)
8) Download a results database: *not yet implemented*
9) Download lexicon information resulting from `ru:lex()`: *not yet implemented*

2 Overview of the CorpusStudio components

The first chapter of this API provides a sketchy general overview of the three main components of CorpusStudio. This chapter shows the web application from a slightly different point of view, dividing it into modules and showing some key settings files.

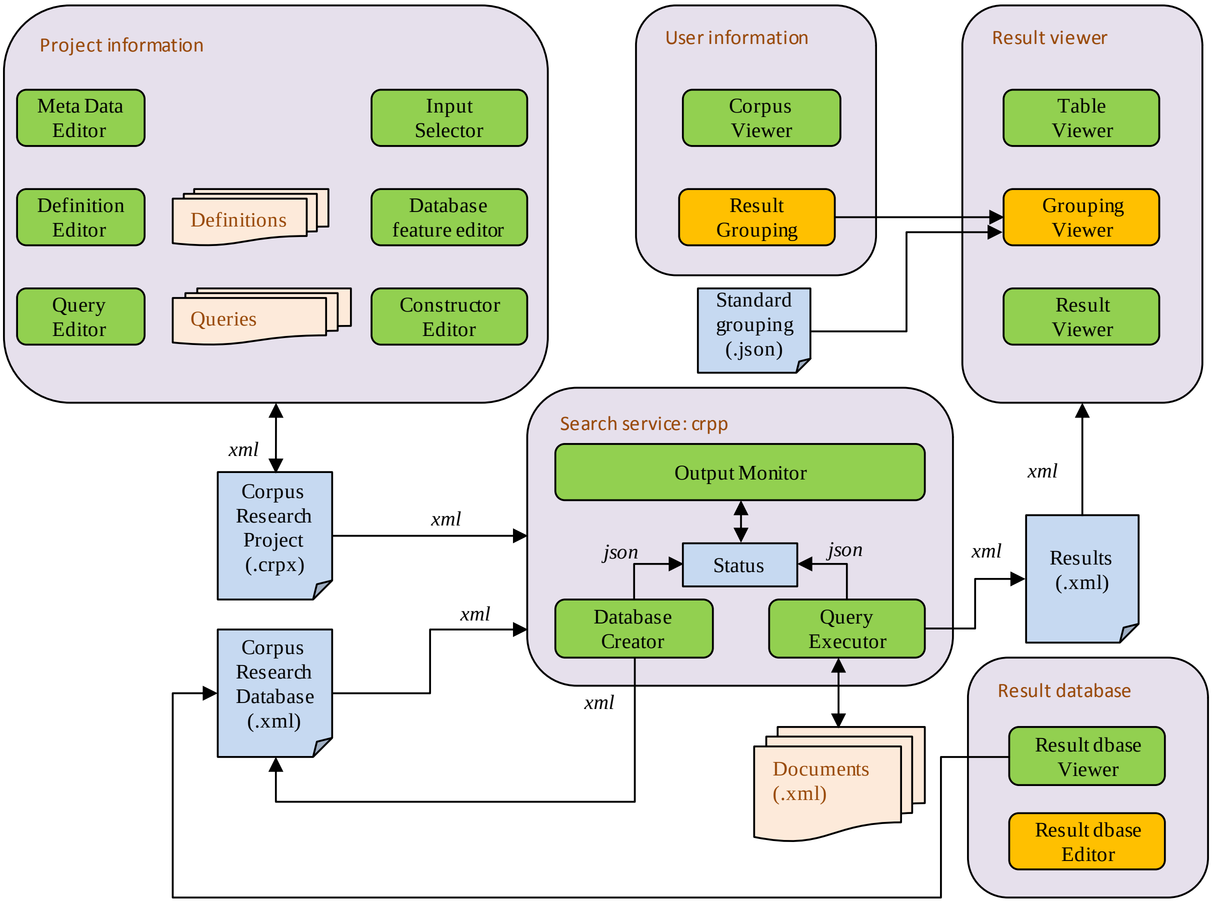

Figure 2 CorpusStudio overview

A central component is the "`Search service`", the `/crpp` web service. This service takes a CRP as input, executes the queries in this CRP in the order specified by the constructor on the documents selected by the input. The input texts are expected to be somewhere in the `/etc/corpora` area, as defined more specifically by the "`indices`" section of the "`crpp-settings.json`" file that can be found in the `/home/erwink/webapps` directory. The output of a CRP consists of several parts:

1) **Table overview**. A file called `[crp-project-name].table.json` in the `/etc/project/{user}/out` directory. This contains an overview of the result counts per QC and per sub category.
2) **Hits per text**. Detailed hit information for each text is created in the directory `/etc/project/{user}/out/{crp-name}/hits`. Hits are divided over the sub categories and each hit contains location information as well as a possible user-defined "msg" string. This string is the database information if the query line of the CRP is meant to produce a database.
3) **Lex results per text**. Detailed information on the results of the `ru:lex()` function for each text is created in the directory `/etc/project/{user}/out/{crp-name}/hits` in files that end on "`.lex`".
4) **Result database**. If a result database is produced (as defined in the Constructor Editor), then it is placed in the directory `/etc/project/{user}/dbase`.

The "`Project Information`" component is implemented by the `/crpstudio` web service and the JavaScript code. The parts of this component all work together to define a CRP. The component logically divides into a number of different 'editors', which provide the user interfaces to these components.

1) **Meta data editor**. Provides access to the 'general' part of a CRP. This contains meta information such as *author*, *goal* and *comments* about the project. The general part also allows operations on whole CRPs, such as: downloading, uploading, removal and execution.
    a) User-interface: "selector.vm", "projectmain.vm" and "prjcreate.vm" (loaded by "projects.vm")
2) **Input selector**. Allow user to specify which corpus, or which part of a corpus, the queries must be executed on.
    a) User interface: "input.vm" (loaded by projects.vm).
3) **Definition editor**. Allow user to provide Xquery global variable and user-function declarations.
    a) User interface: "selector.vm", "definitions.vm", "defcreate.vm" (see projects.vm)
4) **Query editor**. Allow user to edit, upload and download queries written in Xquery.
    a) User interface: "selector.vm", "queries.vm", "qrycreate.vm" (loaded by projects.vm)
5) **Constructor editor**. Define the execution order and hierarchy of queries. The 'query line' selected here can be provided with result database features.
    a) User interface: "selector.vm", "constructor.vm", "qccreate.vm" (see projects.vm)
6) **Feature editor**. Define the result database features (=fields) for the currently selected query line in the constructor editor.
    a) User interface: "selector.vm", "dbfeat.vm", "dbfcreate.vm" (loaded by projects.vm)

The '`User information`' component is realized as the 'Corpora' main menu part. It allows the user to explore the available corpora and it allows the user to define result 'groupings' for each of them. The component divides into two parts:

1) **Corpus viewer**. Explore the corpora that are available to the current user. Note that users can only view, explore and work with corpora that are not 'hidden' to them.
    a) User interface for simple explorer: "crpmain.vm" (loaded by corpora.vm)
    b) User interface for what could potentially serve as editor: "selector.vm", "crpedit.vm" (see corpora.vm)
2) **Result grouping**. Define 'groupings': mappings between texts (file names) and the group label they are to be part of, based on meta data information per text. The meta data is accessable through meta variables. These variables have names that are independent of the corpus they are used in, while their definition depends on the corpus and the project type. *This part awaits implementation*.
    a) User interface for groupings: selector.vm, crpgrouping.vm (see corpora.vm)
    b) User interface for metavariables: selector.vm, crpmetavar.vm (see corpora.vm).

The '`Result information`' component facilitates users viewing the quantitative and qualitative results of a CRP in different ways. The component divides into several parts:

1) **Table viewer**. Give a tabular summary of the results or part of the results.
    a) User interface: "result.vm" (loaded by projects.vm)
2) **Result viewer**. Provide detailed information for each hit. Either organized directly per hit, or by-document-by-hit. Each hit gets context and syntax. The number of hits available per 'view' is adjustable.

   a) User interface: "result.vm" (loaded by projects.vm)

3) **Grouping viewer**. Quantative results of the hits in a table. Each column is one of the groups provided by the current 'grouping', and each row is a sub categorization of the hits.
   a) User interface: "result.vm" (loaded by projects.vm)

The '`Result database`' component allows users to work with 'Corpus Result Databases'. These databases can be created as a result of executing a CRP, they can be downloaded and they can be uploaded. The component divides into several parts:

1) **Result dbase viewer**. Allow uploading, downloading, renaming and exploring of result databases.
   a) User interface: "selector.vm", "dbmain.vm" (loaded by dbases.vm)
2) **Result dbase editor**. View and edit records of the currently selected result database. *This component awaits implementation.*
   a) User interface: "selector.vm", "dbedit.vm" (loaded by dbases.vm)

Most of the 'editors' mentioned above make use of the 'selector.vm' template. This is a velocity template that looks for three variables the user needs to define: `$explorespec`, `$item_rec` and `$item_list`. See the template 'projects.vm' for examples.

3 Notes on the /crpstudio server component

This section contains a number of notes on selected topics implemented in the /crpstudio server.

3.1 User accounts and logging in/off

Information on the users is kept in a file called `crpstudio-settings.json` within the /etc/crpstudio directory of the server implementing the /crpstudio server (see section 5.1.4).

4 The /crpp web service input specification

The /crpp web service accepts POST, PUT and GET. The POST and GET methods can be used to send data (a query) to the web service. Here is a brief overview of the commands:

command	**Goal**
`/crpchg`	Changes in a CRP, creation of a CRP
`/crpdel`	Remove a CRP from the server
`/crpget`	Download a CRP from the server (as plain text)
`/crpinfo`	Get *date* information on one particular CRP: modified/changed
`/crplist`	Get a list of CRPs for the indicated user
`/crpset`	Upload a CRP for the indicated user
`/dblist`	Provied a list of databases in the storage of the indicated user
`/debug`	Provide a message to show the JAVA service works okay
`/exe`	Execute a CRP on a particular corpus and/or database
`/export`	*(not really used right now)*
`/load`	*(not really used right now; see* `/crpget`*)*
`/reset`	Stop execution of the indicated Xq job and all underlying jobs *(doesn't work properly yet)*
`/save`	*(not really used right now)*
`/serverinfo`	Show information available in `crp-info.json`
`/settings`	Show links between CRP and corpus for the indicated user
`/show`	*(not really used right now)*
`/statusxq`	Provide status information on the Xq job for the indicated user
`/txt`	One text: get sentences or details of one sentence
`/txtlist`	Give a list of all texts for a particular lng / dir / ext
`/update`	Give result information for a particular CRP (the 'job' is not looked for)

Important notes:

1) All commands can be issued to the /crpp web service through:
   `http://server-address/crpp/command?args`
2) All commands normally return *json*, but are able to return *xml* if the user would like this. In that case `&outputformat=xml` needs to be appended to the request.
3) The current (dec 2015) server address is:
   `corpus-studio-web.cttnww-meertens.vm.surfsara.nl`.
   This is a virtual CentOS Linux computer in the cloud, hosted by SurfSara.

4.1 Unified response structure

The commands issued to `/crpp` have a unified response structure consisting of three parts: "indexName", "contents" and "status".

```
{ "indexName": "serverinfo",
  "contents": {…  },
  "status": {
    "code": "completed",
    "message": "Some kind of message",
    "userid": "tomcat"
  }
}
```

The "indexName" part copies the name of the index that has been called (the specific /crpp command). The "contents" part differs per command. The "status" part at the very least contains a "code", and this 'code' can only have a limited number of values: "completed", "working" or "error". If there is an error, then the status contains the 'error' value for 'code', and a message may be available in the 'content' part, e.g:

```
{ "content": {
    "code": "INTERNAL_ERROR",
    "message": "RequestHandler is empty. Use: /execute, /show, /statusxq"
  },
  "status": {"code": "error"}
}
```

4.2 Issuing "crpchg"

The query command `/crpchg`, like all other commands, takes a json datastructure as argument. The datastructure may be of two types: defining one key/id/value change, or a list of changes. Here is an example of one change:

```
/crpchg?{     "userid":"erkomen",
              "crp": "ParticleA.crpx",
              "key": "Goal",
              "id": -1,
              "value": "This CRP serves as an example" }
```

The change is for the CRP called ParticleA.crpx belonging to "erkomen". It involves the field "Goal" of the CRP as a whole (hence the -1 id), and the "value" part gives the new definition of the Goal.

Here is an example of a list of changes:

```
/crpchg?{     "crp":"tstSonar",
              "list":"eJy...",
              "userid":"erwin"}
```

The "list" component is compressed and then Base64 encoded (with slightly adapted non-alpha symbols). Un-escaping the list shows:

```
[{"id":3, "value":"","key":"query.create"},
{"id":3, "value":"matObj","key":"query.Name"},
{"id":3, "value":"Find main clauses with an object","key":"query.Goal"},
{"id":3, "value":"Find main clauses with at least one object on the clause
level","key":"query.Comment"},
{"id":3, "value":"<FoLiA>{\n (: Loop through the elements of each text :)\n for
$search in //su[ru:matches(@class,'undefined')]\n \n  (: Retrieve possible
subject :)\n  let $sbj := $search/child::su[ru:matches(@class,'undefined')]\n
\n  (: subject must exist :)\n  where (\n    exists($sbj)\n  )\n  \n (: Return
results :)\n return ru:back($search)\n}<\/FoLiA>","key":"query.Text"},
{"id":3, "value":"","key":"constructor.create"},
{"id":3, "value":"2/out","key":"constructor.Input"},
{"id":3, "value":"matObj","key":"constructor.Query"},
{"id":3, "value":"3_matObj","key":"constructor.Output"},
{"id":3, "value":"matObj","key":"constructor.Result"},
{"id":3, "value":"False","key":"constructor.Cmp"},
{"id":3, "value":"False","key":"constructor.Mother"},
{"id":3, "value":"Find main clauses with an object","key":"constructor.Goal"},
{"id":3, "value":"Find main clauses with at least one object on the clause
level","key":"constructor.Comment"}]
```

What is shown here is the definition of a new query in a Folia-xml project. The first change in the list is not a change at all, but the key "`query.create`" indicates that it involves the creation of a new query with default values. All the changes belong to the CRP with CrpId equal to "3" (the name of the CRP has been defined in the "crp" parameter outside of the list).

The first part of the list, then, involves the definition of a new "query" (its Name, Goal and Text), while the second part specifies that this query should also be taken up in the "constructor".

Any /crpchg command that leads to changes in the CRP leads to follow-up actions: (a) the general section of the CRP gets an adapted date-stamp, (b) the changed CRP is saved on the

/crpp server, and (c) the /crpstudio issues a command to get the changed CRP into its local storage.

4.3 Issuing "crpdel"

The command `/crpdel` requires specification of the item to be deleted. What needs to be specified is the name of the CRP and the userid. Optional specifications are "itemid", "itemtype" and "itemmain". These parameters are *not* used for the deletion process.

```
/crpdel?{     "crp": "VladimirVelikij",
              "itemid": "2",
              "itemtype": "project",
              "itemmain": "ROOT",
              "userid": "erwin"   }
```

Successful deletion of a CRP leads to a confirmational response, which repeats the name of the CRP as well as the location from where it was deleted on the server:

```
{ "indexName": "crpdel",
  "content": {
    "file": "/etc/project/erwin/SbjProperties_V9.crpx",
    "name": "SbjProperties_V9.crpx"
  },
  "status": {
    "code": "completed",
    "message": "See the [crp] in the [content] section",
    "userid": "tomcat"
  }
}
```

4.4 Issuing "crpget"

The command `/crpget` only takes the userid and the name of the CRP as arguments (and the CRP name may be with or without the ending .crpx):

```
/crpget?{     "userid":"erkomen",
              "name": "ParticleA.crpx"}
```

The whole CRP is returned in a "crp" part of the "content":

```
{ "indexName": "crpget",
  "content": {
    "crp": "<?xml ...\n</CorpusResearchProject>"
  },
  "status": {
    "code": "completed",
    "message": "See the [crp] in the [content] section",
    "userid": "tomcat"
  }
}
```

4.5 Issuing "crpinfo"

The command `/crpinfo` is a way to request date/time information on a particular CRP. It takes three arguments: the "userid", the name of the "crp" and the kind of date required: either "modified" or "dateChanged" (but the outcome should be the same). Here is an example:

```
/crpget?{     "userid":"erkomen",
              "crp": "txtSonar.crpx",
              "info": "modified"}
```

The requested date/time information is returned in the "content" part:

```
{ "indexName": "crpinfo",
      "content": {
        "modified": "2015-12-28T17:12:35"
      },
      "status": {
        "code": "completed",
        "message": "See the information in the [content] section",
        "userid": "tomcat"
      }
    }
```

4.6 Issuing "crplist"

The command `/crplist` returns a list of the CRPs for one particular user (if the "userid" is specified) or for all users (if an empty parameter is provided). The command `/crplist?{"userid": "guest"}`, for instance, returns:

```
    { "indexName": "crplist",
      "content": [
        { "CrpId": 2,
          "userid": "guest",
          "crp": "ChechenMarkerA.crpx",
          "loaded": false,
          "size": 126060,
          "file": "/etc/project/guest/ChechenMarkerA.crpx",
          "lng": "che_lat",
          "dir": "NPCMC"
        },
        { "CrpId": 3,
          "userid": "guest",
          "crp": "DutchIOstart.crpx",
          "loaded": false,
          "size": 4044,
          "file": "/etc/project/guest/DutchIOstart.crpx",
          "lng": "nld",
          "dir": "WR-P-P-K_reports"
        },
        { "CrpId": 1,
          "userid": "guest",
          "crp": "LakProDem.crpx",
          "loaded": false,
          "size": 30699,
          "file": "/etc/project/guest/LakProDem.crpx",
          "lng": "lak_cyr",
          "dir": "PCMLBE"
        },
        { "CrpId": 4,
          "userid": "guest",
          "crp": "Possessives.crpx",
          "loaded": false,
          "size": 167044,
          "file": "/etc/project/guest/Possessives.crpx",
          "lng": "che_lat",
          "dir": "NPCMC"
        }
      ],
      "status": {
        "code": "completed",
        "message": "See the list of CRPs in the [content] section",
        "userid": "tomcat"
      }
    }
```

The CRPs are automatically numbered by /crpp, but the numbers may differ as soon as more CRPs are added. Each CRP has some additional information, like its size (in number of bytes) and whether it has been loaded into the CrpManager.

4.7 Issuing “crpset”

The `/crpset` serves for uploading a CRP to the server. Required parameters are the userid, the name of the CRP, the text of the CRP (compressed and in Base64) and the 'overwrite' parameter. Issuing the command looks like this:

```
/crpset?{     "userid":"erwin",
              "crp": "eJzVvW2THMdxLvrZitB/mIPrEABeAdv1X",
              "name": "Dutch-ditransitives",
              "overwrite": "true" }
```

The overwrite parameter resolves the case where a CRP with the indicated name already exists. What the /crpset command returns in the "content" part is the "name" of the CRP:

```
{     "indexName": "crpset",
      "content": {
        "name": " Dutch-ditransitives"
      },
     "status": {
        "code": "completed",
        "message": "The crp has been stored at the server",
        "userid": "tomcat"
      }
}
```

4.8 Issuing “dbinfo”

This command provides detailed information about results stored in a result database. The kind of information to be provided depends on the paramters being passed on. The general layout of the command is:

```
/dbino?{      "userid":"erwin",
              "name":  "omdat_only_QC1_Dbase",
              "start": 0,
              "count": 10 }
```

Note that ‘start’ is zero-based, even though the result number (id) starts with “1”. The parameter ‘count’ may be 0, in which case no information but the size of the database is returned.

What is returned has the following format:

```
{ "indexName": "dbinfo",
  "content": {
    "General": {
      "ProjectName": "omdat_only",
      "Created": "2017-10-24T08:50:24",
      "Language": "nld", "Part": "LassyKlein",
      "Notes": "Created by CorpusStudio (web) from query line 1: [standard]",
      "Analysis": "searchWord;searchPOS",
      "QC": 1,
      "Features": [ "ft_searchWord", "ft_searchPOS" ]
    },
    "Count": 1, "Size": 5380,
    "Results": [
      { "ResId": 1,
        "File": "dpc-bmm-001072-nl-sen.folia.xml",
        "TextId": "dpc-bmm-001072-nl-sen",
        "Search": "dpc-bmm-001072-nl-sen.d.1.p.1.s.13",
        "Cat": "doordat_groep",
        "Locs": "dpc-bmm-001072-nl-sen.d.1.p.1.s.13",
        "Locw": "dpc-bmm-001072-nl-sen.d.1.p.1.s.13.su.20",
        "Notes": "-", "SubType": "", "Text": "",
        "Psd": "", "Pde": "",
        "Features": [ { "ft_searchWord": "door" },
                      { "ft_searchPOS": "VZ" }
        ]
      }
    ],
    "Features": [ "ft_searchWord", "ft_searchPOS" ]
  },
  "status": {
    "code": "completed",
    "message": "See the information in the [content] section",
    "userid": "tomcat"
  }
}
```

4.9 Issuing "dblist"

It is possible to get a list of all the databases available to one user by issuing a `/dblist{"userid": "name_of_user"}` command. The return looks like this:

```
    { "indexName": "dblist",
      "content": [
        { "userid": "guest",
          "dbase": "ChechenMarkerA_QC1_Dbase.xml",
          "file": "/etc/project/guest/dbase/ChechenMarkerA_QC1_Dbase.xml",
          "lng": "che_lat",
          "dir": "NPCMC"
        },
        { "userid": "guest",
          "dbase": "tstSonar_QC2_Dbase.xml",
          "file": "/etc/project/guest/dbase/tstSonar_QC2_Dbase.xml",
          "lng": "nld",
          "dir": "WR-P-P-K_reports"
        }
      ],
      "status": {
        "code": "completed",
        "message": "See the list of data bases in the [content] section",
        "userid": "tomcat"
      }
    }
```

Note that the contents is a JSONArray of structures that provide the database short name, its path and the language/corpus input on which it was executed.

4.10 Issuing "dbset"

This command is used to upload a result database to the /crpp server. The database that is being uploaded must be in the *xml* format of result databases. The /dbset command has the following structure:

```
    /dbset?      {"userid": "erwin",
                 "name": "ParticleA_Dbase.xml",
                 "db": "<Result>…</Result>",
                 "lng": "nld",
                 "dir": "ParsedSonar",
                 "overwrite": true}
```

Note that the "lng" and "dir" parameters are optional; they can be left out completely. The database itself (the argument of "db") must be provided in the CorpusStudio-standard compressed format (gzipped, then base64 with the plus replaced by a tilde).

The result upon success:

TODO – result example

4.11 Issuing "dbupload"

The `/dbupload` command is used to upload a databased divided into chunks to the `/crpp` server. This command is mainly used by the `/crpstudio` server. A number of actions can be triggered by specifying the ‘action’ parameter of the command. The general layout of the command is this:

```
    /dbupload?    {"userid": "erwin",
                  "name": "ParticleA_Dbase",
                  "action": "init",
                  "chunk": 20,
                  "total": 100,
                  "overwrite": true}
```

The ‘chunk’ paramater provides the number of this chunk, and the ‘total’ – the total number of chunks to be expected. The arguments that should be provided depend on the action:

action	arguments needed
init	userid, name, total
stop	userid, name, total
send	userid, name, total, chunk, start

TODO – result example

4.12 Issuing “debug”

The /debug command does not expect any parameters and its return should look like this:

```
{ "indexName": "debug",
  "status": {
    "code": "completed",
    "message": "The Java-part of the CRPP service works fine.",
    "userid": "tomcat"
  }
}
```

Note that this is the only command whose return misses a "content" part.

4.13 Issuing “exe”

The /exe command takes four obligatory parameters in the JSON object: "crp", "userid", "lng" and "dir":

```
    /exe?         {"lng": "nld",
                  "dir": "WR-P-E-C_e-magazines",
                  "crp": "tstSonar.crpx",
                  "userid": "erwin",
                  "cache": false}
```

The fifth argument "cache" is optional. It specifies whether to use any previous results in cache ('true') or not ('false'). What the initial call to /statusxq returns is a single status block without a content part:

```
{ "status": {
    "code": "started",
    "message": "Searching, please wait...",
    "userid": "erwin",
    "jobid": "1",
    "checkAgainMs": 200
  }
}
```

The information provided is valuable: the code "started" indicates that the search job has been accepted, and the "jobid" value of '1' tells the user which jobid needs to be used for subsequent status requests.

4.14 Issuing “reset”

Stopping a job that has been started can be done by issuing a `/reset` command that takes just two arguments: the `userid` and the `jobid`. This `jobid` can be gleaned from the initial response on the `/exe` command. Here is an example of a status request:

```
/reset?{      "userid":"erwin",
              "jobid": "165"}
```

The json fields are “`userid`” and “`jobid`”. The “`userid`” must be the same one as has been used to issue the `/exe` command. The “`jobid`” field must contain the **string** value (a number between quotation marks) of the `jobid` that has been received from the *first* reply on the `/exe` command.

The "content" part that is returned from a `/reset` job contains two elements: "`action`", which should have the value "`aborted`", and "`finished`", which receives the internal boolean value of the job's finish status, so must ideally be `'true'`.

4.15 Issuing “serverinfo”

The `/serverinfo` command takes no arguments at all. What gets returned is a list of available languages as well as the contents of the crp-info.json file (as a string):

```
{"indexName": "serverinfo",
 "contents":{
   "indices":["eng_hist","lak_cyr","eng_sla","nld","che_lat"],
   "corpora": "{...}"},
"status": {
  "code":"completed",
  "message":"See the 'indices' information in @indices and the 'corpora'
    information in @corpora",
  "userid":"tomcat"}
}
```

4.16 Issuing “settings”

The `/settings` command only takes a "userid" argument:

```
/settings?{  "userid":"erkomen"}
```

What is returned is a list of links:

```
{ "indexName": "settings",
  "content": {
    "links": [
      { "crp": "ChechenMarkerA.crpx", "lng": "che_lat", "dir": "NPCMC" },
      { "crp": "LakProDem.crpx", "lng": "lak_cyr", "dir": "PCMLBE" }
    ],
    "userid": "guest",
    "recent": "V2_MEdbCollect_versie8.crpx"
  },
  "status": {
    "code": "completed",
    "message": "See the settings object in the [content] section",
    "userid": "guest"
  }
}
```

The `/settings` command also returns the name of the CRP that has been executed last in the "recent" parameter.

4.17 Issuing "statusxq"

Requesting the progress of an /exe command is done by issuing a /statusxq command that takes just two arguments: the userid and the jobid. This jobid can be gleaned from the initial response on the /exe command. Here is an example of a status request:

```
/statusxq?{   "userid":"erwin",
              "jobid": "165"}
```

The json fields are "userid" and "jobid". The "userid" must be the same one as has been used to issue the /exe command. The "jobid" field must contain the **string** value (a number between quotation marks) of the jobid that has been received from the *first* reply on the /exe command. This first reply contains the status "started", and it is only this reply that returns the correct jobid value.

What the /statusxq returns depends on the progress that is being made. Halfway through the process one might get the following update:

```
{ "indexName": "statusxq",
  "content": {
    "jobid": "165",
    "start": "WR-P-P-K-0000000073.folia",
    "finish": "WR-P-P-K-0000000065.folia.xml",
    "count": 62,
    "total": 81,
    "ready": 44
  },
  "status": {
    "code": "working",
    "message": "please wait",
    "userid": "erwin"
  }
}
```

The parameters that are returned in the "content" part are the following:

start The last text that has been sent as execution job
finish The most recent text that has completed execution
count The number of texts sent in for execution
total The total number of texts that need to be executed
ready The number of texts that have returned from execution

At the end of the execution the /statusxq command does not only return the status "completed", but it also contains a table with a summary of the results found:

```
{ "indexName": "statusxq",
  "content": {
    "jobid": "165",
    "searchParam": { "resultsType": "json",
      "tmpdir": "/var/cache/tomcat/temp",
      "waitfortotal": "no"
    },
    "searchTime": 7478, "searchDone": true,
    "query":
    "{\"crp\":\"ChechenMarkerA.crpx\",\"lng\":\"che_lat\",\"save\":\"2015-12-10
    v13:13:50\",\"dbase\":\"\",\"dir\":\"NPCMC\",\"userid\":\"guest\"}",
    "taskid": 2,
    "table": [
      { "qc": 1,
        "result": "anyParticleA",
        "subcats": [],
        "counts": [],
        "total": 508,
        "hits": [
          { "file": "Arsanukaev1-2013.psdx", "message": [], "count": 106,
            "subs": [] },
          { "file": "Arsanukaev2-2013.psdx", "message": [], "count": 0,
            "subs": [] }
        ]
      }
    ],
    "total": 71
  },
  "status": {
    "code": "completed",
    "message": "The search has finished",
    "userid": "erwin"
  }
}
```

4.18 Issuing "statusxl"

Requesting the progress of an /txtlist command is done by issuing a /statusxl command that takes just two arguments: the userid and the jobid. This jobid can be gleaned from the initial response on the /txtlist command. Here is an example of a status request:

TODO: example

4.19 Issuing "txt"

The /txt command serves two purposes, both related to a *text* that has been specified in terms of (a) language, (b) directory, (c) encoding type (either 'folia' or 'psdx') and (d) text name. When the 'type' argument is set to 'sentences', a list of all the sentences in the text will be supplied. Other types are: "grouping", "hits", "context", "msg", "syntax", "svg", and these types are completely the same as those used for the /update command (see there for an explanation of them).

4.20 Issuing "txtlist"

The /txtlist command can be used to get a list of available texts, but there are a number of obligatory parameters: (a) the language of the texts, (b) the sub-directory in which to search, and (c) the the encoding type of the texts (either 'folia' or 'psdx').

The 'content' part of the txtlist command contains a parameter 'jobid' that can be used to probe the status of the txtlist.

4.21 Issuing "update"

The /update command is used to download results from a job that has been executed. The output of the job is stored on the server and it is this stored output that is fetched by the /update command. Here is a request for some information:

```
/update?{    "crp":"V2_test_versie11",
             "sub":"1:[iSbj-iVf]",
             "qc":2,
             "lng":"eng_hist",
             "start":1,
             "count":10,
             "files":[],
             "type":"context_syntax",
             "dir":"ME",
             "userid":"erwin"}
```

The obligatory parts of the /update command are:
userid - name of the user

`crp` - name of the CRP
`lng` - language on which the CRP has been executed
`start` - index of the first hit to be fetched
`count` - number of hits to be fetched
`qc` - the constructor line (from the constructor editor) for which we want information
`type` - the required info type: `'hits'`, `'context'`, `'msg'`, `'syntax'`
or a combination separated by an underscore '_'

Then there are some optional parameters:
`dir` - part of the corpus on which the CRP has been executed
`sub` - the sub category to which the results need to be confined
`files` - an array of files to which the results must be confined
`div` - the Xquery code that must be used to map each text to a group label

What the `/update` returns depends on the type of information that has been requested. Usually the "content" part of the response consists of an array of results.

```
{ "indexName": "update",
  "content": [
    { "n": 1,
      "file": "WR-P-P-K-0000000001.folia.xml",
      "locs": "WR-P-P-K-0000000001.head.1.s.2",
      "locw": "WR-P-P-K-0000000001.head.1.s.2.su.1",
      "preC": "[WR-P-P-K-0000000001.head.1.s.1] Energiebesparende maatregelen in
    de woningvoorraad [WR-P-P-K-0000000001.head.1.s.2] ",
      "hitC": "KWR 2000 maakt balans op",
      "folC": " [WR-P-P-K-0000000001.head.2.s.1] Voorwoord",
      "allS": {
        "main": "SMAIN",
        "children": [
          { "pos": "MWU-SU", "txt": "KWR 2000" },
          { "pos": "WW", "txt": "maakt" },
          { "pos": "N-OBJ1", "txt": "balans" },
          { "pos": "VZ-SVP", "txt": "op" }
        ]
      },
      "hitS": {
        "main": "SMAIN",
        "children": [
          { "pos": "MWU-SU", "txt": "KWR 2000" },
          { "pos": "WW", "txt": "maakt" },
          { "pos": "N-OBJ1", "txt": "balans" },
          { "pos": "VZ-SVP", "txt": "op" }
        ]
      }
    },
    {
      "n": 2,
      "file": "WR-P-P-K-0000000001.folia.xml",
      "locs": "WR-P-P-K-0000000001.p.1.s.1",
      "locw": "WR-P-P-K-0000000001.p.1.s.1.su.1",
      "msg": "part of me",
      "preC": "[WR-P-P-K-0000000001.head.1.s.2] KWR 2000 maakt balans op[WR-P-P-
    K-0000000001.head.2.s.1] Voorwoord [WR-P-P-K-0000000001.p.1.s.1]
    Nederlandse",
      "hitC": "De isolatiegraad van de gemiddelde Nederlandse woning nam in vijf
    jaar met tien procent toe .",
      "folC": " [WR-P-P-K-0000000001.p.1.s.2] Het aantal hoogrendementsketels
    verdubbelde in dezelfde tijd.",
      "allS": {
        "main": "SMAIN",
        "children": [
          { "pos": "NP-SU",
            "txt": "De isolatiegraad van de gemiddelde Nederlandse woning"
          },
          { "pos": "WW", "txt": "nam" },
          { "pos": "PP-MOD", "txt": "in vijf jaar" },
          { "pos": "PP-MOD", "txt": "met tien procent" },
          { "pos": "VZ-SVP", "txt": "toe" },
          { "pos": ".", "txt": "" }
        ]
      },
      "hitS": {
        "main": "SMAIN",
        "children": [
          { "pos": "NP-SU",
            "txt": "De isolatiegraad van de gemiddelde Nederlandse woning"
          },
          { "pos": "WW", "txt": "nam" },
          { "pos": "PP-MOD", "txt": "in vijf jaar" },
          { "pos": "PP-MOD", "txt": "met tien procent" },
          { "pos": "VZ-SVP", "txt": "toe" },
          { "pos": ".", "txt": "" }
        ]
      }
    }
  ],
  "status": {
    "code": "completed",
    "message": "See the information in the [content] section",
    "userid": "tomcat"
  }
}
```

Each result element has a number '`n`', and contains information that localizes the place where the result comes from: the name of the text `'file'`, the sentence id within the file `'locs'` and the constituent id within the sentence `'locw'`.

Next comes the context information, divided into `'preC'` (preceding context), `'hitC'` (the part of the hit) and `'folC'` (following context). The example above also contains syntax information in the form of `'allS'` (syntax of the whole clause) and `'hitS'` (syntax of the constituent that was returned by the query). The example above has the `allS` and `hitS` equal.

The results for 'grouping' have not been determined yet.

5 Web service maintenance

5.1 Adapt CorpusStudio for use with other corpora

Adapting the CorpusStudio web application for one's own corpora is possible, provided the application is run from one's own Linux server.

5.1.1 Requirements

- **Server**. Linux server. The program has been developed under CentOS, but it should be possible to use other Linux versions.
- **Space**. The server should have enough space to hold one's corpora as well as CorpusStudio and the engine it runs on (Java + Tomcat). The program uses space on the hard drive under the /etc/project directory as well as under its deployment directory.
- **Memory**. The amount of RAM memory needed is roughly 1 Gbyte per processor plus perhaps 20 Gbyte to start with. The amount can be determined experimentally by creating a CRP that searches through the maximum number of input files (it's the number of files that counts, not so much their size), and then watching the amount of 'free memory left' in the cataline log file. E.g: `sudo tail -n 400 /usr/share/tomcat/logs/catalina.out | grep "Free m"`. Note: changes in the amount of memory that is available do *not* automatically become 'known' to Tomcat. The amount of memory that should be used by Tomcat must be specified by adding or adapting the JAVA_OPTS line in the tomcat.conf file. The CentOS server CLARIN's application has been built on has this file in the `/usr/share/tomcat/conf/` directory.
- **Processor**. The speed at which the web application will search through texts increases as more processing cores are available. But note that each processing core also increases the amount of RAM memory needed. Something like 8-20 cores should get a nice speed.
- **Software**. Java (version 1.7 or higher) and tomcat (version 7 or higher) should be installed on the server.

5.1.2 Installation

The installation of the CorpusStudio web application is Linux-system (and version) dependent. A virtual machine that has a CentOS system containing no installation of Java would require the following commands to prepare the machine for CorpusStudio:

```
# Arrange for sudo access
# - Add a user account for the person in charge (e.g "crphome")
# - Log in as root: su
# - Edit the /etc/sudoers file:
#   add a line for 'crphome' that has the same priorities as 'root'
# Changes to the principal's user account: get a directory 'webapps'
cd ~
mkdir webapps
#
# Update to latest installations
sudo yum update
# Install epel-release
sudo yum install -y epel-release
# Manual tweaking of epel-release:
sudo wget http://dl.iuscommunity.org/pub/ius/stable/RedHat/6/x86_64/ius-release-
     1.0-13.ius.el6.noarch.rpm
sudo rpm -Uvh ius-release*.rpm
# Install Java
```

```
sudo yum install -y java
# Install tomcat
sudo yum install -y tomcat
# Check the correct paths for Java, and create (or adapt) the file .bash_profile in
     the directory /home/crpstudio:
     # .bash_profile
     # Get the aliases and functions
     if [ -f ~/.bashrc ]; then
             . ~/.bashrc
     fi
     # User specific environment and startup programs
     PATH=$PATH:$HOME/bin:/usr/share/alpino/bin
     JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.45-28.b13.el6_6.x86_64/jre
     JRE_HOME=$JAVA_HOME
     PATH=$JAVA_HOME/bin:$PATH:$HOME/bin
     CATALINA_HOME=/usr/share/tomcat
     # Make the variables available
     export JAVA_HOME
     export JRE_HOME
     export PATH
# Adapt the file “ajp.conf” in directory /etc/httpd/conf.d so that it contains the
     lines with “crpp” and “crpstudio”:
     ProxyRequests Off
     <Proxy *>
             Order deny,allow
             Deny from none
             Allow from localhost
     </Proxy>
     ProxyPass           /crpp   ajp://localhost:8009/crpp
     ProxyPassReverse    /crpp   ajp://localhost:8009/crpp
     ProxyPass           /crpstudio    ajp://localhost:8009/crpstudio
     ProxyPassReverse    /crpstudio    ajp://localhost:8009/crpstudio
# Go to directory /usr/share/tomcat/conf/Catalina
cd /usr/share/tomcat/conf/Catalina
# Create a file crpp.xml
     <?xml version='1.0' encoding='utf-8'?>
     <Context docBase="/home/crphome/webapps/CrppS" path="/crpp" reloadable="true"
          />
# Create a file crpstudio.xml
     <?xml version='1.0' encoding='utf-8'?>
     <Context antiJARLocking="true" docBase="/home/crphome/webapps/CrpStudio"
          path="/crpstudio" reloadable="true" />
# Adapt the existing file server.xml in directory /usr/share/tomcat/conf, adding a
     ‘localhost2’ section after the existing ‘localhost’ one (depending on the main
     user’s home directory; this makes sure the *.war files in the user’s webapps
     directory are unpacked properly):
      <Host name="localhost2"  appBase="/home/crphome/webapps"
            unpackWARs="true" autoDeploy="true">
        <!-- SingleSignOn valve, share authentication between web applications
             Documentation at: /docs/config/valve.html -->
        <!--
        <Valve className="org.apache.catalina.authenticator.SingleSignOn" />
        -->
        <!-- Access log processes all example.
             Documentation at: /docs/config/valve.html
             Note: The pattern used is equivalent to using pattern="common" -->
        <Valve className="org.apache.catalina.valves.AccessLogValve"
     directory="logs"
               prefix="nederlab_access_log." suffix=".txt"
               pattern="%h %l %u %t &quot;%r&quot; %s %b" />
      </Host>
# Adapt the lines for JAVA_HOME and JAVA_OPTS in the file tomcat.conf that resides
     in directory /usr/share/tomcat/conf (this says Java uses 45 Gbytes maximum and
     1Gbytes minimum):
     # Where your java installation lives
     JAVA_HOME="/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.45-28.b13.el6_6.x86_64/jre"
     # You can pass some parameters to java here if you wish to
     JAVA_OPTS="-Xmx45G -Xms1G"
```

```
# add user crphome to the 'tomcat' group
sudo usermod -a -G tomcat crphome
#
# log off and log in again as crphome to let the changes become effective
#
# Create a directory structure that at least contains:
# /etc/corpora      - this must contain file crp-info.json
# /etc/project      - this must contain a directory for each user
#                     Crpstudio will create a settings.json file
#                       inside each user's directory
# /etc/crpstudio    - this must contain a directory for each user
#                     it must also contain crpstudio-settings.json
#
# Make sure the directories created are owned by tomcat:tomcat
#
# Make sure the crp-info.json file is adapted to one's use and is validated
```

5.1.3 *File crp-info.json*

The file 'crp-info.json' must reside in the server's /etc/corpora directory. Here is a sample of its possible contents. It contains a definition for one corpus (the Lak language) and all its associated information.

```
{ "corpora": [
    {"lng": "lak_cyr", "name": "Lak (lbe), Cyrillic orthography", "eth": "lbe",
     "metavar": "caucasian",
     "parts": [
        {"name": "PCMLBE", "dir": "PCMLBE", "descr": "Parsed Corpus of Modern Lak",
                "metavar": "caucasian", "psdx": "", "folia": "",
                "url": "http://erwinkomen.ruhosting.nl/lbe/crp/"}

     ]}
  ],
  "metavar": [
    {"name": "caucasian",
     "variables": [
      {"name": "author",    "descr": "Author",                       "type": "txt",
                "loc": "header", "value": "descendant::titleStmt/@author"},
      {"name": "date",      "descr": "Estimated manuscript date",    "type": "int",
                "loc": "header", "value": "descendant::creation/@manuscript"},
      {"name": "editor",    "descr": "Editor",                       "type": "txt",
                "loc": "header", "value": "descendant::titleStmt/@editor"},
      {"name": "genre",     "descr": "Type of text",                 "type": "txt",
                "loc": "header", "value": "descendant::creation/@genre"},
      {"name": "subtype",   "descr": "Time/genre/translated combi",  "type": "txt",
                "loc": "header", "value": "descendant::creation/@subtype"},
      {"name": "title",     "descr": "Title of document",            "type": "txt",
                "loc": "header", "value": "descendant::titleStmt/@title"},
      {"name": "translated","descr": "Is this translated?",          "type": "txt",
                "loc": "header", "value": "if (ends-
                with(descendant::creation/@subtype, 't')) then 'true' else 'false'"}
     ],
     "tagset": [
        {"title": "clsAny", "def": "IP*"},
        {"title": "clsMain","def": "IP-MAT*"},
        {"title": "clsSub", "def": "IP-SUB*"},
        {"title": "clsInf", "def": "IP-INF*"},
        {"title": "npSbj",  "def": "NP-SBJ*"},
        {"title": "npObj",  "def": "NP-OB*"},
        {"title": "npAny",  "def": "NP|NP-*"},
        {"title": "ppAny",  "def": "PP|PP-*"},
        {"title": "vbAny",  "def": "V*|AX*"},
        {"title": "vbFin",  "def": "VB[PD]*|AX*"}
     ],
     "groupings": [
       {"name": "subtype", "descr": "Combination of time and translated",
                "value": "$subtype"},
       {"name": "titleAlphabet", "descr": "First letter of title",
                "value": "if ($title = '') then '(n.t.)' else substring($title, 0,
                1)"},
       {"name": "authorAlphabet", "descr": "First letter of author surname",
                "value": "if ($author = '') then '(n.a.)' else substring($author, 0,
                1)"},
       {"name": "date", "descr": "Year of manuscript",
                "value": "$date"}
     ]
    }
  ]
}
```

The essential sections from the crp-info.json file:

- **corpora**. This defines key ingredients used by CorpusStudio:
    - **lng** the language code. This must coincide with a directory name: `/etc/corpora/lak_cyr`
    - **metavar** the name of the set of variables used for input restriction and grouping
    - **parts** one or more parts of the corpus, as identifiable by the sub directory (the attribute ‘`dir`’) within the corpus directory. Here: `/etc/corpora/lak_cyr/PCMLBE`
- **metavar**. This provides common variable names and their Xquery definitions.
    - **name** The name of this section. It must correspond with the value of the ‘`metavar`’ attribute in the ‘`corpora`’ section.
    - **variables** A list of all the variables that can be used in the definition of input restrictions as well as groupings. It is the variable names defined here that occur in the selection box for the input restriction definition of the Projects/Input page of CorpusStudio. Only the variables that are defined here may occur later in the ‘`groupings`’ section.
    - **tagset** This is, in the current version of CorpusStudio, a fixed list of names and the labels (POS or class tags) that are to be associated with them. The labels are generally defined by a ‘def’ specifiction. Where such a specification is not enough, a ‘`fs`’ specification may specify the name and the value of a feature. The ‘`vbFin`’ definition for ‘sonar’ files, for instance, is defined as: `{"title": "vbFin",     "def": "WW*", "fs": [{"alp/wvorm": "pv"}]}`
    - **groupings** The idea of result-groupings is in the process of being coded in CorpusStudio. The indicated section is a sample of the standard groupings every ‘`metavar`’ section must provide. The ‘`value`’ part of each grouping is an Xquery expression that results into a string label that identifies the ‘result group’ one particular file belongs to.

5.1.4 *File crpstudio-settings.json*

The directory /etc/crpstudio must contain a file named crpstudio-settings.json with the user/password information for the users that are allowed to make use of the version of CorpusStudio being used.

```
{ "users": [
    {"name": "erwin",    "password": "xx", "admin": true,  "include": []},
    {"name": "erkomen",  "password": "xx", "admin": false, "include": []},
    {"name": "guest",    "password": "xx", "admin": false, "include": []}
  ]
}
```

5.1.5 *File crpp-settings.json*

The /crpp ‘machine’ is the query-crunching motor serving the CorpusStudio web application. Its parameters are specified by the file crpp-settings.json that should be located in the main user’s webapps directory (that would be `/home/crphome/webapps` in the running example). The format and contents of this settings file have been borrowed from the BlackLab code.[1] Here is a sample of its contents.

```
{ // Corpus Research Project Processor config file
  // ===============================================================
  // A list of IPs that will run in debug mode.
  "debugModeIps": [
    "127.0.0.1",      // IPv4 localhost
    "0:0:0:0:0:0:0:1" // IPv6 localhost
  ],

  // Specify the maximum number of "XqF" jobs allowed
  "maxparjobs": 18,

  // List of important directories
  "projectBase": "/etc/project",
  "corpusBase": "/etc/corpora",

  // A list of possible Engines, with the default engine identified
  "pinfo": {
    "ProjectType": "Xquery-psdx",
    "Xquery-psdx": {
      // Description
      "Descr": "Xquery with XML output",
      // Extension of query files
      "Qext": ".xq",
      // Extension of definition files
      "Dext": ".xq",
      // Extension of source files
      "SrcExt": ".psdx",
      // Start and end of comments
      "ComBeg": "(:", "ComEnd": ":)",
      // The 'engine' that is being used
      "Engine": "Xquery",
      // Default definition and period file locations
      "DefaultDefFile": "",
      "DefaultPerFile": "",
      // Text of default query
      "DefaultQuery": [
        "<TEI>",
        "{",
        " for $search in //eTree[ru:matches(@Label, $_matrixIP)]",
        "",
        "  (: Use your own 'let' definitions here :)",
        "  let $sbj := tb:SomeChildNo($search, $_subject, $_nosubject)",
        "",
        "  (: Define your 'where' definition here :)",
        "  where ( exists($sbj)",
        "         )",
        " return ru:back($search)",
        "}",
        "</TEI>"]
    },
    "FoLiA-xml": {
      // Description
      "Descr": "Xquery with XML output",
      // Extension of query files
      "Qext": ".xq",
      // Extension of definition files
      "Dext": ".xq",
      // Extension of source files
      "SrcExt": ".folia.xml",
      // Start and end of comments
      "ComBeg": "(:", "ComEnd": ":)",
      // The 'engine' that is being used
```

[1] Blacklab is a program written by INL. See: https://github.com/INL/BlackLab.

```
      "Engine": "Xquery",
      // Default definition and period file locations
      "DefaultDefFile": "",
      "DefaultPerFile": "",
      // Text of default query
      "DefaultQuery":  [
        "<FoLiA>",
        "{",
        "  (: Look for all main clauses within this sentence :)",
        "  for $search in //su[ru:matches(@cat, 'smain')]",
        "",
        "  (: Get the subject of this particular main clause :)",
        "  let $sbj := $search/child::su[@rel='su']",
        "",
        "  (: Do not allow empty (elided) subjects :)",
        "  where ( ",
        "           exists($sbj) and (count($sbj/child::su)>0)",
        "        )",
        "  (: Return the main clause :) ",
        "  return ru:back($search)",
        "}",
        "</FoLiA>"]
    }
  },
  // The location and parameters for each language-corpus index
  // ---------------------------------------------------------------
  // (missing indices will be skipped)
  "indices": {
    "lak_cyr": {
      "dir": "/etc/corpora/lak_cyr",
      "mayViewContent": true
    }
  },
  // Settings related to tuning server load and client responsiveness
  // ---------------------------------------------------------------
  "performance": {

    // Settings for job caching.
    "cache": {
      // How many search jobs will we cache at most?
      "maxNumberOfJobs": 200,

      // After how much seconds will a search job be removed from the cache?
      "maxJobAgeSec": 3600,

      // How many MB free memory the cache should aim for while cleaning up.
      "targetFreeMemMegs": 500,

      // When there's less free memory available than targetFreeMemMegs,
      // each time a job is created and added to the cache,
      // get rid of this number of older jobs in order
      // to free up memory
      "numberOfJobsToPurgeWhenBelowTargetMem": 2
    },

    // The minimum amount of free memory required to start a new search job.
    "minFreeMemForSearchMegs": 100,

    // The maximum number of jobs a user is allowed
    // to have running at the same time.
    "maxRunningJobsPerUser": 100
    }
  }}
```

5.1.6 *Software sources*

The source for the Java/JavaScript software that needs to be compiled consists of three parts, all of which can be downloaded from https://github.com/ErwinKomen:

1) CrpxProcessor - The Xquery search ‘engine’. Can run as command-line application.
2) CrppServer - The ‘service’ shell around CrpxProcessor. Requires the latter.
3) CrpStudio - CorpusStudio web application. Uses CrpxProcessor internally too.

The three programs have been developed under NetBeans, but it should be possible to get them running under different IDE’s. The ‘github’ sources only provide the Java and JavaScript sources.

Each of the programs listed above contains a ‘lib’ directory with .jar files it makes use of (not on github):

Package	Libraries needed

CrpxProcessor	`log4j-1.2.15.jar`
	`saxon9-s9api.jar`
	`saxon9.jar`
	`saxon9-dom.jar`

CrppServer	`log4j-1.2.15.jar`
	`CrpxProcessor/dist/CrpxProcessor.jar`
	`CrpxProcessor/lib/saxon9-s9api.jar`
	`CrpxProcessor/lib/saxon9.jar`
	`CrpxProcessor/lib/saxon9-dom.jar`

CrpStudio	`log4j-1.2.15.jar`
	`velocity-tools-view-2.0.jar`
	`velocity-1.7.jar`
	`commons-collections-3.2.1.jar`
	`commons-lang-2.4.jar`
	`vtd-xml.jar`
	`CrpxProcessor/dist/CrpxProcessor.jar`

5.1.7 *Running*

The CorpusStudio web application can be started up by (re-)starting tomcat. A possible startup script could be “crpstudio.sh”:

```
sudo service tomcat restart
sudo cp ~/*.empty /usr/share/tomcat/logs/catalina.out
sudo tail -n 400 /usr/share/tomcat/logs/catalina.out
```

This assumes the presence of an empty file called “catalina.empty” in the user’s home directory, it assumes one is running CentOS (hence the “sudo”) and one’s account has sudo-rights established.

The application should be running once the Tomcat log file logs a message like “`INFO: Server startup in 9309 ms`”. See `/usr/share/tomcat/logs/catalina.out`.

5.2 Logs and cleanup

There are a number of log files in use within the Linux system. What is relevant for the `crpp` and `crpstudio` services is but a subset of these.

The file `/usr/share/tomcat/logs/catalina.out` logs debug and error messages issued by the Java web service that runs under tomcat. **This file can get quite large**, so it needs to be removed periodically. Note: the log file is 'automatically' overwritten with an empty log-file when the startup script `/home/erwink/crpstudio.sh` is executed.

6 Appendices

6.1 Trace code: change in query

What is the chain of events that occur (or need to occur) for a change in a part of a query to percolate through completely?

Event handling
Changes start with key strokes or mouse selections being detected.

Enabling of events
But detection needs to be enabled by the event-handler being coupled to the events.
The detection-enabling is done by `crpstudio.project.addChangeEvents()`.
This function is called from `crpstudio.project.switchTab()`. Whenever a user switches to a different tab-page, a call to `addChangeEvents()` is made, enabling the detection of events.

Event-handler
There is one central event-handler for changes in any part of a CRP: `crpstudio.project.ctlTimer()`. This function gets two arguments: the (html div) element from which it originates and a string indicating whether it is an event from a "input", "textarea" or "select" element. This latter argument is not actually used right now ☺. The type of element is, where necessary, determined by using the jQuery `$(el).is("element-name")` function.

The code inside `ctlTimer()` first of all determines what the *value* of the (form) element is, and then determines the environment 'type' (query, definition, constructor, dbfeat, project) from the "id" attribute of the element (through the `crpstudio.project` private method `getItemObject(sCallerId)`).
Note: This is done by comparing the `sCallerId` value with all the "`loc`" values that are stored in the variable `crpstudio.config.prj_access`. So any form-field whose changes need to perculate further *must* have their "`id`" field specifed as a "`loc`" value in `prj_access`.

7 References

INL. 2014. *Blacklab Server Overview*. Accessed September 2014.
URL: https://github.com/INL/BlackLab-server/wiki/BlackLab-Server-overview

Komen, Erwin R. 2009. CorpusStudio manual. Ms. Nijmegen, Netherlands,
<http://erwinkomen.ruhosting.nl/software/CorpusStudio/CrpStu_Manual.pdf >.