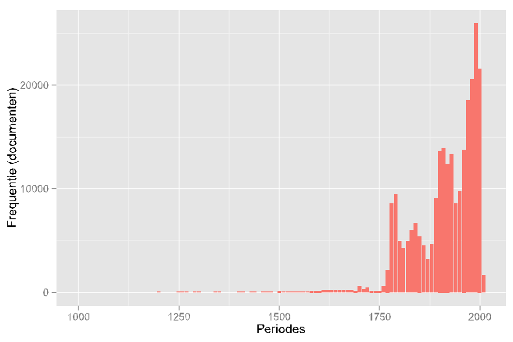
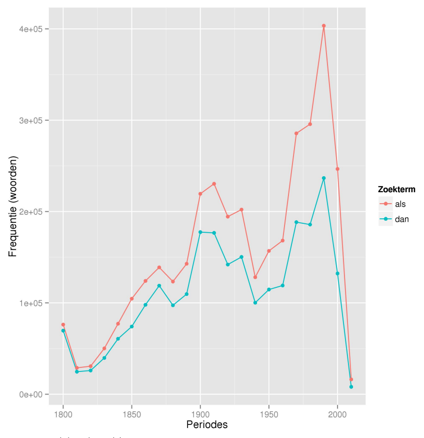
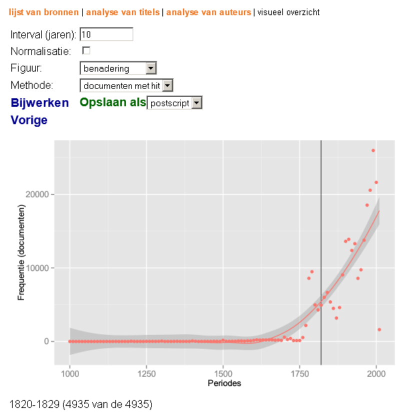
An insider’s guide to the Nederlab R visualization webserver

Erwin R. Komen






lijst van bronnen | analyse van titels | analyse van auteurs | visueel overzicht

Interval (jaren): 10

Normalisatie: ☐

Figuur: benadering

Methode: documenten met hit

Bijwerken Opslaan als postscript

Vorige

1820-1829 (4935 van de 4935)

Start: *January 20, 2015*

This version: 2.6

Date: *September 3, 2015 (10:05)*

CONTENTS

1 Introduction

The “R-webservice” is a separate unit within the Nederlab project, and it can be used separately for other purposes. Its main purpose is to (a) accept search requests, and (b) to provide a visual and a tabular outcome of the search results. The R-webservice executes its search on the Nederlab documents through the “broker” interface, which is a separate web service. The R-webservice runs on a (physical or virtual/cloud) Linux CentOS computer, and has been programmed in Java (running under Tomcat/Apache) and “R” (accessed from Java either through JRI or by opening a new ‘connection’ to a running “Rserve” thread).

The Java program that contains the web-service interface to the “R” functions has been derived from the open-source “BlackLab Server” (BLS) program available on github and developed by INL (2014). This is why some of the input and output JSON is similar to the BLS interface. There are, however, much differences of which a former user of BLS should be aware.

1.1 Short overview: principle components

The main components of the R visualization webservice and its relations with the "onderzoeksportaal" (the web user interface) as well as the "broker" (the web service that handles text search requests) are depicted in Figure 1.

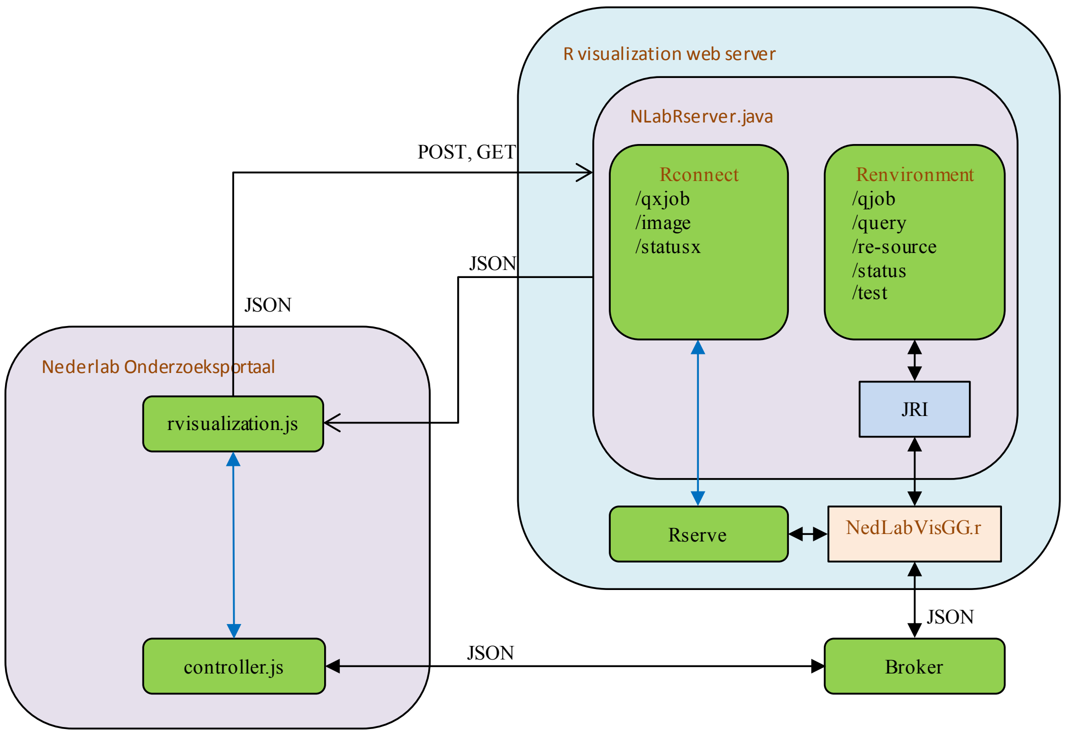


Figure 1 The R visualization webservice in its relation to the "onderzoeksportaal"

The "onderzoeksportaal" contains a number of JavaScript modules, two of which are shown in Figure 1: (a) the "`controller.js`", which communicates with the broker to handle search requests, and (b) the "`rvisualization.js`", which communicates with the "R visualization webservice" running under Apache/Tomcat on a Linux computer. The "controller.js" also is the JavaScript module that communicates with the "`rvisualization.js`" one. Section 7 describes the latter JavaScript component.

The "R visualization webservice" communicates with the outside world through the "`NLabRserver.java`" function (which is part of the NlabR project contained in the `/home/nederlab/webapps` directory of the "`R`" web server computer). The Java part of the web service handles the incoming commands in three different ways:

1) The `/debug` and `/task` commands are handled internally.
2) The `/qxjob`, `/image` and `/statusx` commands are part of the "`Rconnect.java`" interface, which communicates with the "`R`" code in "`NedLabVisGG.r`"through the "`rserve`" program, which runs independent of the Java code on the Linux server.
3) The `/qjob`, `/query`, `/re-source`, `/test` and `/status` commands are part of the "`Renvironment.java`" interface, which communicates with the "`R`" code in "`NedLabVisGG.r`" through the "`JRI`" package of Java (so this "`R`" runs *inside* the Java code).

The web service 'backslash' commands are discussed in chapter 2, while chapter 3 discusses their responses. The "`R`" code in "`NedLabVisGG.r`" is discussed in chapter 8, and the Java web service interface is discussed in chapter 4.

1.2 Short overview: interacting with the R webservice

The most important steps for the web service part of the R-visualization are these:

1) Set up the **Linux** computer that contains the (Java) web service (6.1)
2) Issue a `/qxjob` command for a visualization **request** (2.1) and extract the `jobid` from the initial response (3.2)
3) Monitor the **progress** by issuing a `/statusx` command every now and then (2.2).
4) Once the response (3.3) has the status code "`completed`", use the **svg picture** (as well as the `table` and the `qlist` if needed).
   a) The svg picture can be put in a `<div>` on a html page.
5) **Saving** a figure in a particular format:
   a) See the `/image` command (2.4) and response (3.5).
6) Going back to a **previous** visualization:
   a) Issue a `/task` command with "`pop`" (see 2.3 and 3.4).
7) **Checking** if it works
   a) Check if the Java part of the interface is up and running: `/debug` (2.5)
   b) Check if the "R" part of the interface is up and running: `/test` (2.7)

1.3 Short overview: the JavaScript interface to the R webservice

Important points relating to the "rvisualization.js" module that provides the interface between the Nederlab 'onderzoeksportaal' and the "R" visualization web service are these:

1) **Getting a figure**: initiate using `rvisualization.init` (7.1), and either (a) pass on a `<div>` for the location, or (b) pass on a function that will be called once the figure is available. You need the latter if you want to put it in a custom place and/or take actions that need to take place *after* the figure has been made, such as placing the figure somewhere by yourself (7.2), get/process the data table (7.3) or the list of queries (7.4).
2) Allow the user to **save** a figure in a different **format** (jpeg, epf etc): initiate making the figure in the required format using `rvisualization.ext_initSaving` (7.5). Pass on a function that takes care of the URL received where the image is available for download.
3) Return to the **previous query**: use `rvisualization.ext_prevquery` (7.7). The "formquery" that was used for the previous query is accessible too.

2 Web service input specification

The R-webservice accepts POST, PUT and GET. The POST and GET methods can be used to send data (a query) to the web service. There are, in fact, a number of methods accepted by the R-webservice:

command	**parameter**	**Return**
`/query`	JSON: query	SVG picture + table
`/qjob`	JSON: query	initially: userid + jobid + taskid finally: JSON object with figure, table etc.
`/qxjob`	JSON: query	initially: userid + jobid + taskid finally: JSON object with figure, table etc.
`/status`	JSON: userid, jobid	status object for "qjob"
`/statusx`	JSON: userid, jobid	status object for "qxjob"

`/task`	JSON: cmd, task	cmd= "pop" – remove last task + result of prec. task
		cmd= "get" – result of indic. task
		cmd= "job" – jobid of task

`/image`	JSON: jobid/query, format, figtype	
`/debug`	- *(none)*	
`/re-source`	- *(none)*	Read the "R" functions afresh for "query" and "qjob" processing
`/test`	one word	

Important notes:
1) All commands can be issued to the R-webservice through: `http://server-address/tomcat/nlabr/command?args`
2) The current (feb 2015) server address is `145.100.57.84`. This is a virtual CentOS Linux computer in the cloud.
3) Calling "**query**" may result in a time-out.
   a) This uses the JRI interface, and waits until the task is completely executed
   b) Alternative: use "qxjob"
4) Calling "**qjob**" may cause others calling "qjob" to wait.
   a) This uses the JRI interface, which means that only one person can execute "R" code at any given time.
   b) Alternative: use "qxjob"
5) Calling "**re-source**" only works for "query" and "qjob".
   a) The same functionality for "qxjob" is achieved by going to the R-web server, closing all current "Rserve" processes and executing "sh startRserve.sh".
6) Make sure you call "**status**" to check on "qjob" and "**statusx**" to check on "qxjob"
7) The "**task**" command deals with three different task-number related issues. The "task number" is the number of the r-visualization (q and/or qx)-job that has been issued by one particular user/window (each user/window/tab gets a unique identifier under which the server stores tasks). The "jobid" is a unique number under which *all* the (q and/or qx) jobs requested by *all* users have been stored on the R-visualisation web server. The "task" commands provide an interface between the two.
   a) Command "pop": remove the last task from the current user's stack and then return the "response" object belonging to the new last task (so the penultimate original task).

b) Command "get": return the "response" object that belongs to the indicated task number of the current user.
c) Command "job": get the "jobid" number associated with the indicated task number of the current user.

2.1 Issuing "query", "qjob" or "qxjob"

The query commands `/query`, `/qjob` and `/qxjob` take one json datastructure as argument. The json response datastructure is treated in 3.1 for `/query` and in 3.2 for `/qjob` and `/qxjob`. The difference between the three commands is shown in the following table:

Command	Function
/query	Issue a query and wait until receiving back the SVG picture. (Note: this may lead to a time-out, when much data needs to be gathered.)
/qjob	Start a query job through the JRI interface and report back immediately and supply status updates through `/status`. (Note: only one user can, at any time, be served through the JRI interface.)
/qxjob	Start a query job through a free 'connection' with the 'Rserve' service running locally on the R-visualization web server. Report back immediately and supply status updates through `/statusx`.

Since the R-function that is used for `/query`, `/qjob` and `/qxjob` is identical, the Json components accepted by all three commands are identical too:

`srchTerms`	(Array of strings) List of search items
`cnds*`	(Array of "condition" objects) One "condition" object for each search item
`flts*`	(Array of "filter" objects) One "filter" object for each search item
`colors*`	(Array of strings) One color name for each search item
`labels*`	(Array of strings) One label name for each search item
`userid`	(String) Unique id of user/session/window, grouping a number of search jobs into one 'history'
`iEnv`	(Number) The number of the visualization 'environment' on the user's page
`legend`	show a legend next to the figure (true) or not (false)
`width`	width of the canvas (in inches)
`height`	height of the canvas (in inches)
`yrFrom`	(Numeric) First year to include. If negative (e.g. "-1"), then the first year is taken to be the first year of the decade of first occurrence of the search term.
`yrTo`	(Numeric) Last year
`interval`	(Numeric) Number of years per interval
`norm`	(Boolean) Normalize the frequencies against the number of words (for termfreq) or documents (for docfreq) available in a year/period? `'true'` give normalized frequencies (use intNorm for the factor) `'false'` use absolute frequencies (but the picture may be distorted) (default)
`intNorm`	(Numeric) Normalization factor (this results in the relative frequency being 'n' occurrances per 'intNorm' words)
`file*`	(String) File name to save image to (extension determines type). This particular component will not be of much use for queries to the R-visualisation web server, since the resulting file will be stored locally on the web server, where the caller will not be able to retrieve it. Use the `/image` command instead (see section 2.4).

`server`	(String) Which server to contact for the search: `'nederlab'` The real "Nederlab" project server (default) `'nederlab2'` The “Nederlab” server using broker2 `'radboud'` A mock server providing randomized data (But it won’t be able to handle all broker requests.)
`debugL`	(Numeric) Level of (internal) debugging (0-3) (default=0)

The simplest query may be to look for one word and use as many default values as possible. An example for such kind of JSON argument is in (1).

```
(1) …/qxjob?{ "srchTerms": ["hans"], "yrFrom": -1 }
```

The request in (1) looks for the search term “hans” from the decade it first occurs (hence the “-1” value for the “`yrFrom`” parameter) until the default ending year. All other parameters have their default values: no color or label specifications, use normalisation per 1000 words, look for the term-frequency, and use the default server.

A more complex search would be as in (2):

```
(2) …/qxjob?{ "srchTerms": [ "oorlog", "vreede", "staaking"],
             "colors":     [ "red", "gold", "darkblue"],
             "labels":     [ "oorlog", "vrede", "staking"],
             "source":     "content",
             "method":     "termfreq",
             "vis":        "pointline",
             "yrFrom":     1800,
             "yrTo":       1895,
             "interval":   5,
             "norm":       true,
             "legend":     true,
             "intNorm":    10000,
             "server":     "nederlab",
             "debugL":     0      }
```

The request in (2) looks for the history of three terms (‘oorlog’, ‘vreede’ and ‘staaking’), manually making sure that they are depicted in the three indicated colors, and that the legend contains the modern Dutch words ‘oorlog’, ‘vrede’ and ‘staking’. The starting and finishing years have been specified, as has the interval (5 years). There is normalisation but per 10.000 words.

Calls to the R-visualization web service that are issued from the Nederlab ‘onderzoeksportaal’ will usually have just one search item, but this item may be a complex combination of meta data requirements. This is why such requests generally come with their own ‘broker’ condition, as in (3):

```
(3) …/qxjob? {"vis":"bar","yrFrom":-1, "yrTo":2010, "interval":10,
    "norm":false, "method":"docfreq",
    "events":{
         "fig_mouseover": "nederlab.rvisualization.nlabRshowTip",
         "fig_mouseout":  "nederlab.rvisualization.nlabRhideTip",
         "fig_click":     "nederlab.rvisualization.nlabRclick",
         "vert_mousemove":"nederlab.rvisualization.nlabRshowVert",
         "vert_mouseout": "nederlab.rvisualization.nlabRhideVert"},
    "srchTerms":["Zoek naar: braden;;
                  Zoek in: tekst: ;
                  Verfijningen:  geslacht=man;"],
    "cndlist":[{"type":"and",
                  "list":[
                           {"type":"equals","field":"content","value":"braden"},
                           {"type":"range","field":"nederl_time_order"}]}],
    "userid":"session_2015_1_2_16:19:1.299_62",
    "formq":{     "searchtype":"simple","words-and":"braden",
                  "resultsort":"relevantie","resultorder":"desc",
                  "choices":"tekst","geslacht":["man"],
                  "words-phrase":"","words-or":"","words-not":""}}
```

The query in (3) contains a specification of the broker query to be used in the “cndlist” field. Since the “cndlist” field is not *null*, the specification of the search term in “srchTerms” only serves to distinguish between queries (and re-use the results, where appropriate).

The “userid” field assigns a unique id to each instance of a browser (or a tab within a browser). The value is used by the R-visualization service to keep track of a user’s history, and allow returning to a previously issued query.

Another feature to notice in (3) is the “events” specification. The “events” list contains the correct JavaScript callback function names that are to be called when the listed events occur.

The json query specification also contains a field “formq”. This field is not known to the R-visualization service, but the service won’t complain over this (or other) unknown fields. The JavaScript rvisualization function in the ‘onderzoeksportaal’ has added this field to the query, so that it can have all necessary specifications together in one place.

2.2 Issuing “status” or “statusx”

The /status and /statusx commands are meant to check on the progress of the /qjob and /qxjob commands respectively. Both commands take the same JSON argument:

```
    /statusx?{    "userid":"session_2015_1_2_16:19:1.299_62",
                  "jobid": "25"}
```

The json fields are “userid” and “jobid”. The “userid” should preferably be the same one as has been used to issue the /qjob or /qxjob commands. 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 /qjob or /qxjob commands. This first reply contains the status “started”, and it is only this reply that returns the correct jobid value. See 3.3 for the structure of the response objects to /status and /statusx.

2.3 Issuing “task”

The /task command actually contains four sub-commands, depending on the make-up of the json argument it contains. The /task command takes one JSON argument, properly speaking, but this JSON argument is a list that may consist of up to three fields: “cmd”, “userid” and “taskid”. Here are some examples:

Command	Function

`/task?{"cmd": "show"}`	Show the stack of available jobs. The stack is shown *internally*, inside the catalina.out log file in the R-web server machine.
`/task?{"cmd": "pop", "userid": "session_2015_1_2_62"}`	Remove the last task from the task-stack, and return the previous json `/qxjob` (or `/qjob`) query. This query can then be re-issued, and may lead to a fast response, provided the result is still in the server's cache.
`/task?{"cmd": "get", "userid": "session_2015_1_2_62", "taskid": "4"}`	Leave the task-stack as it is, but retrieve the query belonging to task number "`taskid`". This query can then be re-issued, and may lead to a fast response, provided the result is still in the server's cache.
`/task?{"cmd": "job", "userid": "session_2015_1_2_62", "taskid": "4"}`	Return the "`jobid`" number belonging to the indicated `taskid`.

See 3.4 for the possible responses.

2.4 Issuing "image"

The `/image` command is the preferred way of having the R visualization web server prepare an image file (such as a .jpeg one). The command contains a number of obligatory and optional arguments:

Argument	Type	Obl/Opt	Description
query	string	`query` or `jobid`	The "`query`" value must be the complete "`rquery`" string (that is: stringified JSON object) as it has originally been issued through a `/query`, `/qjob` or `/qxjob` command.
jobid	number	`query` or `jobid`	Whenever the "`jobid`" parameter is specified, the "`query`" parameter (if given) is ignored. The figure will be derived from the information stored on the R-visualization web server for the job with the given `jobid`. The "`jobid`" number itself can be derived from the "taskid" through the `/task{"cmd": "job"}` command.
userid	string	*optional*	The userid-string as returned by the original `/qjob` or `/qxjob` command's "`started`" status.
format	string	obligatory	The extension of the image that has to be produced. This extension does double duty as a label for the format: `jpeg` JPEG raster image (compressed) `png` portable network graphics (compressed raster) `bmp` BMP raster image `tex` TeX/LaTeX formatted line drawing `eps` (Extended) PostScript format `pdf` PDF document `svg` scalable vectore graphics picture
figtype	string	*optional*	The kind of figure that has been produced ('`bar`', '`barstack`', '`line`' and so on – see the "vis" option in section 2.1). When specified, this string will be included in the name of the figure.

Example:

`/image?{"jobid": 10, "format": "jpeg"}` - Make an image from the search job with id "10" and save it on the server in the "jpeg" format. Return a link to this image. The syntax of the returned (JSON) string is described in section 3.5.

2.5 Issuing "debug"

The `/debug` command is to be issued without any argument. Its purpose is to see if the R-visualization service is up at all. See 3.6 for the response.

2.6 Issuing "re-source"

The `/re-source` command forces the R-code in the server to be re-loaded. See 3.7 for the response. This command only has effect on the functioning of `/query` and `/qjob`: these execute "R" through the JRI interface (so "R" is, in a sense, part of the Java package). The `/qxjob` command calls "R" through a locally running "Rserve" connection, which means that the `/re-source` command does not affect it. The equivalent to the `/re-source` command for `/qxjob` is: (a) enter the Linux server that runs the R webservice, (b) kill all the "Rserve" processes, (c) run the script `startRserve.sh` (in the nederlab home directory).

2.7 Issuing "test"

The `/test` command may be given without argument, or with just one (non-JSON) argument. A simple word will do, such as: `/test?kees`. See 3.8 for the responses.

Note: the `/test` command right now only tests the correct functioning of the JRI interface from Java to "R". The commands using this interface are: `/qjob`, `/query`, `/re-source`, `/status`. Other commands (that is: `/qxjob` and `/statusx`) make use of the independently running "`rserve`" service. The correct working of "`rserve`" is not (yet) obtained through the `/test` command.

3 Web service output specification

Most of the commants specified in section 1.2 result in a JSON response from the R-webservice. The structure of these responses is not always the same in version 1.4. This section provides an overview of all the possible response structures.

Note: efforts are underway to reach a unified response structure that will consist of a JSON list containing the following elements:

Argument	Type	Description
`indexName`	string	This is a copy of the reqesting index name (e.g "`query`", "`qxjob`", "`statusx`" and so on)
`status`	object	The status object is a key-value pair list that has at least the elements "`code`" and "`message`". The "`code`" will, in general, be one of four: "`started`", "`working`", "`completed`" or "`error`". The "`message`" part is task-specific and attempts to pass on the status in prosa, and, where possible, in more detail.
`content`	object	A key-value pair list containing command-specific content.

3.1 Response to "query"

The response to a `/query` command is a JSON list of 8 key-value pairs:

Argument	Type	Description
`indexName`	string	Should have the value "`query`"
`resource`	string	Empty; not used
`status`	string	Contains "Completed" upon successful completion Contains "Error" when an error was found
`rest`	string	Empty; not used
`queryString`	string	A stringified copy of the original `rquery`
`query`	String	A message accompanying the reply by the server
`figure`	String	A string containing the SVG picture (if successful)
`table`	string	A stringified JSON object containing the table with the found numbers.

An example of a successfully returned JSON object is this one, where the "figure" and the "table" string have been replaced by "…" in order to save space:

```
{ "indexName": "query",
  "resource": "",
  "status": "Completed",
  "rest": "",
  "queryString": "{\"srchTerms\": [\"aap\"], \"yrFrom\": 1800, \"yrTo\": 1860}",
  "query": "R-query has been executed",
  "figure": "…"
  "table": "…" }
```

When the server detects an error in the /query argument, the following type of answer is returned:

```
{ "indexName": "query",
  "resource": "",
  "status": "Error",
  "rest": "",
  "queryString": "{\"srchTerms\": [\"aap\"], \"yrFrom\": 1800, \"yrTo\": \"aap\"}",
  "query": "R-query returned an error",
  "message": "Error in intFrom:intTo : NA/NaN argument\n",
  "figure": "",
  "table": "" }
```

3.2 Response to “qjob” and “qxjob”

The initial response to a `/qjob` or `/qxjob` command is a JSON coded object that consists of a “`status`” part and a “`content`” part. The most important return parameter is the `content.jobid`, which is subsequently needed to retrieve the results of the correct query job:

```
{ "status": {
    "code":     "started",
    "message": "Searching, please wait...",
    "userid":  "F029586E3E34FF2FC40DC63820EA5EDB",
    "checkAgainMs": 200  },
  "content": {
    "jobid": "67"}
}
```

The status object consists of the following parts:

Key	Type	Description
`code`	String	Either “`started`”, “`completed`” or “`error`”
`message`	String	An explanation or sub-status to the “`code`” field
`userid`	String	The unique user/session id that has been supplied by the calling party
`checkAgainMs`	number	The number of milliseconds that should pass minimally before checking for the status again (this is *not* used right now)

The contents object only has one part:

Key	Type	Description
`jobid`	String	The number that is used internally in the web server for this particular visualization job

The progress of the `/qjob` or `/qxjob` can be tracked by issuing `/status` or `/statusx` command (see section 2.2). Once the `/status` or `/statusx` command returns a status code “completed”, the resulting data can be obtained by re-issuing the original `/qjob` or `/qxjob`. This will return a JSON structure consisting of 11 list items:

Key	Type	Description
`indexName`	String	This containst the command: "`qjob`" or "`qxjob`"
`content`	Object	A JSON key-value pair list containing details of the search that has been conducted: `searchParam` The original search (see also queryString) `searchTime` The time taken for the search in milliseconds `searchDone` Since the search is completed, the value is **true** `searchStatus` This has the value "completed", signalling success
`resource`	String	Empty; not used
`status`	String	Contains "Completed" upon successful completion Contains "Error" when an error was found
`rest`	String	Empty; not used
`queryString`	String	A stringified copy of the original `rquery`
`taskid`	Number	The task number relative to the user/session/browser-tab-page
`query`	String	A message accompanying the reply by the server
`figure`	String	A string containing the SVG picture (if successful)
`table`	String	A stringified JSON object containing the table with the found numbers.
`qlist`	String	A stringified JSON object containing all the broker queries that have been used by "R"

An example of a successful response from a `/qxjob` looks like this (compare this with the return to the `/query` command in section 3.1):

```
{ "indexName": "qxjob",
  "status": {
    "code": "completed",
    "message": "R has finished",
    "checkAgainMs": 200
  },
  "content": {
    "searchParam": {
      "query": "{ \"srchTerms\": [ \"oorlog\", \"vreede\",
    \"staaking\"],\"colors\":     [ \"red\", \"gold\", \"darkblue\"],\"labels\":
    [ \"oorlog\", \"vrede\", \"staking\"],\"source\":     \"content\",\"method\":
    \"termfreq\",\"vis\":         \"pointline\",\"yrFrom\":     1800,\"yrTo\":
    1895,\"interval\":    5,\"norm\":        true,\"legend\":     true,\"intNorm\":
    10000,\"server\":     \"nederlab\",\"debugL\":     0}"
    },
    "searchTime": 19169,
    "searchDone": true,
    "searchStatus": "completed"
    "userid": "AD7EF1711ECB64C0264BAB8E99F12506",
    "jobid": "67",
    "taskid": 1,
    "query": "R-query has been executed",
    "figure": "…",
    "table": "…",
    "qlist": "…"  }
  } }
```

3.3 Response to "status" and "statusx"

The response to the `/status` and `/statusx` commands contain a json structure with a general status code and a more specific status message. A valid in-between response would be:

```
    {    "indexName": "statusx",
         "status":{"code":"working",
                 "message":"searching:36:223",
                "userid":"session_2015_1_2_16:19:1.299_62",
                "jobid":"1",
                "checkAgainMs":200}}
```

Should there be an error, then the code is:

```
    {    "indexName": "statusx",
         "status":{"code":         "error",
                  "message":       "this contains an error message",
                  "userid":        "session_2015_1_2_16:19:1.299_62",
                  "jobid":         "0",
                  "checkAgainMs":  200}}
```

The way to keep track of the status, then, is to monitor the value of `status.code`. The following values can be expected:

status.code	explanation
working	The job is bein processed. A more informative message indicating the progress of the job can be found in `status.message`.
completed	The job has successfully completed. (The `status.message` value should also be "completed".) The *result* of the job can be collected by re-sending the original `/qjob` or `/qxjob` query, and harvesting the response to it.
error	An error has occurred. A more specific message may be found in `status.message`.

Please note: a `/statusx` or `/status` call never returns the result of a `/jobx` or `/job`.

The resulting (svg) figure can only be collected by re-issuing the original `/jobx` or `/job`.

3.4 Response to "task"

The `/task` command consists of a number of sub-commands, and the responses vary accordingly. The overal response is the list of the `indexName`, a `content` object and a `status` object (should there be an error, then the status object contains the `status.code` "`error`" as well as an explanation in the `status.message` part). The "`get`" and "`pop`" sub-commands are aimed at returning a previously issued `/qjob` or `/qxjob` query. A possible response is this:

```
{   "indexName":"task",
    "content":{
         "query":         "{\"vis\":\"bar\",\"yrFrom\":-
    1,\"yrTo\":2010,\"interval\":10,\"norm\":false,\"method\":\"termfreq\",\"events\
    ":{\"fig_mouseover\":\"nederlab.rvisualization.nlabRshowTip\",\"fig_mouseout\":\
    "nederlab.rvisualization.nlabRhideTip\",\"fig_click\":\"nederlab.rvisualization.
    nlabRclick\",\"vert_mousemove\":\"nederlab.rvisualization.nlabRshowVert\",\"vert
    _mouseout\":\"nederlab.rvisualization.nlabRhideVert\"},\"srchTerms\":[\"er\"],\"
    cndlist\":null,\"userid\":\"session_2015_1_2_16:19:1.299_62\",\"formq\":{\"searc
    htype\":\"simple\",\"words-
    and\":\"er\",\"resultsort\":\"relevantie\",\"resultorder\":\"desc\",\"words-
    phrase\":\"\",\"words-or\":\"\",\"words-not\":\"\"}}",
         "jobid":1}
    "status":{"code":     "completed",
             "message":   "task [pop] has been successfully executed",
             "userid":    "session_2015_1_2_16:19:1.299_62"   }}
```

The "`content`" object of the response differs, depending on the sub-command of `/task`. The "`get`" and "`pop`" sub-commands return the requested `/qjob` or `/qxjob` query (as a string) in "`content.query`". The "`get`", "`job`" and "`pop`" sub-commands return the relevant `jobid` in "`content.jobid`". The "`pop`" sub-command returns a stringified JSON array of objects containing a `jobid, userid` and `query` in "`content.stack`".

3.5 Response to "image"

When the /image command is issued with the correct parameters and results in the creation of an image, a JSON object is returned containing the "indexName", "content" and "status" parts. The "content.location" gives the location where the prepared image can be downloaded (the server's address needs to be prepended: http://server/img/nlabout_bar.21.jpeg):

```
{   "indexName":"image",
    "content":{  "location":      "img/nlabout_bar.21.jpeg"}
    "status":{"code":     "completed",
              "message":  "The image is ready at the indicated location",
              "userid":   "session_2015_1_2_16:19:1.299_62"  }}
```

The "location" contains the part of the URL relative to the server's main URL. The status contains "completed" upon success, and "error" when something is wrong. When, for instance, a request is made for a non-existent job number, the following JSON string returns:

```
{   "indexName":"image",
    "status":{"code":     "error",
              "message":  "Cannot find image file of job: 1",
              "userid":   "session_2015_1_2_16:19:1.299_62"  }}
```

The /image command checks whether the search job for which an image is being requested has actually finished. If this is not the case, it also returns an error. This error can be avoided by first checking for the status of a job (through /statusx) and making sure the /image command is only issued when the job has finished.

3.6 Response to "debug"

There is only one successful response to the /debug command and that is the following JSON structure:

```
  {   "indexName":         "debug",
      "status":{"code":   "completed",
                "message":"The Java-part of the R-webservice works fine.",
                "userid": "session_2015_1_2_16:19:1.299_62"  }}
```

The "indexName" copies the /debug command and the status object contains a "completed" status with an accompanying message, as well as a copy of the userid.

3.7 Response to "re-source"

The response to a /re-source command is a json structure consisting of the "indexName" and a "status" object. The status code can be either "completed" or "error". In the latter case an error message is contained in status.message.

```
  {   "indexName":         "re-source",
      "status":{"code":   "completed",
                "message":"R-functions have been re-sourced",
                "userid": "session_2015_1_2_16:19:1.299_62"  }}
```

3.8 Response to "test"

The command /test can be used with and without arguments. The JSON response copies the indexName and contains a "status" and "content" object. If "R" returns an error, the status.code equals "error" and the status.message contains additional information. The response to a call without arguments is this:

```
{   "indexName":"test",
    "content":{"testRes": "R-function test() without arguments"}
    "status":{"code":     "completed",
              "message":  "R has executed test()",
              "userid":   "session_2015_1_2_16:19:1.299_62"  }}
```

When there is an argument, for instance “kees”, the response will be like this:

```
{   "indexName":"test",
    "content":{"testRes":"Arg=[kees], lib=[
         [1]: /home/nederlab/R/x86_64-redhat-linux-gnu-library/3.1
         [2]: /usr/lib64/R/library
         [3]: /usr/share/R/library],
         wd=[/usr/share/tomcat],
         R_HOME=[/usr/lib64/R],
         R_LIBS=[~/R/x86_64-redhat-linux-gnu-library/3.1],
         R_LIBS_USER=[/home/nederlab/R/x86_64-redhat-linux-gnu-library/3.1]
         R_ENVIRON=[/usr/lib64/R/etc/Renviron]"}
    "status":{"code":      "completed",
              "message":   "R has executed test()",
              "userid":    "session_2015_1_2_16:19:1.299_62"  }}
```

The test function not only copies the argument, it also provides the values for a number of server-internal locations within the `content.testRes` field:

1) All the locations where “R” looks for libraries
2) The working directory of “R” (the result of applying `getwd()`)
3) A few environment variables: `R_HOME`, `R_LIBS`, `R_LIBS_USER` and `R_ENVIRON`

Note: the `/test` command right now only tests the correct functioning of the JRI interface from Java to "R". The commands using this interface are: `/qjob`, `/query`, `/re-source`, `/status`. Other commands (that is: `/qxjob` and `/statusx`) make use of the independently running "`rserve`" service. The correct working of "`rserve`" is not (yet) obtained through the `/test` command.

4 Internal make-up of the web service

The R visualization web service is controlled by a Java program that runs under “tomcat” (which runs under an Apache httpd service) on a Linux computer (see section 6.1 for installation instructions). The compiled Java code (developed under “NetBeans”) is a “`.war`” file, and as soon as this `.war` file is placed on the `~/webapps` directory of the Linux server, the Tomcat server unpacks and installs it.

4.1 The web service’s body: Java

The body of the Java web service program has been copied from INL’s BlackLab Server, but it has been completely adapted for the R-visualization purpose. The Java service itself is aimed at supplying as little as possible ‘content’ processing; such processing is transferred to the “R” engine(s). What the Java service does handle is all the communication with the requester and some job caching. The main components of the Java code, as well as their interaction with the "R" code and the broker, are illustrated in Figure 2.

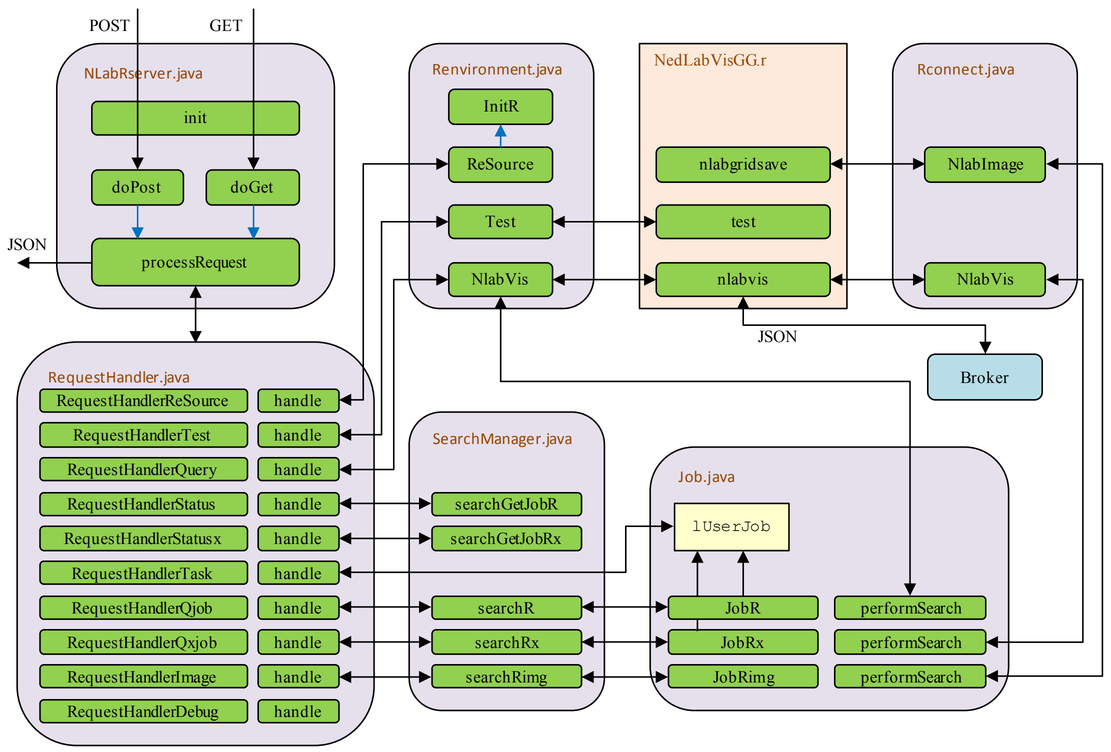


Figure 2 The Java web service program

As soon as the Tomcat server is started up, it executes the "`init`" function in the `NlabRserve.java` class. This `init` function reads the configuration parameters, and then sets up the "`Renvironment.java`" and the "`Rconnect.java`" classes, which, respectively, provide interfaces to "`R`" through `JRI` and `Rserve`.

Once started up, any POST or GET requests enter `NlabRserver.java` through "`doPost`" and "`doGet`", but they are all treated alike in the central function "`processRequests`". This latter function calls upon the "`RequestHandler.java`" class to deal with the individual requests through "`handle`" functions. The requests are handled in a number of ways:

Request	Handling
`/re-source`	Call the JRI interface handling class `Renvironment.java` and execute `InitR`
`/test`	Call the JRI interface handling class `Renvironment.java` and execute the "`test`" function in the `NedLabVisGG.r` code.
`/query`	Call the "`NlabVis`" function in `Renvironment.java` and execute the "`nlabvis`" function in the `NedLabVisGG.r` code. This uses the JRI interface, and since there is no job-threading the function only returns when the job is done. Larger jobs get timed out.
`/status`	Access the `SearchManager.java` class and locate the "`qjob`" with the indicated jobid in the search Job cache. Then pass back the job's "`status`" and "`content`".
`/statusx`	Access the `SearchManager.java` class and locate the "`qxjob`" with the indicated jobid in the search Job cache. Then pass back the job's "`status`" and "`content`".
`/task`	Work with the "`lUserJob`" list contained in "`Job.java`". This list connects users through their unique "`userid`" to jobs (through the `jobid`)
`/qjob`	Call the "`NlabVis`" function in `Renvironment.java` (the JRI interface) and execute the "`nlabvis`" function in the `NedLabVisGG.r` code. The call goes through the `SearchManager.java` class, which turns the search into a job inside a new thread. The job is archived in the "`lUserJob`" list.
`/qxjob`	Call the "`NlabVis`" function in `Rconnect.java` (the Rserve interface) and execute the "`nlabvis`" function in the `NedLabVisGG.r` code. The call goes through the `SearchManager.java` class, which turns the search into a job inside a new thread. The job is archived in the "`lUserJob`" list.
`/image`	Call the "`NlabImage`" function in `Rconnect.java` (the Rserve interface) and execute the "`nlabgridsave`" function in the `NedLabVisGG.r` code. The call goes through the `SearchManager.java` class, which turns the search into a job inside a new thread.
`/debug`	This is handled inside the `RequestHandler.java` class.

4.2 Configuration

There are a number of parameters that can be specified in a configuration file. There are default settings for many of the parameters, and there is a default configuration file (`nlabr-server-defaults.json.txt`) that comes as part of the `NlabR.war` package. Instead of relying on this default configuration file, however, the software deployer should provide a configuration file named "`nlabr-server.json`", which should be placed in the `/home/nederlab/webapps` directory, where it gets picked up by the "init" function of `NlabRserve.java`.

The configuration file is a list of JSON objects:

Object	Description
`debugModeIps`	A list of IPs that run in debug mode
`rinfo`	Contains important file and directory specifications for "R". Make sure "`institute`" selects the "`nederlab`" one.
`indices`	NOT USED *(could be deleted)*
`requests`	Settings here affect how requests are handled (see BLS). Make sure `defaultOutputType` is set to `json`. The other settings are not used (I think).
`performance`	Settings here affect how jobs are handled (see BLS). Changes here affect `/qjob`, `/qxjob` and `/image` commands. Test and check to see if the number "`20`" is not too low for `maxNumberOfJobs`.

4.3 The JRI interface to “R”

The JRI interface to "R" is contained in the `Renvironment.java` class. JRI directly calls the "C" code functions through which "R" is interpreted. The "`InitR`" function loads the "`R`" code contained in the `NedLabVisGG.r` file. The location of this file is set in the configuration file in `rinfo.nederlab.rCodeLoc`.

The "`init`" function in the `NlabRserver.java` class creates one instance of the `Renvironment` class, and this instance is available through the `getRenv` function.

4.4 The “Rserve” interface to “R”

The Rserve interface to "R" is contained in the `Rconnect.java` class. The "`init`" function in the `NlabRserver.java` class creates one instance of the `Rconnect` class, and this instance is available through the `getRcon` function.

The "`rserve`" program is started up independently (see section 6.2), and when it does, it loads the `NedLabVisGG.r` file.

The `Rconnect.java` class contains a cache of "connections" to `Rserve`. Whenever a request reaches `NlabImage` or `NlabVis` inside the `Rconnect.java` class, a free connection is obtained from this cache. Connections are not tied to users, which means that a user may make use of different connections from job to job. This design choice, together with the "caching" feature of results in "R" described in section 5.3, means that requests for slightly changed visualizations that could be handled without making additional calls to the broker, may, nevertheless, still result in repeated broker requests. Timing, then, depends on the caching capabilities of the broker.

4.5 Job cancellation

Every job (be it JobR, JobRx or JobRimg) has a unique "`jobid`", and each job can have multiple 'clients' waiting for it to finish. Since it is undesirable to have more than one job per user-context (that is: a visualization location on the screen of a user), there is a facility to decrease the number of clients for a particular job from a particular user. The procedure involves a number of steps:

1) A call to the function `changeClientsWaiting(-1)` for the job in question.
2) If there are no more ‘clients’ waiting for the fulfillment of this job, then:
   a) The job is ‘cancelled’: the job's thread receives an interrupt and ceases to exist.
   b) The job’s “reusable” flag is set to “false”.
   c) A flag file with this `jobid` and the extension "`.stop`" is created.
   d) The “R” code stops as soon as it detects the presence of the flag file, and if it does this, it deletes the flag file again.

This arrangement has only been activated for the "Qxjob" type – the jobs that result from a `/qxjob` command. The `RequestHandlerQxjob.java` class's "`handle`" function is the place where job inspection and potential cancellation takes place. The code makes use of the static `lUserJob` list that keeps track of all the user-job combinations. It inspects which of the user's jobs of type "`jobrx`" (which is the internal code for `/qxjob`) are still active, and it then takes two steps: (a) it creates a file with this `jobid` and the extension "`.stop`", and (b) it decreases the number of clients for the job by one. The former action serves to signal to the "R" code that the `nlabvis()` function should be stopped.

The “R” code “`NedLabVisGG.r`” contains the function “`CheckStop`”. When this function is called, it checks for the presence of the file created by Java consisting of its `jobid` and the extension "`.stop`". If that file exists, `CheckStop` removes it and causes the “R” execution to stop. (It returns with an “error” message “Interrupted”, but this error message is not captured by the Java code anymore, since it already abandoned taking care of this particular function.)

Using a "`.stop`" file has a potential danger: if the “R” code does not come to a point where “`CheckStop`” identifies the `.stop` file and deletes it, then the `.stop` file remains there. When the “tomcat” service is re-started and a job with the same number starts, the “R” code might think it needs to stop. However, the "R" code clears any old `.log` and `.stop` files *of its own jobid* at the very beginning of `nlabvis()` by calling `InitStatus`.

5 Caching and history

Caching of results in the R visualization web service takes place on a number of places:

1) The JavaScript "rvisualization.js" uses an array to keep the variables belonging to one visualization realization on the screen of a user together.
2) The Java web service contains a cache of the latest 20 jobs carried out by all users connected to the service. These jobs include: `/qjob`, `/qxjob` and `/image` instances.
3) The Java web service contains a list of user-job pairs: which job (that is: `/qjob`, `/qxjob` and `/image`) has been carried out by which user. This list remains active as long as the tomcat web service is running.
4) The Java web service contains an array of Rserve 'connection'. Each job is handled by a connection from this array, and if there is no free one, then a new connection is added.
5) Every "R" session (that is, within one connection in the array of connections) has a small cache of the last 10 visualization requests that have been carried out.

5.1 User/session history in rvisualization.js

There is, properly speaking, no *history* of user sessions within rvisualization.js, nor is there a real *cache*. However, the `/task` command provides access to the list of jobs (that is: `/qjob`, `/qxjob` and `/image`) that have been carried out through the current user's context.

This 'context' is a unique string for the combination of: (a) Nederlab visualization user, (b) browser instance (or tab page within a browser), and (c) visualization location on the screen (since one Nederlab screen may have more than one location where a visualization is being shown).

The variables that are global within "`rvisualization.js`" are kept in the array `arEnv`, where each element in the array can be reached through its unique context string.

5.2 The Java web service: job cache

The Java web service contains a cache of the latest 20 jobs carried out by all users connected to the service. These jobs include: `/qjob`, `/qxjob` and `/image` instances. This job caching mechanism is retained from the BlackLab Server code (see BLS).

5.3 The Java web service: user-job history

The Java web service contains a list of user-job pairs: which job (that is: `/qjob`, `/qxjob` and `/image`) has been carried out by which user. This list remains active as long as the tomcat web service is running. There is no size limit to this array.

The array elements are JSON objects, and each object contains the following elements:

Element	Description
`job`	The kind of job: "`jobr`" or "`jobrx`"
`userid`	The "`userid`" value used for this job
`timestamp`	Date and time when the job got added to this array
`jobid`	The "`jobid`" identifying the job in the job cache
`query`	The rQuery that was used as argument to the `/qjob` or `/qxjob` search request

Jobs are added through "`addUserJob`", the current array is listed through "`showUserJob`", the last element is popped from the array through "`popUserJob`", and a "`jobid`" can be found through "`getJobFromTask`".

5.4 The Java web service: rserve connections

The Java web service contains an array of Rserve 'connection'. Each job is handled by a connection from this array, and if there is no free one, then a new connection is added. There is no size limit to this array. The connections are only reset when "rserve" is killed and started up again.

5.5 Caching of results within “R”

Every "R" session (that is, within one connection in the array of connections) has a small cache of the last 10 visualization requests that have been carried out.

This cache is realized as a global array "`arNlab`" in the `NedLabVisGG.r` module. The size of the array is the global variable `arNlabSize` that can be changed in the "R" code. If the number of visualizations for a particular connection is larger than 10, then the array elements are copied down, and the oldest one drops out. The array is accessed through the functions `nlabpush` and `nlabget`.

6 Web service maintenance

6.1 Setting up a completely new server

Assumptions:

1) A virtual host cloud machine has been prepared based on Linux RedHat/CentOS
2) There is an account “nederlab” for this machine

Installing apache (=httpd):

```
sudo yum install httpd
```

Installing tomcat:

```
sudo yum install tomcat
```

Or more specific:

```
sudo yum install tomcat6 tomcat6-webapps tomcat6-admin-webapps
```

Letting tomcat work under apache:
Find directory `/etc/httpd/conf.d` and add a new file `ajp.conf`:

```
ProxyRequests Off
<Proxy *>
        Order deny,allow
        Deny from none
        Allow from localhost
</Proxy>
ProxyPass           /tomcat/nlabr ajp://localhost:8009/nlabr
ProxyPassReverse    /tomcat/nlabr ajp://localhost:8009/nlabr
```

Adapt the file `/usr/share/tomcat/conf/server.xml`, adding after the `<Host>` section for “`localhost`” another section:

```
      <Host name="localhost2"  appBase="/home/nederlab/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>
```

This section only serves to make sure that any `.war` file that is put in `/home/nederlab/webapps` gets automatically unpacked.

Add a context specification file `nlabr.xml` to the `localhost` in `/usr/share/tomcat/conf/Catalina/localhost`:

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="/home/nederlab/webapps/NLabR" path="/nlabr" reloadable="true" />
```

Installing R:

```
sudo yum install R
```

If this does not work, try:

```
# For El6 or CentOS 6
su -c 'rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-
    8.noarch.rpm'
sudo yum update
sudo yum install R
```

Preparing basics for R libraries:

```
sudo yum install curl curl-devel
sudo yum install libxml2 libxml2-devel
```

Getting R started up correctly:

```
R
setwd("/home/nederlab")
source("NedLabVisGG.r")
package("rJava")
package("Rserve")
```

The file `.bash_profile` needs to be adapted to contain several variables…

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
# User specific environment and startup programs
JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.71.x86_64/jre
JRE_HOME=$JAVA_HOME
R_HOME=/usr/lib64/R
JRI_HOME=/home/nederlab/R/x86_64-redhat-linux-gnu-library/3.1/library/rJava
PATH=$JAVA_HOME/bin:$R_HOME/bin:$JRI_HOME:$PATH:$HOME/bin
CATALINA_HOME=/usr/share/tomcat
# Make the variables available
export JAVA_HOME
export JRE_HOME
export R_HOME
export PATH
R_SHARE_DIR=/usr/share/R
export R_SHARE_DIR
R_INCLUDE_DIR=/usr/include/R
export R_INCLUDE_DIR
R_DOC_DIR=/usr/share/doc/R-3.1.2
export R_DOC_DIR
JRI_LD_PATH=${R_HOME}/lib:${R_HOME}/bin:
if test -z "$LD_LIBRARY_PATH"; then
  LD_LIBRARY_PATH=$JRI_LD_PATH
else
  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JRI_LD_PATH
fi
```

Tomcat needs to start in such a way that it knows where to find the additional libraries. So adapt `/etc/tomcat/tomcat.conf`:

```
# Use JAVA_OPTS to set java.library.path for libtcnative.so
JAVA_OPTS="-Djava.library.path=/home/nederlab/R/x86_64-redhat-linux-gnu-
    library/3.1/library"

java.library.path = /home/nederlab/R/x86_64-redhat-linux-gnu-library/3.1
        /usr/lib64/R/lib
        /usr/lib64/R/bin
        /usr/java/packages/lib/amd64
        /usr/lib64
        /lib64
        /lib
        /usr/lib
```

This conf file also needs a last line with `R_HOME` defined as `/usr/lib64/R/bin`.

Directories to be created:

`/usr/share/tmp`   Must be readable and writable by *everyone*!!!

6.2 Re-starting the service

Starting and stopping of the Apache server:
```
sudo service httpd start
```

Starting and stopping tomcat:
```
sudo service tomcat start
sudo service tomcat restart
sudo service tomcat stop
```

If there is a pid error message, then remove the **tomcat PID lock** files:
(NOTE: this is usually not necessary)
```
sudo rm /var/run/tomcat.pid
sudo rm /var/lock/subsys/tomcat
sudo service tomcat start
```

If the cataline.out file has become too large, then replace the existing catalina.out with the **empty** one in the nederlab home directory:
```
sudo cp /home/nederlab/catalina.empty /usr/share/tomcat/logs/catalina.out
```

Any changes in the R-location and the location or name of the `NedLabVisGG.r` file should be processed in the file: `/home/nederlab/webapps/nlabr-server.json`

Also restart the Rserve program **after having killed any Rserve processes**:
```
ps -ef | grep Rserve
kill xxxx         (fill in the process number to be killed)
cd ~
sh startRserve.sh
```
This re-loads the current `~/NedLabVisGG.r`

Note: the "`Rserve`" program makes use of a configuration file `~/Rserve.conf`:
```
workdir /tmp/rserve
remote disable
port 6311
encoding utf8
interactive no
source /home/nederlab/NedLabVisGG.r
eval library("RCurl")
eval library("rjson")
eval library("ggplot2")
eval library("XML")
eval library("gridSVG")
eval library("Rserve")
eval library("grid")
```
This configuration file contains a number of essential initializations that are meant to spead-up the functioning of the "`R`" webservice. It reads the `NedLabVisGG.r` source, and it pre-loads the libraries that are needed for "`R`" to function correctly. These libraries are loaded with the "`R`" command "`library()`", but this assumes that they have already been successfully installed using the "`R`" command `install.packages("name")`.

6.3 Logs and cleanup

There are a number of log files in use within the Linux system. What is relevant for the visualization service 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.

The "Rserve" program logs matters under its connections. Each connection gets an own directory under the `/tmp/rserve` one. The files do not seem to be that large.

The directory `/usr/share/tmp` is used to keep ".log" and ".grid" files belonging to visualization jobs. These files are used in order to fall back on earlier jobs. The directory could be cleaned when tomcat is restarted. Individual files can probably be removed safely when they are older than 24 hours.

7 The Nederlab "rvisualization.js" interface

The web service as described in chapters 1-6 serves the Nederlab "onderzoeksportaal" through the JavaScript module "rvisualization.js". This chapter gives an overview of how to use this module and describes essential features of its internal make-up.

The rvisualization module contains a number of JavaScript functions that interact with the R-webserver where needed. A short overview of these functions is provided in Table 1.

Table 1 JavaScript functions that interface with the "R" web service

Function	Purpose
`init(rvisArgs)`	Calculate the visualisation specified in "`rvisArgs`".
`ext_getFigure(sContext)`	Return the visualisation available in `sContext` as svg object.
`ext_getTable(sContext)`	Return the values of the visualization in `sContext` as a table.
`ext_getQlist(sContext)`	Return a list of all the broker queries used to calculate the visualization in `sContext`.
`ext_initSaving(format, sContext, fn_endSaving)`	Start creating a picture in the specified `format` of the visualization of `sContext`. The URL string to this picture must be retrieved through the function `fn_endSaving`.
`ext_changevis(sContext, oVisArgs)`	Change the visualization according to the specifications in `oVisArgs`.
`ext_prevquery(sContext)`	Go to the previous visualization of `sContext`.

The following subsections provide more explanations to the rvisualization JavaScript functions and come with examples for each of them.

7.1 Function "nederlab.rvisualization.init"

The `init()` function translates the visualization specification into a format request for the R visualization web service, sends this request to the web service, receives the response, and, depending on the specifications of `rvisArgs`, it will either show the figure immediately, or else make it available for subsequent functions such as `ext_getFigure()`.

The `rvisArgs` object that needs to be passed on to the `init()` function may contain the key-value pairs as listed in Table 2.

Table 2 The possible components of the `rvisArgs` object

Key	Explanation
`formquery`	The form query object (specifying the search request on the user-form)
`brokerquery`	The broker query object (as has been used for the search request)
`context`	String that uniquely describes the place of the picture on the user's window/tab
`rquery`	Internal use: the JSON string passed on to the R-visualization service
`width`	Optional: width of the picture (in pixels)
`height`	Optional: height of the picture (can be left unspecified if equal to `width`)
`context`	visualization key string (e.g. the name of the visualization on a web page)
`cb_function`	Function to be called after a visualization has been created
`cb_arg`	Second argument to `cb_function`. (The first argument is the `context` string.)
`divVis`	Name of the `<div>` inside which this visualization should be positioned
`divStatus`	Name of status `<div>`
`divError`	Name of error `<div>`
`outFig`	Make a `<div>` for the figure and show it there (provided `divVis` is specified)
`outTab`	Make a `<div>` for the table and show it there (provided `divVis` is specified)
`outQry`	Make a `<div>` for the query-list and show it there (provided `divVis` is specified)
`outRef`	Make a `<div>` for the figure-save-url and show it there (provided `divVis` is specified)
`outRform`	Make a `<div>` for the visualization parameter form and show it there (provided `divVis` is specified)
`events`	list specifying the JavaScript callback function names of events in the SVG figure.

The value given for the `events` key must be an object containing the items shown in Table 3.

Table 3 The obligatory components of the `events` object

Key	Arguments	Event
`fig_mouseover`	evt, svgElmId, intvNum, srchItemNum	Mouse movement over data element (line, bar, point) in the svg figure
`fig_mouseout`	-	Mouse leaves the data element in the svg figure
`fig_click`	evt, svgElmId, intvNum, srchItemNum	Left mouse button click on data element in the svg figure
`vert_mousemove`	evt, numIntvls, numSrchItems, figType	Mouse movement anywhere in the svg figure (Note: the 'standard' function calculates the nearest vertical line through the data points and shows it.)
`vert_mouseout`	evt, numIntvls	Mouse is outside the svg figure

The internal work-flow of the `init()` function is shown in Figure 3.

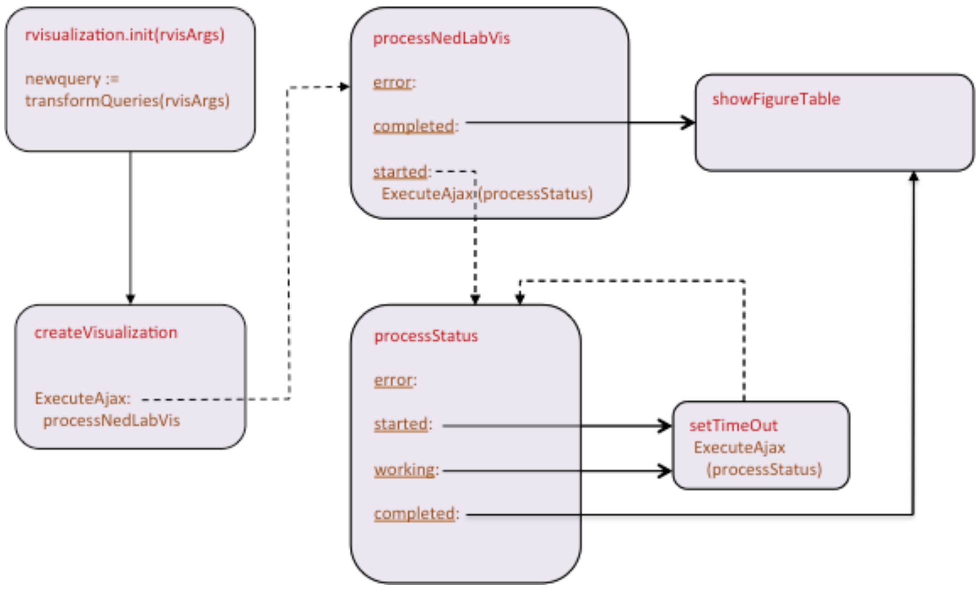

rvisualization.init(rvisArgs)
newquery := transformQueries(rvisArgs)
createVisualization
ExecuteAjax: processNedLabVis
processNedLabVis
error:
completed:
started:
ExecuteAjax(processStatus)
showFigureTable
processStatus
error:
started:
working:
completed:
setTimeOut ExecuteAjax (processStatus)

Figure 3 Control flow from a visualization request to its result

The parameters specified in the object `rvisArgs` are translated into a visualization query, and then passed on to the "private_methods" function `createVisualization()`. This latter function issues an Ajax request to the R-visualization web service. Since this request is asynchronous (hence the dotted line), this is the point where the `rvisualization.init()` function returns to the caller. The visualization, however, has not yet been made.

Note: following up a call to the `init()` function with a call to the `ext_getFigure()` function will *not* lead to correct results!!

The internal JavaScript function "`processNedLabVis`" picks up the "job" number issued to this visualization by the R visualization web service. The job number is used in subsequent Ajax calls to "`processStatus`", which periodically (once every 200 ms) check the progress of the visualization job. Once the status "completed" arrives, the resulting visualization is being gathered in the following way:

1) The internal function "`showFigureTable`" is called, and if the object `rvisArgs` has specified a visualization `<div>` (through "`divVis`") and if it has set the "`outFig`" flag to "`true`", then the svg picture is placed in a `<div>` under `divVis`.
2) If the object `rvisArgs` contains a specification of a callback function (`cb_function` plus `cb_arg`), then this function is called with two arguments: (a) the context name string, and (b) the object `cb_arg`. Since the visualization has been made, this user-defined callback function is free to make a call to the `ext_getFigure()` function, and place the svg code where it wants to have it.

The following example illustrates using the `init()` function with a specified target `<div>` for the visualization:

```
maakVisViaDiv: function (strLine, strDivName, strDivStatus) {
  var formquery = private_methods.getFormQuery(strLine);
  var brokerquery = null;
  var divParent = document.getElementById(strDivName);
  var iWidth = divParent.clientWidth;
  var iHeight = divParent.clientHeight;
  var year_start     = private_methods.getFormNumber(1);
  if (year_start<0) { year_start = 1800; }
  var year_end       = private_methods.getFormNumber(3);
  if (year_end<0) { year_end = 1899; }
  var interval       = private_methods.getFormNumber(4);
  if (interval<0) { interval = 1; }
  var bNorm         = document.getElementById("form").elements[5].checked;
  var search_method = private_methods.getFormString(6);
  var iEnv = nederlab.rvisualization.init({
    formquery: formquery,
    brokerquery: brokerquery,
    divVis: strDivName,    // Specificeer de visualisatie <div>
    divStatus: strDivStatus,  // De naam van de status <div>
    width: iWidth,         // Optionele breedte (binnen "divVis")
    height: iHeight,       // Optionele hoogte (binnen "divVis")
    context: strDivName,   // Sleutelnaam voor dit visualisatieobject op deze pagina
    yrFrom: year_start,    // Optionele visualisatiespecificatie
    yrTo: year_end,        // Optionele visualisatiespecificatie
    interval: interval,    // Optionele visualisatiespecificatie
    norm: bNorm,           // Optionele visualisatiespecificatie
    method: search_method,// Optionele visualisatiespecificatie
    outFig: true,          // Maak visualisatie ergens in divVis aan
    outRform: false        // Maak *geen* <form> aan voor input van parameters
  });
}
```

The following example illustrates using how the `init()` function can be approached using a callback function:

```
  maakVisViaCallback: function (strLine, strDivName, strDivStatus) {
    var formquery = private_methods.getFormQuery(strLine);
    var brokerquery = null;
    var divParent = document.getElementById(strDivName);
    var iWidth = divParent.clientWidth;
    var iHeight = divParent.clientHeight;
    var year_start    = private_methods.getFormNumber(1);
    if (year_start<0) { year_start = 1800; }
    var year_end      = private_methods.getFormNumber(3);
    if (year_end<0) { year_end = 1899; }
    var interval      = private_methods.getFormNumber(4);
    if (interval<0) { interval = 1; }
    var bNorm         = document.getElementById("form").elements[5].checked;
    var search_method = private_methods.getFormString(6);
    var iEnv = nederlab.rvisualization.init({
      formquery: formquery,
      brokerquery: brokerquery,
      cb_function: nederlab.controller.plaatsfig,
      cb_arg: {target: strDivName},
      divStatus: strDivStatus,
      width: iWidth,
      height: iHeight,
      context: strDivName,
      yrFrom: year_start,       // Optionele visualisatiespecificatie
      yrTo: year_end,           // Optionele visualisatiespecificatie
      interval: interval,       // Optionele visualisatiespecificatie
      norm: bNorm,              // Optionele visualisatiespecificatie
      method: search_method,    // Optionele visualisatiespecificatie
      outFig: false,            // Do not make my own output
      outRform: false           // Do not make a form with input parameters
    });
    var dummy = iEnv;
  },
  // Position the figure on the right place
  plaatsfig: function (sContext, objArg) {
    // Get the figure
    var sFig = nederlab.rvisualization.ext_getFigure(sContext);
    // Place the figure on the right spot
    var divTarget = document.getElementById(objArg.target);
    divTarget.innerHTML = sFig;
    // Pas de grootte van de <svg> binnen divTarget aan
    // Get a handle to the <svg>
    var divSvg = divTarget.firstElementChild;
    if (divSvg !== null) {
      divSvg.setAttribute('width', divTarget.clientWidth);
      divSvg.setAttribute('height', divTarget.clientHeight);
    }
  }
```

7.2 Function "nederlab.rvisualization.ext_getFigure"

This JavaScript function serves to collect the SVG object of the figure that has been created in an `rvisualization.init()` call. It takes one argument, which is the context name string as has been used. An example of how this function can be used is here:

```
  // Position the figure on the right place
  plaatsfig: function (sContext, objArg) {
    // Get the figure
    var sFig = nederlab.rvisualization.ext_getFigure(sContext);
    // Place the figure on the right spot
    var divTarget = document.getElementById(objArg.target);
    divTarget.innerHTML = sFig;
    // Pas de grootte van de <svg> binnen divTarget aan
    // Get a handle to the <svg>
    var divSvg = divTarget.firstElementChild;
    if (divSvg !== null) {
      divSvg.setAttribute('width', divTarget.clientWidth);
      divSvg.setAttribute('height', divTarget.clientHeight);
    }
  }
```

Note that once the svg figure object has been collected, it can be put into a div's "innerHTML" without further modifications. The figure does not, however, automatically adapt to the div's size. To get it fitted inside the div, the attributes "width" and "height" of the svg's first element need to be adapted, as shown in the illustration. The figure will adapt its size within the width and height specified, *but it will retain its original width/height correlation*.

7.3 Function "nederlab.rvisualization.ext_getTable"

Provided a call to `rvisualization.init()` has been successful, the function `ext_getTable` returns the values of the visualization in `sContext` as a table object.

An example of a Javascript function that 'interprets' the table object is here:

```
function getTableAsHtml(sContext) {
  // Get the table object
  var objTbl = nederlab.rvisualization.ext_getTable(sContext);
  // Interpret the table and put it in HTML format
  html = [""];
  html.push("<table><tr><th>Zoekterm</th><th>Periode</th><th>Frequentie</th>" +
            "<th>Totaal</th><th>Relatief</th></tr>");
  $.each(objTbl.intvHits, function (index, hit) {
    // Add all relevant info to the table rows
    html.push("<tr><td>" + objTbl.srchTerm[index] + "</td><td>" +
      objTbl.intvNames[index] + "</td><td align='right'>" + objTbl.intvAbs[index] +
      "</td><td align='right'>" + objTbl.intvWords[index] +
      "</td><td align='right'>" + objTbl.intvHits[index] +  "</td></tr>");
  });
  html.push("</table>");
  var sResult = html.join("\n");
  // Return the html table
  return sResult;
}
```

7.4 Function "nederlab.rvisualization.ext_getQlist"

Provided a call to `rvisualization.init()` has been successful, the function `ext_getQlist` returns the values of the visualization in `sContext` as a table object.

An example of a Javascript function that 'interprets' the query list object is here:

```
function getQlistAsHtml(sContext) {
  // Get the query list
  var objTbl = nederlab.rvisualization.ext_getTable(sContext);
  // Interpret the query list and put it in HTML format
  html = [""];
  html.push("<table><tr><th>#</th><th>Broker</th></tr>");
  // Walk the list inside [objQlist]
  for (i in objQlist) {
    // Add each broker query to the row
    html.push("<tr><td>" + i + "</td><td>" +
            objQlist[i] + "</td></tr>");
  }
  html.push("</table>");
  // Combine the result as a string
  var sResult = html.join("\n");
  // Return the html table containing the list of queries
  return sResult;
}
```

7.5 Function "nederlab.rvisualization.ext_initSaving"

Provided a call to `rvisualization.init()` has been successful, the function `ext_initSaving` *starts* the process of creating a picture file (such as jpeg or bmp) on the "R" visualization web server. The function takes three arguments, as shown in X.

Table 4 Arguments to the `ext_initSaving` function

Argument	Explanation
`sFormat`	The kind of picture that needs to be made. The choice is between the following kinds: `jpeg` JPEG raster image (compressed) `png` portable network graphics (compressed raster) `bmp` BMP raster image `tex` TeX/LaTeX formatted line drawing `eps` (Extended) PostScript format `pdf` PDF document `svg` scalable vectore graphics picture
`sContext`	The unique string identifying the visualization context on the user's page/tab.
`fn_endSaving`	A function that needs to be called *once the picture has been created*. This function receives one argument, which is an object that may contain the following three key-value pairs: `status` Either "completed" or "error" `message` An error message, should the status be "error" `location` The URL of the picture that has been created

An example of a Javascript function that triggers the creation of a downloadable picture and uses a callback function to process the resulting URL is here:

```
// Plaatje maken
 function  haalop_start(strTarget, strFormat) {
    // Zet the URL in de juiste div
    var divDownload = document.getElementById("download");
    // Initialiseer
    divDownload.innerHTML = "...";
    // Laat een plaatje maken van strTarget
    nederlab.rvisualization.ext_initSaving(strFormat, strTarget,
                                 nederlab.controller.haalop_einde);
```

```
  }

  // Beeindig ophalen plaatje
 function haalop_einde(objArg) {
    // Zet the URL in de juiste div
    var divDownload = document.getElementById("download");
    var sUrl;
    // Controleer de status
    if ( objArg === undefined || objArg === null ||
         objArg.status === undefined || objArg.status === null) {
      sUrl = "(fout)";
    } else {
      if (objArg.status === "completed") {
        var sCombi = "<a href='" + objArg.location +
                     "'>figuur downloaden (" + objArg.format + ")</a>";
        sUrl = sCombi;
      } else {
        sUrl = objArg.status + ": " + objArg.message;
      }
    }
    // Process the result
    divDownload.innerHTML = sUrl;
  }
```

7.6 Function "nederlab.rvisualization.ext_changevis"

Provided a call to `rvisualization.init()` has been successful, the function `ext_changevis` issues a new call to the R visualization web server, but with adapted visualization specifications. The function takes two arguments:

Table 5 Arguments to the `ext_ changevis` function

Argument	Explanation	
`sContext`	The unique string identifying the visualization context on the user's page/tab.	
`oVisArgs`	An object containing a revision of the visualization specifications. This object may contain the following elements:	
	`interval`	The number of years for each interval
	`norm`	Boolean specifying whether normalization is used or not
	`vis`	The kind of visualization (bar, line, point, linepoint, smooth)
	`method`	The counting method (docfreq, termfreq)

Execution of this javascript function results either (a) in a changed svg picture in the `divVis` specified in the original call to `rvisualization.init()`, or (b) in a changed svg picture that is made available to the callback function specified in the `rvisArgs` argument of the original call to `rvisualization.init()`.

7.7 Function "nederlab.rvisualization.ext_prevquery"

This JavaScript function serves to return to the preceding SVG figure, as has been created with a previous call to `rvisualization.init()`, or to `rvisualization.changevis`. It takes one argument, which is the context name string as has been used. The picture is either taken from cache, or else re-calculated, and it is re-drawn according to the specifications of a preceding `rvisualization.init()` call.

7.8 Event handling functions

There are about five standard event-handling functions that are used inside the SVG visualizations made by the R webservice. All of these functions are part of the “exposed” functionality of rvisualization.js.

7.8.1 *Mouse click:* `nlabRclick()`

This event is triggered whenever a visualization data part (that is: a data line, a data point or a data bar) is clicked by the user. The current action depends on the kind of visualization:

Visualization	Action
line	Change the thickness of the line. Toggle between 1.42 and 2.5.
bar	Visual reaction: Change the opacity of the bar (toggle between 0.5 and 1.0) Data reaction: Issue a "createVisualization" request where the visualization are: - Year from and to take the interval of the bar - Interval = 1 year - Visualization = "linepoint"
point	Change the size and opacity of the point. Toggle between large/small.

7.8.2 *Mouse enters visualization part:* `nlabRshowTip()`

This function is triggerd by a "mouseover" event, when the mouse finds itself over a data line, data point or data bar. The action depens on the kind of visualization:

Visualization	Action
line	1) Create and show a "tooltipRect" + "tooltipText" element containing the text of the search term. The "rect" is temporarily added physically to the SVG. 2) Set the same text in the "divStatus"
bar, point	1) Create and show a "tooltipRect" + "tooltipText" element containing: - Absolute frequency for the selected interval - Number of words or documents for the interval - Years of the interval 2) Set the same text in the "divStatus"

7.8.3 *Mouse leaves visualization part:* `nlabRhideTip()`

This function is triggerd by a "mouseout" event, when the mouse leaves a data line, data point or data bar. Any "tooltipText" and "tooltipRect" elements that were created under the mouseover event (see 7.8.2) are removed.

7.8.4 *Show vertical bar:* `nlabRshowVert()`

Line, point and smooth figures contain vertical lines that are normally not visible. Any mouse move within the figure triggers a call to "`nlabRshowVert`", which looks for the position of the mouse, calculates what the nearest vertical line is, and then shows it (by changing the line's opacity).

7.8.5 *Hide vertical bar:* `nlabRhideVert()`

Line, point and smooth figures contain vertical lines that are normally not visible. When the mouse moves *outside* the figure area, a call to `nlabRhideVert` is made, which should hide (make opaque) any remaining vertical lines.

8 The “R” code

The “R” code comes in the form of one file called “NedLabVisGG.r”. This file is packaged and sub-versioned in the “src/R” subdirectory of the Java code for the R-webservice (although it functions independently of that Java code). It is located in the Nederlab home directory (currently /home/nederlab) on the Linux server where the R-webservice is running.

8.1 The structure of NedLabVisGG.r

The entrance to the “R” code from Java is the “R” function “nlabvis()”, which takes a JSON-coded string as argument. The flow from this “R” function to the other, internal, “R” functions is depicted graphically in Figure 4.

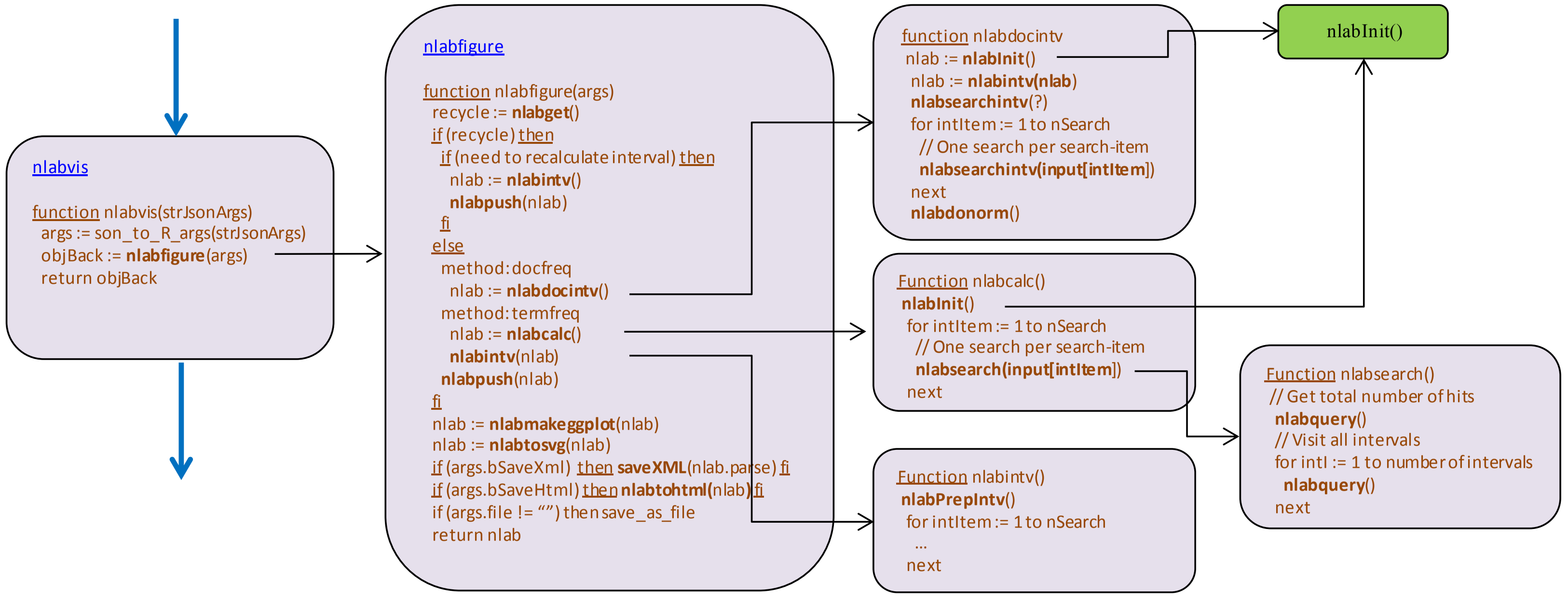
The “nlabvis()” function mainly serves to translate the JSON specification of the search request into individual parameters, which are then passed on to the “nlabfigure()” function. This function ultimately returns an object that either contains the required SVG picture (as well as a table of results and a table of used broker-queries), or an error object containing an error specification.

The “nlabfigure()” function is the core of the “R” functionality, as can be seen by a pseudocode representation of it:

```
function nlabfigure(args)
  recycle := nlabget()
  if (recycle) then
    if (need to recalculate interval) then
      nlab := nlabintv()
      nlabpush(nlab)
    fi
  else
    method: termfreq
      nlab := nlabcalc()
      nlabintv(nlab)
    method: docfreq
      nlab := nlabdocintv()
    nlabpush(nlab)
  fi
  nlab := nlabmakeggplot(nlab)
  nlab := nlabtosvg(nlab)
  if (args.bDoSaveXml)  then saveXML(nlab.parse) fi
  if (args.bDoSaveHtml) then nlabtohtml(nlab) fi
  if (args.file != “”) then save_as_file
  return nlab
end function
```

What “nlabfigure()” first does is check if the requested visualization has been requested before, since every “R” connection (a connection is one in terms of “Rserve”, an “R” handling service running locally on the Linux computer) holds a small cache of its most recent search results. The size of this cache is currently set to “10”, but experiments may have it longer. (Its size is set using a *global* variable in the NedLabVisGG.r code.)

Having decided a fresh visualization calculation is required, “nlabfigure()” acts, depending on the “method” that has been chosen by the user. This method is either “termfreq” (calculate the number of search hits in all texts) or “docfreq” (calculate the number of documents containing at least one search hit). If the “termfreq” method is chosen, the functions “nlabcalc()” and “nlabintv()” are called consecutively.



```
nlabvis

function nlabvis(strJsonArgs)
  args := son_to_R_args(strJsonArgs)
  objBack := nlabfigure(args)
  return objBack
```

```
nlabfigure

function nlabfigure(args)
  recycle := nlabget()
  if (recycle) then
    if (need to recalculate interval) then
      nlab := nlabintv()
      nlabpush(nlab)
    fi
  else
    method:docfreq
      nlab := nlabdocintv()
    method:termfreq
      nlab := nlabcalc()
      nlabintv(nlab)
    nlabpush(nlab)
  fi
  nlab := nlabmakeggplot(nlab)
  nlab := nlabtosvg(nlab)
  if (args.bSaveXml)  then saveXML(nlab.parse) fi
  if (args.bSaveHtml) then nlabtohtml(nlab) fi
  if (args.file != "") then save_as_file
  return nlab
```

```
function nlabdocintv
 nlab := nlabInit()
  nlab := nlabintv(nlab)
  nlabsearchintv(?)
  for intItem := 1 to nSearch
    // One search per search-item
    nlabsearchintv(input[intItem])
  next
  nlabdonorm()
```

```
Function nlabcalc()
 nlabInit()
  for intItem := 1 to nSearch
    // One search per search-item
    nlabsearch(input[intItem])
  next
```

```
Function nlabintv()
 nlabPrepIntv()
  for intItem := 1 to nSearch
    …
  next
```

```
Function nlabsearch()
 // Get total number of hits
  nlabquery()
  // Visit all intervals
  for intI := 1 to number of intervals
    nlabquery()
  next
```

Figure 4 Flow chart of the "R" functions inside NedLabVisGG.r

The former serves to initialize and execute the search for hits via the broker interface (the broker search makes "function: termfreq" requests to all documents in the time interval and divides the results into years within the "R" code), while the latter takes the results and divides them properly into intervals. It is only the interval data that results in a visualization and that gets returned to the user.

Note that the data calculated per year stays available in the cache, so that consecutive calls on the same "connection" can re-use these data, if the only difference in the search request is a difference in interval size.

The "`nlabcalc()`" function defers the actual searching to the "`nlabsearch()`" function. Here is the pseudocode of `nlabcalc()`, `nlabsearch()` and `nlabintv()` respectively:

nlabcalc

```
nlabInit()
for intItem := 1 to nSearch
  // Do one search for each search-item
  nlabsearch(input[intItem])
next
```

nlabsearch

```
// Get the stats: total number of hits
nlabquery()
// Visit all intervals
for intI := 1 to number of intervals
  nlabquery()
next
```

nlabintv

```
nlabPrepIntv()
for intItem := 1 to nSearch
  …
next
```

If the "`docfreq`" method is chosen, the broker can be requested to give all resulting hits per interval, so that requests are formulated that result in a faceted response. This is handled through the "`nlabdocintv()`" function. This function makes use of the "`nlabsearchintv()`" one, which contains a call to "`nlabquery()`". Here is the pseudocode of "`nlabdocintv()`":

```
nlab := nlabInit()
nlab := nlabintv(nlab)
nlabsearchintv(?)
for intItem := 1 to nSearch
  // Do one search for each search-item
  nlabsearchintv(input[intItem])
next
nlabdonorm()
```

Once the search results have been calculated the main function "`nlabfigure()`" continues its course of action by making a call to "`nlabmakeggplot()`". This latter function calls the "`grid`" package through "`R`" functions such as "`qplot()`" (a simplification of "`ggplot()`"), resulting in a grid graphical object.

The function "`nlabtosvg()`" converts this graphical object into `SVG`, but it does more than that. The resulting `SVG` needs to be tweaked in some places (in order to be compliant with all current browsers), and callback functions need to be added. The function also adds vertical lines that are normally invisible, unless activated by mouse movement. This latter functionality depends on the calling JavaScript and it making use of the correct event-driven functions.

The `nlabfigure()` function contains a few more optional actions: saving the `SVG` picture straight away, saving the `SVG` as `HTML` straight away and saving a copy of the graphical object in another format (such as `JPEG`, `PDF` and so on). These options are not activated by default.

The nlabfigure() returns its result to the calling function, which is nlabvis() in the case of the R visualization webservice.

8.2 Normalization

Depending on the search method chosen by the user of the webservice, "R" is able to normalize the resulting data.

8.2.1 Document frequency normalization

Normalization of data found using the "docfreq" search method is done "on the fly". That is to say, an additional query is fired to the broker in order to find out what the amount of documents (satisfying the "condition" and "filter" part) per interval is. This amount is then used to calculate a relative frequency (per "intNorm" documents, where "intNorm" can be set in the parameters to the "R" webservice).

In sum, the normalization calculation for the "docfreq" method is dynamic (it adjusts to the conditions and filters) and broker-independent, which means that it can be freely used and that it gives the desired results.

The location where document frequency normalization is handled is within the "R" code module "NedLabVisGG.r". The docfreq search is handled in the "nlabdocintv()" function. This function makes use of the "nlabsearchintv()" one, which contains a call to "nlabquery()". The first call to the "nlabsearchintv()" function is one with "?" as an argument, which signals the "nlabsearchintv()" function to adapt the "condition" part of the broker query by using the function "cond_docfreq_adapt()". This latter function is a recursive one that looks for "type" parts called "equals", "wildcard" or "phrase" (provided they are part of a "list"). An additional requirement for the "type" parts called "equals" is that the "field" value must be one of the known 'searchable' fields within the broker interface (content and title fields). This requirement is handled internally by the function "is_search_field()", and the kinds of fields fulfilling this requirement are shown in Table 6.

Field	Comments
NLContent*	These contain, for instance: NLContent_text_lowercase NLContent_text NLContent_ticcl_lowercase Excluded: NL*_available
NL*_title	
NL*_subtitle	

Table 6 Fields that are known to contain searchable text

These type parts, then, are all changed to "wildcard" types with "?" as value. There is some additional filtering that makes sure the "condition" only consists of "type", "list", "field" and "value" elements. The broker query that has been adapted in this way (and that is extended with an appropriate facetrange response part) yields the total amounts of documents (that have at least some content) broken down over the facet ranged year intervals, and filtered by any "filter" part of the main broker query.

8.2.2 Term frequency normalization

Normalization for the "termfreq" search method works differently, and currently uses a static method. The termfreq normalization option should no longer be used, since it makes use of outdated static data. The data it uses currently is a list of the number of words available per year in the total Nederlab corpus based on the index that was used up to January 2015. Even if this list would be adapted to the current (or another future) state of the index available under the broker, there still is a fundamental problem: this type of normalization does not take the user's "`condition`" and "`filter`" specifications into account.

Future development of the Nederlab project should, therefore, seek to find a dynamic method to calculate the total number of words in the documents selected by the user's "`condition`" and "`filter`" specifications.

8.3 Manual pages for selected "R" functions

The subsections here contain the manual pages for the most important "R" functions that are part of the "NedLabVisGG.r" package.

8.3.1 Function `nlabvis`

Description

Determine the frequency of occurrence per time-period of a number of 'search items'. The search items may be single words or phrases. (No wildcards are allowed.) All the documents (or just the titles of documents) that are available in "Nederlab" are searched through. The frequency is the number of times the search item actually occurs in the documents (or just their titles) available for a particular time-period. The results can be visualized in a number of different formats (see under arguments), and they can also be stored in a file. The function `nlabvis` is an intermediate function between the "Nederlab Onderzoeksportaal", which can issue a search request to `nlabvis` in a JSON format (see below), which the function `nlabvis` translates into a call to `nlabfigure`.

Usage

```
nlabvis(dtJson, sSessionUserJob = "", sTmpDir = "", bCheckPkg = T, debugL=0)
```

Arguments

`dtJson`	a string in the JSON format, containing the specification of the search
`sSessionUserJob`	unique string code from the caller
`sTmpDir`	which directory to use for important temporary files (.grid, .log)
`bCheckPkg`	flag telling nlabvis to check and load libraries (T) or not (F)
`debugL`	debugging level: 0, 1, 2

Value

An object which is a key-value pair list. (If there has been an error, the only member of the list is "error".)

`svg`	an SVG (xml) representation of the figure visualizing the requested data.
`tbl`	the nlab$intvData section (see "nlab")
`info`	
`qlist`	the list of broker queries (the nlab$lQuery section)
`error`	more specific error message

See Also

`nlabfigure`.

Examples

```
## Look for three terms (oorlog, vrede and staking) between 1800 and 1895,
##   then produce a 'pointline' figure of the results, divided over 5-year
   intervals,
##   and normalize the (term) frequencies per 10.000 words found in these periods.
## Use the indicated session-user-job-string, and the /home/nederlab/tmp directory
##   as a temporary one to store .grob and .log files. Do *not* check whether
##   the required "R" packages have been loaded.
## Upon completion, get the "svg" picture into the variable 'svgOut'.
> dtJson <- '{
    "srchTerms": [ "oorlog", "vreede", "staaking"],
    "colors":    [ "red",    "gold",   "darkblue"],
    "labels":    [ "oorlog", "vrede",  "staking"],
    "source":    "content",
    "method":    "termfreq",
    "vis":       "pointline",
    "yrFrom":    1800,
    "yrTo":      1895,
    "interval":  5,
    "norm":      true,
    "legend":    true,
    "intNorm":   10000,
    "server":    "nederlab",
    "debugL":    0
}'
> oBack <- nlabvis(dtJson, 'sessionxd4567_bottom-left', '/home/nederlab/tmp', F)
> svgOut <- oBack$svg
```

8.3.2 Function `nlabfigure`

Description

Determine the frequency of occurrence per time-period of a number of 'search items'. The search items may be single words or phrases. (No wildcards are allowed.) All the documents (or just the titles of documents) that are available in "Nederlab" are searched through. The frequency is the number of times the search item actually occurs in the documents (or just their titles) available for a particular time-period. The results can be visualized in a number of different formats (see under arguments), and they can also be stored in a file. The function `nlabfigure` calls a chain of other functions to do the actual work: `nlabcalc` communicates with the Nederlab 'broker' and creates the initial `nlab` data structure; `nlabintv` divides the frequencies over periods and calculates the normalized frequencies; `nlabmakeggplot` uses the `ggplot2` package to produce a grid structure plot; `nlabtosvg` converts the grid structure into SVG (using `gridSVG`), supplementing it with JavaScript event calls. The function `nlabfigure` returns an SVG picture that can be embedded in an <html> page.

Usage

```
nlabfigure(srchTerms, cnds=NULL, flts=NULL, intFrom=1800, intTo=2020,
        labels=NULL, colors=NULL, legend=T, width=7, height=7, strType="content",
    strLine="line",
        interval=1,  norm=T, intNorm=1000, intEnv=0, method="termfreq", file="",
    server="nederlab",
    timeorder="nederl_time_order", events=NULL, show=T, dotime=F, stats="",
    user="e",
        bCheckPkg = T, bSaveSvg = F, bSaveHtml = F, debugL=0)
```

Arguments

`srchTerms`	a vector of search items, each of which may be a single word or a phrase. For example: `c("de oorlog", "vrede van").`
`cnds`	a vector of broker "condition" parts - one for each search item
`flts`	a vector of broker "filter" parts - one for each search item
`intFrom`	the first year to include in the search.
`intTo`	the last year to include in the search.
`labels`	a vector of short string labels - one for each search item
`colors`	a vector of figure colors - one for each search item
`legend`	show a legend next to the figure (T) or not (F)
`width`	width of the canvas in inches
`height`	height of the canvas in inches
`strType`	location to search: "title" or "content". Default value is "content".
`strLine`	the kind of graph that is to be produced. Only a limited selection of `ggplot` types and combinations are allowed: `point` (one point for each hit), `pointline` (one point for each hit, and a line connecting the points), `smooth` (one point for each hit, an approximate smooth line through the points, and a half opaque background area), `line` (a simple line without clear points), `bar` (side-by-side bars), `barstack` (bars stacked upon one another), `linebar` (side-by-side bars with a line through the data points)
`interval`	the number of years that should be grouped together. Default is 1 (one), which means that no grouping is done.
`norm`	flag to indicate whether normalization of the data should be done (T) or not (F). If set, the number of hits is divided by the total number of words available in the

	Nederlab corpus for the indicated time interval, and this is then multiplied by `intNorm`
`intNorm`	if `norm` is set to TRUE, then `intNorm` specifies the amount of words that serve as basis for the "normalized frequency" given in the figure. The default figure of 1000 indicates that the frequencies are "per 1000 words".
`intEnv`	number of the user's page environment that is being used. This is used to keep apart multiple figures on one user's display.
`method`	nlabfigure allows for a number of different calculation methods: `termfreq` (even though an interval may be specified, information is calculated for all the years), `docfreq` (only information for the requested intervals is calculated, making use of a facet query)
`file`	relative or full path of the file where the graph should be saved to. The format of the graph is automatically determined by the extension of the file name. Recognized extensions are: `.pdf`, `.wmf` (windows meta file), `.png`, `.jpeg`, `.bmp` (bitmap), `.svg`, `.tex` (pictex), `.tiff` and `.ps`/`.eps` (postscript/extended postscript). The windows meta file format `.wmf` allows adaptation of the figure's components.
`server`	specify which server is to be used: "nederlab" or "radboud". The "radboud" server does not connect to the Nederlab information; it gives random data back. It can be used for test-purposes when the Nederlab server is out of order.
`timeorder`	name of the date/year field to be used in the broker queries
`events`	list of callback function names to be used in the resulting SVG figure
`show`	show the resulting plot on the server’s "R" panel (since the server does not support plotting, this has no effect)
`dotime`	time is measured (T) or not (F). If set, the "response" parts in the broker queries are extended with "cache=F".
`stats`	not used
`user`	unique session/user string
`bCheckPkg`	check and load "R" libraries (T) or not (F)
`bSaveSvg`	save the svg version of the figure (T) or not (F)
`bSaveHtml`	save an html version of the figure (T) or not (F)
`debugL`	level of debugging: 0, 1, 2.

Value

An "R" object is returned, which is a list containing the following elements:

`years`	the names of the time-intervals
`freqabs`	the absolute frequencies per interval
`freqnorm`	the normalized frequencies per interval
`docwords`	the number of words in all texts for this interval
`nlab`	a data structure of the nlab type

See Also

`ggplot`, `nlab`, `nlabcalc`, `nlabintv`, `nlabmakeggplot`, `nlabtosvg`.

Examples

```
## Look for the word "oorlog" between 1850 and 1918:
nlabfigure(c("oorlog"), intFrom=1850, intTo=1918)
```

```
## Compare the occurrences of "oorlog" and "vrede" between 1820-1920, taking
    intervals of 10 years,
##   give normalized frequencies, use a bar plot:
nlabfigure(c("oorlog", "vrede"), intFrom=1820, intTo=1920, interval=10, norm=T,
    strLine="bar")

## Look for the phrase "de oorlog" between 1800-1850, and save the results in a
    .wmf file:
nlabfigure("oorlog", intFrom=1850, intTo=1918, file="oorlog_1850-1918.wmf")

## Compare "Frankrijk" and "Duitschland" between 1800-1880 in 15 year intervals,
##   return the frequency per 100.000 words, and put the results in list "t":
> t <- nlabfigure(c("frankrijk", "duitschland"), intFrom=1800, intTo=1880,
    interval=15, strLine="bar", intNorm=100000, norm=T)
> t$nlab$intvData
      srchTerm srchId intvNames intvYears  intvHits intvAbs intvWords
1    frankrijk      1 1800-1814      1800 10.717305    1813  16916566
2    frankrijk      1 1815-1829      1815 21.976168    3871  17614536
3    frankrijk      1 1830-1844      1830 17.473356    5787  33118995
4    frankrijk      1 1845-1859      1845 21.134211   10573  50027890
5    frankrijk      1 1860-1874      1860 18.526302   12818  69188119
6    frankrijk      1 1875-1880      1875 20.896434    4911  23501617
7  duitschland      2 1800-1814      1800  7.560636    1279  16916566
8  duitschland      2 1815-1829      1815  8.935802    1574  17614536
9  duitschland      2 1830-1844      1830  8.445305    2797  33118995
10 duitschland      2 1845-1859      1845 11.859385    5933  50027890
11 duitschland      2 1860-1874      1860 11.698540    8094  69188119
12 duitschland      2 1875-1880      1875 10.429070    2451  23501617
```

8.3.3 Function `nlabcalc`

Description

Count absolute frequencies of occurrence of one or more search items. Counting is done within the specified subset of the historical Dutch texts that are made available in the Nederlab project through the `broker` interface. The subset restricts the search to the texts within a time frame. The `nlabcalc` function can only be called from within "R" and it returns an "R" `nlab` object.

Usage

```
nlabcalc(srchTerms, cnds, flts, strType="content", intFrom=1800, intTo=2014,
    labels=NULL, colors=NULL, legend=T, width, height, server="nederlab",
    timeorder="nederl_time_order", events=NULL, dotime=F, user="", bCheckPkg = T,
    debugL=0)
```

Arguments

`srchTerms`	a vector of search items, each of which may be a single word or a phrase. For example: `c("de oorlog", "vrede van").`
`cnds`	a vector of broker "condition" parts - one for each search item
`flts`	a vector of broker "filter" parts - one for each search item
`strType`	search field to be used: "title" or "content" (nederlab). Default value is "content". Nederlab2 uses other search fields.
`intFrom`	the first year to include in the search.
`intTo`	the last year to include in the search.
`labels`	a vector of short string labels - one for each search item
`colors`	a vector of figure colors - one for each search item
`legend`	show a legend next to the figure (T) or not (F)
`width`	width of the canvas in inches
`height`	height of the canvas in inches
`server`	specify which server is to be used: "nederlab", "nederlab2", or "radboud". The "radboud" server does not connect to the Nederlab information; it gives random data back. It can be used for test-purposes when the Nederlab server is out of order.
`timeorder`	name of the date/year field to be used in the broker queries
`events`	list of callback function names to be used in the resulting SVG figure
`dotime`	time is measured (T) or not (F). If set, the "response" parts in the broker queries are extended with "cache=F".
`user`	unique session/user string
`bCheckPkg`	check and load "R" libraries (T) or not (F)
`debugL`	level of debugging: 0, 1, 2.

Value

An "R" object of type `nlab` containing absolute frequency counts for the search terms as well as numbers of words in Nederlab documents for the specified years. The interval-directed parts of the returned object are created and initialized.

See Also

`nlab`, `nlabfigure`.

Examples

```
## Calculate the occurrences of the word "oorlog" from 1850-1918:
```

```
nli <- nlabcalc(c("oorlog"), intFrom=1850, intTo=1918)

## Calculate the frequencies of "oorlog" and "vrede" from 1820-1920
##   but do this in the \strong{titles} of the documents:
nli <- nlabcalc(c("oorlog", "vrede"), intFrom=1820, intTo=1920, strType="title")

## Check the structure of what is returned after calculation
> nli <- nlabcalc(c("frankrijk", "duitschland"), intFrom=1800, intTo=1880)
> str(nli)
List of 13
 $ input  : chr [1:2] "frankrijk" "duitschland"
 $ cCol   : chr [1:2] "red" "blue"
 $ cName  : chr [1:2] "frankrijk" "duitschland"
 $ nYears : int 81
 $ nSearch: int 2
 $ yrFirst: num 1800
 $ yrLast : num 1880
 $ years  : int [1:81] 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 ...
 $ fig    : NULL
 $ parse  : NULL
 $ norm   : logi FALSE
 $ intNorm: num 1
 $ yrData :'data.frame': 162 obs. of  5 variables:
  ..$ srchTerm: chr [1:162] "frankrijk" "frankrijk" "frankrijk" "frankrijk" ...
  ..$ srchId  : int [1:162] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ year    : int [1:162] 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 ...
  ..$ words   : num [1:162] 1205922 945174 1081396 1012319 1193755 ...
  ..$ hits    : num [1:162] 0 30 12 0 0 212 154 203 314 140 ...
```

8.3.4 Function `nlabintv`

Description

The `nlabintv` function takes an `nlab` data object as input. This data object must contain the absolute frequencies of occurrence of one or more search terms, as they have been calculated by `nlabcalcl`. Supplied with further specifications, the `nlabintv` function calculates interval boundaries, absolute frequencies per interval as well as normalized frequencies per interval. The output of the `nlabintv` function can function as input to the `nlabmakeggplot` function.

Usage

```
nlabintv(nlab, interval=1,  norm=T, intNorm=1000, debugL=0)
```

Arguments

`nlab`	An "R" object of type `nlab` containing absolute frequency counts for the search terms as well as numbers of words in Nederlab documents for the specified years.
`interval`	the number of years that should be grouped together. Default is 1 (one), which means that no grouping is done.
`norm`	flag to indicate whether normalization of the data should be done (T) or not (F). If set, the number of hits is divided by the total number of words available in the Nederlab corpus for the indicated time interval, and this is then multiplied by `intNorm`
`intNorm`	if `norm` is set to TRUE, then `intNorm` specifies the amount of words that serve as basis for the "normalized frequency" given in the figure. The default figure of 1000 indicates that the frequencies are "per 1000 words".
`debugL`	level of debugging: 0, 1, 2.

Value

The "R" object of type `nlab` as has been supplied for the input. The object now contains:

`frequencies per year`	counts of occurrances for the search terms as well as numbers of words in Nederlab documents for the specified years.
`frequencies per interval`	counts of occurrances for the search terms as well as numbers of words in Nederlab documents for the specified intervals.
`normalization per interval`	the frequency of occurrence of the search terms per `intNorm` words of texts that are available in the specified intervals.

See Also

`nlabfigure`, `nlabcalc`, `nlabmakeggplot`.

Examples

```
## Calculate the frequency of occurrance of "oorlog" between 1800-1880,
##   and then draw bar graphs
> nlab <- nlabcalc(c("oorlog"), intFrom=1800, intTo=1880)
> nIntv   <- 0
> # Loop through the data with an increasing bin size
> for (yr in nlab$yrFirst:nlab$yrLast) {
+     # calculate bin size
+     bin = yr - nlab$yrFirst + 1
+     # Calculate the division
+     nli <- nlabintv(nlab, bin, T, 10000)
+     # Check if the number of intervals differs
+     if (nli$nIntervals != nIntv) {
+         nIntv <- nli$nIntervals
+         # Tell what we are doing
+         cat("Intervals: ", nIntv, " bin size = ", bin, "\n")
+         # create a figure
+         nli <- nlabmakeggplot(nli, "bar")
+         # show the figure
+         print(nli$fig, newpage=F)
+     }
+ }

## Calculate the frequency of occurrence of "frankrijk" and "duitschland" between
    1800-1880,
##   and then divide the data in 5-year intervals and calculate the frequencies per
    10.000 words.
> nli <- nlabcalc(c("frankrijk", "duitschland"), intFrom=1800, intTo=1880)
> nli <- nlabintv(nli, 5, norm=T, intNorm=10000)
> str(nli)
List of 16
 $ input     : chr [1:2] "frankrijk" "duitschland"
 $ cCol      : chr [1:2] "red" "blue"
 $ cName     : chr [1:2] "frankrijk" "duitschland"
 $ nYears    : int 81
 $ nSearch   : int 2
 $ yrFirst   : num 1800
 $ yrLast    : num 1880
 $ years     : int [1:81] 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 ...
 $ fig       : NULL
 $ parse     : NULL
 $ norm      : logi TRUE
 $ intNorm   : num 10000
 $ yrData    :'data.frame':       162 obs. of  5 variables:
  ..$ srchTerm: chr [1:162] "frankrijk" "frankrijk" "frankrijk" "frankrijk" ...
  ..$ srchId  : int [1:162] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ year    : int [1:162] 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 ...
  ..$ words   : num [1:162] 1205922 945174 1081396 1012319 1193755 ...
  ..$ hits    : num [1:162] 0 30 12 0 0 212 154 203 314 140 ...
 $ nIntervals: num 17
 $ intvYears : num [1:17(1d)] 1800 1805 1810 1815 1820 ...
 $ intvData  :'data.frame':       34 obs. of  7 variables:
  ..$ srchTerm : chr [1:34] "frankrijk" "frankrijk" "frankrijk" "frankrijk" ...
  ..$ srchId   : int [1:34] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ intvNames: chr [1:34] "1800-1804" "1805-1809" "1810-1814" "1815-1819" ...
  ..$ intvYears: num [1:34] 1800 1805 1810 1815 1820 ...
  ..$ intvHits : num [1:34] 0.0772 1.6291 1.4389 2.4349 2.3694 ...
  ..$ intvAbs  : num [1:34] 42 1023 748 1328 1410 ...
  ..$ intvWords: num [1:34] 5438566 6279524 5198476 5454026 5950887 ...
```

8.3.5 Function `nlabmakeggplot`

Description

The `nlabmakeggplot` function takes an `nlab` data object as input. This data object must contain the search terms, the interval boundaries, the absolute frequencies per interval as well as normalized frequencies per interval. The output of the `nlabmakeggplot` is an enhanced form of the `nlab` data object, where nlab$fig contains the grid object representation of the type of visualisation chosen by the `strLine` input parameter.

Usage

```
nlabmakeggplot(nlab, strLine="line", theme="default", debugL=0)
```

Arguments

`nlab`	An "R" object of type `nlab` containing absolute frequency counts for the search terms as well as numbers of words in Nederlab documents for the specified years.
`strLine`	the kind of graph that is to be produced. Only a limited selection of `ggplot` types and combinations are allowed: `point` (one point for each hit), `pointline` (one point for each hit, and a line connecting the points), `smooth` (one point for each hit, an approximate smooth line through the points, and a half opaque background area), `line` (a simple line without clear points), `bar` (side-by-side bars), `barstack` (bars stacked upon one another), `linebar` (side-by-side bars with a line through the data points)
`theme`	default (background) theme for "ggplot" figures. Any other name here triggers using the "theme_bw()". See the "ggplot" documentation on rcran.
`debugL`	level of debugging: 0, 1, 2.

Value

The "R" object of type `nlab` as has been supplied for the input. The object now contains:

`frequencies per year`	counts of occurrances for the search terms as well as numbers of words in Nederlab documents for the specified years.
`frequencies per interval`	counts of occurrances for the search terms as well as numbers of words in Nederlab documents for the specified intervals.
`normalization per interval`	the frequency of occurrence of the search terms per `intNorm` words of texts that are available in the specified intervals.
`grid object`	the grid object representation of the type of visualisation chosen by the `strLine` input parameter.

See Also

`nlabfigure`, `ggplot`.

Examples

```
## Calculate the frequency of occurrance of "oorlog" between 1800-1880,
##   and then draw a 'smooth' line through the 5-year interval points
> nlab <- nlabcalc(c("oorlog"), intFrom=1800, intTo=1880)
> nli <- nlabintv(nlab, 5, T, 10000)
> nli <- nlabmakeggplot(nli, "smooth")
```

8.3.6 Function `nlabtosvg`

Description

Convert the information contained in a Nederlab data structure into an `SVG` (scaled vector graphics) type `xml` format. The Nederlab data structure `nlab` contains a grid object produced through `ggplot`, which is a visualisation of search terms and their frequency of occurrence in specified time intervals. The `nlabtosvg` function converts the grid object to SVG (xml) and adds event calls to `JavaScript` functions.

Usage

```
nlabtosvg(lstNlab, user, debugL=0)
```

Arguments

`lstNlab`	an `nlab` data structure containing count data (frequency of occurrence for specified search terms) as well as a grid object (a visualisation of the count data).
`user`	unique string code for current user/session
`debugL`	level of debugging: 0, 1, 2.

Value

SVG (xml) representation of the graphical object.

See Also

`nlab`, `nlabfigure`.

Examples

```
## Get an SVG representation of the current count data for user "e":
nlabtosvg(nlab, "e")
```

8.3.7 Object `nlab`

Description

The `nlab` data structure consists of the input parameters to a search in the Nederlab documents the results of that search, and the information that is needed to make a visualisation of the results. The `nlab` structure undergoes a gradual enrichment as it is passed around through the four functions it serves: (1) `nlabcalc` generates the `nlab` structure and fills its `yrData` data frame, (2) `nlabintv` enriches it with visualisation interval parameters and fills the `$intvData` data frame, (3) `nlabmakeggplot` creates a graphical object and adds it to `$fig` within `nlab`, and (4) the `nlabtosvg` function creates an SVG xml representation of the figure, adding it within `nlab` to the `$parse` element.

Usage

`nlab`

Format

The main format is a list:

`input`	vector	Search items
`cnds`	vector	Conditions (or NULL)
`colors`	vector	Colors (or NULL)
`cName`	vector	Same as Search
`legend`	logical	Show legend or not?
`method`	chr	docfreq or termfreq
`width`	numeric	width of figure (inches)
`height`	numeric	height of figure (inches)
`nYears`	integer	Number of years to cover
`nSearch`	integer	Number of search items
`source`	chr	Broker field to search (content, title)
`server`	chr	broker server (nederlab, nederlab2)
`timeorder`	chr	field holding the doc date/year
`yrFirst`	numeric	First year
`yrLast`	numeric	Last year
`yrIntv`	numeric	Years per interval
`years`	int array	Every year spelled out
`figType`	chr	The kind of graphical object (line/bar)
`fig`	grid	Graphical object
`parse`	xml	SVG representation of $fig
`events`	list	callback functions
`norm`	logical	Use normalisation or not?
`intNorm`	numeric	Normalization factor
`yrData`	data-frame	Parameters and search results per year
`nIntervals`	numeric	Number of intervals
`intvYears`	int array	Starting year of each interval
`intvData`	data-frame	Parameters and search results per interval

This list contains two data-frames. The first dataframe, `$yrData`, holds search parameters and search results as specified for and found for each year in the user-supplied range of years. The data frame consists of five arrays, which can be seen as a dataset, where each row consists of the information for one index in all five arrays:

`srchTerm`	chr array	Search term belonging to this index
`srchId`	int array	Search term number
`year`	int array	Year for this index
`words`	num array	Number of words available in Nederlab for this index's year
`hits`	num array	Number of times the search term of this index occurs in the year of this index

The second dataframe is `$intvData`, and it holds the search parameters and search results for each interval:

`srchTerm`	chr array	Search term belonging to this index
`srchId`	int array	Search term number
`intvNames`	chr array	Interval specification for this index
`intvYears`	num array	Interval start year for this index
`intvHits`	num array	Normalized number of times the search term of this index occurs in the interval of this index
`intvAbs`	num array	Absolute frequency for the search term of this index in the interval of this index
`intvWords`	num array	Number of words available in Nederlab for this index's interval

Examples

```
> nlab <- nlabcalc(c("hans", "piet"), NULL, NULL, 1800, 1895, user="e")
> nlab <- nlabintv(nlab, interval=5)
> nlab <- nlabmakeggplot(nlab, "linepoint")
> nlab <- nlabtosvg(nlab, user="e")
> str(nlab)
List of 28
 $ input     : chr [1:2] "hans" "piet"
 $ cnds      : NULL
 $ colors    : NULL
 $ cName     : chr [1:2(1d)] "hans" "piet"
 $ legend    : logi TRUE
 $ width     : num 7
 $ height    : num 5
 $ nYears    : int 120
 $ nSearch   : int 2
 $ source    : num 1800
 $ yrFirst   : num 1895
 $ yrLast    : num 2014
 $ yrIntv    : num 5
 $ years     : int [1:120] 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 ...
 $ lQuery    :List of 7
  ..$ query_1: chr
   "{\"filter\":null,\"condition\":{\"type\":\"and\",\"list\":[{\"type\":\"phrase\"
   ,\"field\":1800,\"value\":\"hans\"},{\"type\":\""| __truncated__
  ..$ query_2: chr
   "{\"filter\":null,\"condition\":{\"type\":\"and\",\"list\":[{\"type\":\"phrase\"
   ,\"field\":1800,\"value\":\"hans\"},{\"type\":\""| __truncated__
  ..$ query_3: chr
   "{\"filter\":null,\"condition\":{\"type\":\"and\",\"list\":[{\"type\":\"phrase\"
   ,\"field\":1800,\"value\":\"hans\"},{\"type\":\""| __truncated__
  ..$ query_4: chr
   "{\"filter\":null,\"condition\":{\"type\":\"and\",\"list\":[{\"type\":\"phrase\"
   ,\"field\":1800,\"value\":\"hans\"},{\"type\":\""| __truncated__
  ..$ query_5: chr
   "{\"filter\":null,\"condition\":{\"type\":\"and\",\"list\":[{\"type\":\"phrase\"
   ,\"field\":1800,\"value\":\"piet\"},{\"type\":\""| __truncated__
  ..$ query_6: chr
   "{\"filter\":null,\"condition\":{\"type\":\"and\",\"list\":[{\"type\":\"phrase\"
   ,\"field\":1800,\"value\":\"piet\"},{\"type\":\""| __truncated__
  ..$ query_7: chr
   "{\"filter\":null,\"condition\":{\"type\":\"and\",\"list\":[{\"type\":\"phrase\"
   ,\"field\":1800,\"value\":\"piet\"},{\"type\":\""| __truncated__
 $ figType   : chr "linepoint"
 $ fig       :List of 9
  ..$ data       :'data.frame':   48 obs. of  7 variables:
  .. ..$ srchTerm : chr [1:48] "hans" "hans" "hans" "hans" ...
  .. ..$ srchId   : int [1:48] 1 1 1 1 1 1 1 1 1 1 ...
  .. ..$ intvNames: chr [1:48] "1895-1899" "1900-1904" "1905-1909" "1910-1914" ...
  .. ..$ intvYears: num [1:48] 1895 1900 1905 1910 1915 ...
  .. ..$ intvHits : num [1:48] 0.0759 0.0709 0.0609 0.0658 0.0812 ...
  .. ..$ intvAbs  : num [1:48] 2173 2073 2475 2051 2156 ...
  .. ..$ intvWords: num [1:48] 28632896 29233831 40630814 31149653 26562382 ...
  ..$ layers     :List of 2
  .. ..$ :Classes 'proto', 'environment' <environment: 0x176dd524>
  .. ..$ :Classes 'proto', 'environment' <environment: 0x176d0218>
  ..$ scales     :Reference class 'Scales' [package "ggplot2"] with 1 fields
  .. ..$ scales:List of 1
  .. .. ..$ :List of 14
  .. .. .. ..$ call       : language discrete_scale(aesthetics = "colour",
   scale_name = "hue", palette = hue_pal(h,      c, l, h.start, direction), name =
   "Zoekterm", na.value = na.value)
  .. .. .. ..$ aesthetics: chr "colour"
  .. .. .. ..$ scale_name: chr "hue"
  .. .. .. ..$ palette   :function (n)
  .. .. .. ..$ range     :Reference class 'DiscreteRange' [package "scales"] with 1
   fields
  .. .. .. .. ..$ range: NULL
  .. .. .. .. ..and 15 methods, of which 3 are possibly relevant:
  .. .. .. .. ..  initialize, reset, train
  .. .. .. ..$ limits    : NULL
  .. .. .. ..$ na.value  : chr "grey50"
  .. .. .. ..$ expand    : list()
  .. .. .. .. ..- attr(*, "class")= chr "waiver"
  .. .. .. ..$ name      : chr "Zoekterm"
  .. .. .. ..$ breaks    : list()
  .. .. .. .. ..- attr(*, "class")= chr "waiver"
  .. .. .. ..$ labels    : list()
  .. .. .. .. ..- attr(*, "class")= chr "waiver"
  .. .. .. ..$ legend    : NULL
  .. .. .. ..$ drop      : logi TRUE
  .. .. .. ..$ guide     : chr "legend"
  .. .. .. ..- attr(*, "class")= chr [1:3] "hue" "discrete" "scale"
  .. ..and 21 methods, of which 9 are possibly relevant:
  .. ..  add, clone, find, get_scales, has_scale, initialize, input, n,
  .. ..  non_position_scales
  ..$ mapping    :List of 3
  .. ..$ colour: symbol srchTerm
  .. ..$ x     : symbol intvYears
  .. ..$ y     : symbol intvHits
  ..$ theme      : list()
  ..$ coordinates:List of 1
  .. ..$ limits:List of 2
  .. .. ..$ x: NULL
  .. .. ..$ y: NULL
  .. ..- attr(*, "class")= chr [1:2] "cartesian" "coord"
  ..$ facet      :List of 1
  .. ..$ shrink: logi TRUE
  .. ..- attr(*, "class")= chr [1:2] "null" "facet"
  ..$ plot_env   :<environment: 0x1795e0d0>
  ..$ labels     :List of 3
  .. ..$ y      : chr "Frequentie per  1000  woorden"
  .. ..$ x      : chr "Periodes"
  .. ..$ colour: chr "srchTerm"
  ..- attr(*, "class")= chr [1:2] "gg" "ggplot"
 $ parse      :Classes 'XMLInternalElementNode', 'XMLInternalNode',
   'XMLAbstractNode' <externalptr>
 $ norm       : logi TRUE
 $ intNorm    : num 1000
 $ events     :List of 5
  ..$ fig_mouseover : chr "fig_showTip"
  ..$ fig_mouseout  : chr "fig_hideTip"
  ..$ fig_click     : chr "fig_click"
  ..$ vert_mousemove: chr "vert_showVert"
  ..$ vert_mouseout : chr "vert_hideVert"
 $ yrData     :'data.frame':       240 obs. of  5 variables:
  ..$ srchTerm: chr [1:240] "hans" "hans" "hans" "hans" ...
  ..$ srchId  : int [1:240] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ year    : int [1:240] 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 ...
  ..$ words   : num [1:240] 4877068 5279616 6906461 6024377 5545374 ...
  ..$ hits    : num [1:240] 377 392 414 484 506 319 450 568 386 350 ...
 $ timeorder : chr "nederl_time_order"
 $ server    : chr "radboud"
 $ nIntervals: num 24
 $ intvYears : num [1:24(1d)] 1895 1900 1905 1910 1915 ...
 $ intvData  :'data.frame':       48 obs. of  7 variables:
  ..$ srchTerm : chr [1:48] "hans" "hans" "hans" "hans" ...
  ..$ srchId   : int [1:48] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ intvNames: chr [1:48] "1895-1899" "1900-1904" "1905-1909" "1910-1914" ...
  ..$ intvYears: num [1:48] 1895 1900 1905 1910 1915 ...
  ..$ intvHits : num [1:48] 0.0759 0.0709 0.0609 0.0658 0.0812 ...
  ..$ intvAbs  : num [1:48] 2173 2073 2475 2051 2156 ...
  ..$ intvWords: num [1:48] 28632896 29233831 40630814 31149653 26562382 ...
 $ method    : chr "termfreq"
```

9 Appendices

9.1 Installation of the service (older notes)

Assumptions:
1) A virtual host cloud machine has been prepared based on Linux RedHat/CentOS
2) There is an account “nederlab” for this machine

Installing apache (=httpd):
```
sudo yum install httpd
```

Installing tomcat:
```
sudo yum install tomcat
```

Letting tomcat work under apache:
Find directory `/etc/httpd/conf.d` and add a new file `ajp.conf`:
(NOTE: see further below for an improved version)
```
ProxyRequests Off
<Proxy *>
        Order deny,allow
        Deny from none
        Allow from localhost
</Proxy>
ProxyPass /tomcat ajp://localhost:8009/
ProxyPassReverse    /tomcat ajp://localhost:8009/
ProxyPass /manager ajp://localhost:8009/manager
ProxyPass /docs     ajp://localhost:8009/docs
ProxyPassReverse  /manager ajp://localhost:8009/manager
ProxyPassReverse  /docs    ajp://localhost:8009/docs
```
What this does: it defines one part of the interface between Apache (httpd) and Tomcat (the other part is defined in `/usr/share/tomcat/conf/server.xml`, in the line that defines an “AJP 1.3 connector on port 8009). This particular implementation says:
1) The connection is defined relative to “`localhost`”
2) Any access to http://myhost/tomcat is interpreted as a local access to the tomcat root (And this implies that any directories *under* the tomcat root can be accessed likewise: http://myhost/tomcat/nlabr accesses http://localhost/nlabr)
3) Access to http://myhost/manager and http://myhost/docs is interpreted as access to these respective tomcat pages. This means we can access the tomcat manager from the outside (provided we know the username – `tomcat -` and the password respectively, as defined in `tomcat-users.xml`.)

Per Feb/2015 there is an improved version, which increases the safety:
```
ProxyRequests Off
<Proxy *>
        Order deny,allow
        Deny from none
        Allow from localhost
</Proxy>
ProxyPass           /tomcat/nlabr ajp://localhost:8009/nlabr
ProxyPassReverse    /tomcat/nlabr ajp://localhost:8009/nlabr
```
This version does not allow access to http://server/tomcat, but only to http://server/tomcat/nlabr. This means that no logging-in is invited (unless a potential user attempts to connect through sftp or ssh, obviously).

Adapt the file `/usr/share/tomcat/conf/server.xml`, adding after the `<Host>` section for “`localhost`” another section:

```
      <Host name="localhost2"  appBase="/home/nederlab/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>
```

This section only serves to make sure that any `.war` file that is put in `/home/nederlab/webapps` gets automatically unpacked.

Add a context specification file `nlabr.xml` to the `localhost` in `/usr/share/tomcat/conf/Catalina/localhost`:

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="/home/nederlab/webapps/NLabR" path="/nlabr" reloadable="true" />
```

Result of this exercise:

- The `.war` files in `~/webapps` are automatically unpacked and deployed upon renewal
- Http accessing is served in this way:
  - Access to **Fout! De hyperlinkverwijzing is ongeldig.** – Apache (httpd) handling by looking in `/var/www/html`
  - Access to http://145.100.57.84/tomcat - Tomcat homepage handles this
  - Access to http://145.100.57.84/manager - Handled by tomcat manager
  - Nederlab visualisatie basis: http://145.100.57.84/tomcat/nlabr
  - Nederlab visualisatie commando: http://145.100.57.84/tomcat/nlabr/qxjob?{...}

Installing R:

```
sudo yum install R
```

Preparing basics for R libraries:

```
sudo yum install curl curl-devel
sudo yum install libxml2 libxml2-devel
```

Getting R started up correctly:

```
R
setwd("/home/nederlab")
source("NedLabVisGG.r")
package("rJava")
package("Rserve")
```

The file `.bash_profile` needs to be adapted to contain several variables…

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
# User specific environment and startup programs
JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.71.x86_64/jre
```

```
JRE_HOME=$JAVA_HOME
R_HOME=/usr/lib64/R
JRI_HOME=/home/nederlab/R/x86_64-redhat-linux-gnu-library/3.1/library/rJava
PATH=$JAVA_HOME/bin:$R_HOME/bin:$JRI_HOME:$PATH:$HOME/bin
CATALINA_HOME=/usr/share/tomcat
# Make the variables available
export JAVA_HOME
export JRE_HOME
export R_HOME
export PATH
R_SHARE_DIR=/usr/share/R
export R_SHARE_DIR
R_INCLUDE_DIR=/usr/include/R
export R_INCLUDE_DIR
R_DOC_DIR=/usr/share/doc/R-3.1.2
export R_DOC_DIR
JRI_LD_PATH=${R_HOME}/lib:${R_HOME}/bin:
if test -z "$LD_LIBRARY_PATH"; then
  LD_LIBRARY_PATH=$JRI_LD_PATH
else
  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JRI_LD_PATH
fi
```

Tomcat needs to start in such a way that it knows where to find the additional libraries. So adapt `/etc/tomcat/tomcat.conf`:

```
# Use JAVA_OPTS to set java.library.path for libtcnative.so
JAVA_OPTS="-Djava.library.path=/home/nederlab/R/x86_64-redhat-linux-gnu-
    library/3.1/library"

java.library.path = /home/nederlab/R/x86_64-redhat-linux-gnu-library/3.1
        /usr/lib64/R/lib
        /usr/lib64/R/bin
        /usr/java/packages/lib/amd64
        /usr/lib64
        /lib64
        /lib
        /usr/lib
```

This conf file also needs a last line with `R_HOME` defined as `/usr/lib64/R/bin`.

Directories to be created:

`/usr/share/tmp`   Must be readable and writable by *everyone*!!!

9.2 Updating (notes)

Starting and stopping of the Apache server:

```
sudo service httpd start
```

Starting and stopping tomcat:

```
sudo service tomcat start
sudo service tomcat restart
sudo service tomcat stop
```

If there is a pid error message, then remove the **tomcat PID lock** files:

```
sudo rm /var/run/tomcat.pid
sudo rm /var/lock/subsys/tomcat
sudo service tomcat start
```

If there is a jriReadConsole infinite loop, then replace the existing catalina.out with the **empty** one in the nederlab home directory:

```
sudo cp /home/nederlab/catalina.empty /usr/share/tomcat/logs/catalina.out
```

Any changes in the R-location and the location or name of the `NedLabVisGG.r` file should be processed in the file: `/home/nederlab/webapps/nlabr-server.json`

Also restart the Rserve program **after having killed any Rserve processes**:

```
ps -ef | grep Rserve
kill xxxx         (fill in the process number to be killed)
cd ~
sh startRserve.sh
```

This re-loads the current `~/NedLabVisGG.r`

9.3 Using the service (notes)

Getting debug info from the service (see section 2.5 and the response in section 3.6):

```
http://145.100.57.84/tomcat/nlabr/debug
```

Testing execution of an R-function through the service (see sections 2.7 and 3.8):

```
http://145.100.57.84/tomcat/nlabr/test?aapjes
```

An example of a real 'life' call to the service:

```
http://145.100.57.84/tomcat/nlabr/qxjob?{"srchTerms": ["koe", "paart"], "vis":
    "bar", "yrFrom": 1800, "yrTo": 1895, "interval": 5}
```

The output can be monitored on the server by looking at:
`sudo cat /usr/share/tomcat/logs/catalina.out` (using sudo)
`sudo tail -n 100 /usr/share/tomcat/logs/catalina.out` (using sudo)

Testing the service can be done locally:

```
curl
    http://localhost:8080/nlabr/qxjob?%7B%22srchTerms%22:%20%5B%22koe%22,%20%22paart
    %22%5D,%20%22vis%22:%20%22bar%22,%20%22yrFrom%22:%201800,%20%22yrTo%22:%201895,%
    20%22interval%22:%205%7D
```

The contents of the execution **cache** can be inspected locally by:

```
curl http://localhost:8080/nlabr/cache-info
```

The **status** of the currently committed job(s) can be checked locally by looking in the directory /usr/share/tmp for the most recent .log files.
The status can be checked externally by (1) getting the userid and jobid, and (2) requesting:

```
http://145.100.57.84/tomcat/nlabr/statusq?{"userid": "yy", "jobid": "xx"}
```

Where "yy" and "xx" need to be filled in with the information received within the "qxjob" answer!!!

10 References

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