Package 'RSelenium'

January 3, 2019

Type Package

Title R Bindings for 'Selenium WebDriver'

Version 1.7.5

Description Provides a set of R bindings for the 'Selenium 2.0 WebDriver'
(see <https://seleniumhq.github.io/docs/wd.html>
for more information) using the 'JsonWireProtocol' (see
<https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol> for more
information). 'Selenium 2.0 WebDriver' allows driving a web browser
natively as a user would either locally or on a remote machine using
the Selenium server it marks a leap forward in terms of web browser
automation. Selenium automates web browsers (commonly referred to as
browsers). Using RSelenium you can automate browsers locally or
remotely.

License AGPL-3

URL `http://ropensci.github.io/RSelenium`

BugReports `http://github.com/ropensci/RSelenium/issues`

Additional_repositories http://www.omegahat.net/R

Depends R (>= 3.0.0)

Imports XML, methods, caTools, tools, utils, openssl, httr, wdman(>=
0.2.2), binman

Suggests testthat, knitr, Rcompression, covr, rmarkdown

VignetteBuilder knitr

LazyData true

Collate 'RSelenium.R' 'errorHandler.R' 'remoteDriver.R' 'rsDriver.R'
'selKeys-data.R' 'util.R' 'webElement.R'

Encoding UTF-8

RoxygenNote 6.1.1

NeedsCompilation no

Author John Harrison [aut] (original author),
Ju Yeong Kim [cre] (rOpenSci maintainer)

Maintainer Ju Yeong Kim <jkim2345@fredhutch.org>

Repository CRAN

Date/Publication 2019-01-03 09:20:11 UTC

R topics documented:

`RSelenium-package` *An R client for Selenium Remote Webdriver*

Description

These are R bindings for the WebDriver API in Selenium 2. They use the JsonWireProtocol defined at https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol to communicate with a Selenium RemoteWebDriver Server.

Author(s)

John Harrison

References

http://seleniumhq.org/projects/webdriver/

`checkForServer` *Check for Server binary*

Description

Defunct. Please use `rsDriver`

Usage

```
checkForServer(dir = NULL, update = FALSE, rename = TRUE,
  beta = FALSE)
```

Arguments

`dir`	A directory in which the binary is to be placed.
`update`	A boolean indicating whether to update the binary if it is present.
`rename`	A boolean indicating whether to rename to "selenium-server-standalone.jar".
`beta`	A boolean indicating whether to include beta releases.

Details

`checkForServer` A utility function to check if the Selenium Server stanalone binary is present.

Detail

The downloads for the Selenium project can be found at http://selenium-release.storage.googleapis.com/index.html. This convience function downloads the standalone server and places it in the RSelenium package directory bin folder by default.

Examples

```
## Not run:
checkForServer()

## End(Not run)
```

`errorHandler-class` *CLASS errorHandler*

Description

class to handle errors

Details

This class is an internal class used by remoteDriver and webElement. It describes how drivers may respond. With a wide range of browsers etc the response can be variable.

Fields

`statusCodes` A list with status codes and their descriptions.

`status` A status code summarizing the result of the command. A non-zero value indicates that the command failed. A value of one is not a failure but may indicate a problem.

`statusclass` Class associated with the java library underlying the server. For Example: org.openqa.selenium.remote.Respo

`sessionid` An opaque handle used by the server to determine where to route session-specific commands. This ID should be included in all future session-commands in place of the :sessionId path segment variable.

`hcode` A list

`value` A list containing detailed information regarding possible errors:

- `message`**:** A descriptive message for the command failure.
- `screen`**:** string (Optional) If included, a screenshot of the current page as a base64 encoded string.
- `class`**:** string (Optional) If included, specifies the fully qualified class name for the exception that was thrown when the command failed.
- `stackTrace`**:** array (Optional) If included, specifies an array of JSON objects describing the stack trace for the exception that was thrown when the command failed. The zeroeth element of the array represents the top of the stack.

`responseheader` There are two levels of error handling specified by the wire protocol: invalid requests and failed commands. Invalid Requests will probably be indicted by a status of 1.

All invalid requests should result in the server returning a 4xx HTTP response. The response Content-Type should be set to text/plain and the message body should be a descriptive error message. The categories of invalid requests are as follows:

- `Unknown Commands`**:** If the server receives a command request whose path is not mapped to a resource in the REST service, it should respond with a 404 Not Found message.
- `Unimplemented Commands`**:** Every server implementing the WebDriver wire protocol must respond to every defined command. If an individual command has not been implemented on the server, the server should respond with a 501 Not Implemented error message. Note this is the only error in the Invalid Request category that does not return a 4xx status code.

`Variable Resource Not Found`**:** If a request path maps to a variable resource, but that resource does not exist, then the server should respond with a 404 Not Found. For example, if ID my-session is not a valid session ID on the server, and a command is sent to GET /session/my-session HTTP/1.1, then the server should gracefully return a 404.

`Invalid Command Method`**:** If a request path maps to a valid resource, but that resource does not respond to the request method, the server should respond with a 405 Method Not Allowed. The response must include an Allows header with a list of the allowed methods for the requested resource.

`Missing Command Parameters`**:** If a POST/PUT command maps to a resource that expects a set of JSON parameters, and the response body does not include one of those parameters, the server should respond with a 400 Bad Request. The response body should list the missing parameters.

`debugheader` Not currently implemented

Methods

`checkStatus(resContent)` An internal method to check the status returned by the server. If status indicates an error an appropriate error message is thrown.

`errorDetails(type = "value")` Return error details. Type can one of c("value", "class", "status")

`obscureUrlPassword(url)` Replaces the username and password of url with ****

`queryRD(ipAddr, method = "GET", qdata = NULL)` A method to communicate with the remote server implementing the JSON wire protocol.

`getChromeProfile` *Get Chrome profile.*

Description

`getChromeProfile` A utility function to get a Chrome profile.

Usage

```
getChromeProfile(dataDir, profileDir)
```

Arguments

`dataDir`	Specifies the user data directory, which is where the browser will look for all of its state.
`profileDir`	Selects directory of profile to associate with the first browser launched.

Detail

A chrome profile directory is passed as an extraCapability. The data dir has a number of default locations

Windows XP Google Chrome: C:/Documents and Settings/%USERNAME%/Local Settings/Application Data/Google/Chrome/User Data

Windows 8 or 7 or Vista Google Chrome: C:/Users/%USERNAME%/AppData/Local/Google/Chrome/User Data

Mac OS X Google Chrome: ~/Library/Application Support/Google/Chrome

Linux Google Chrome: ~/.config/google-chrome

The profile directory is contained in the user directory and by default is named "Default"

Examples

```
## Not run:
# example from windows using a profile directory "Profile 1"
cprof <- getChromeProfile(
  "C:\\Users\\john\\AppData\\Local\\Google\\Chrome\\User Data",
  "Profile 1"
)
remDr <- remoteDriver(browserName = "chrome", extraCapabilities = cprof)

## End(Not run)
```

`getFirefoxProfile` *Get Firefox profile.*

Description

`getFirefoxProfile` A utility function to get a firefox profile.

Usage

```
getFirefoxProfile(profDir, useBase = FALSE)
```

Arguments

`profDir`	The directory in which the firefox profile resides
`useBase`	Logical indicating whether to attempt to use zip from utils package. Maybe easier for Windows users.

Detail

A firefox profile directory is zipped and base64 encoded. It can then be passed to the selenium server as a required capability with key firefox_profile

Examples

```
## Not run:
fprof <- getFirefoxProfile("~/.mozilla/firefox/9qlj1ofd.testprofile")
remDr <- remoteDriver(extraCapabilities = fprof)
remDr$open()

## End(Not run)
```

`makeFirefoxProfile` *Make Firefox profile.*

Description

`makeFirefoxProfile` A utility function to make a firefox profile.

Usage

```
makeFirefoxProfile(opts)
```

Arguments

`opts`	option list of firefox

Detail

A firefox profile directory is zipped and base64 encoded. It can then be passed to the selenium server as a required capability with key firefox_profile

Note

Windows doesn't come with command-line zip capability. Installing rtools `https://CRAN.R-project.org/bin/windows/Rtools/index.html` is a straightforward way to gain this capability.

Examples

```
## Not run:
fprof <- makeFirefoxProfile(list(browser.download.dir = "D:/temp"))
remDr <- remoteDriver(extraCapabilities = fprof)
remDr$open()

## End(Not run)
```

phantom — *Start a phantomjs binary in webdriver mode.*

Description

Defunct. Please use `rsDriver` or `phantomjs`

Usage

```
phantom(pjs_cmd = "", port = 4444L, extras = "", ...)
```

Arguments

`pjs_cmd`	The name, full or partial path of a phantomjs executable. This is optional only state if the executable is not in your path.
`port`	An integer giving the port on which phantomjs will listen. Defaults to 4444. format [[<IP>:]<PORT>]
`extras`	An optional character vector: see 'Details'.
`...`	Arguments to pass to `system2`

Details

`phantom` A utility function to control a phantomjs binary in webdriver mode.

Detail

phantom() is used to start a phantomjs binary in webdriver mode. This can be used to drive a phantomjs binary on a machine without selenium server. Argument extras can be used to specify optional extra command line arguments see `http://phantomjs.org/api/command-line.html`

Value

phantom() returns a list with two functions:

getPID returns the process id of the phantomjs binary running in webdriver mode.

stop terminates the phantomjs binary running in webdriver mode using `pskill`

Examples

```
## Not run:
pJS <- phantom()
# note we are running here without a selenium server phantomjs is
# listening on port 4444
# in webdriver mode
remDr <- remoteDriver(browserName = "phantomjs")
remDr$open()
remDr$navigate("http://www.google.com/ncr")
remDr$screenshot(display = TRUE)
```

```
webElem <- remDr$findElement("name", "q")
webElem$sendKeysToElement(list("HELLO WORLD"))
remDr$screenshot(display = TRUE)
remDr$close()
# note remDr$closeServer() is not called here. We stop the phantomjs
# binary using
pJS$stop()

## End(Not run)
```

`remoteDriver-class` *CLASS remoteDriver*

Description

remoteDriver Class uses the JsonWireProtocol to communicate with the Selenium Server. If an error occurs while executing the command then the server sends back an HTTP error code with a JSON encoded reponse that indicates the precise Response Error Code. The remoteDriver class inherits from the `errorHandler` class. If no error occurred, then the subroutine called will return the value sent back from the server (if a return value was sent). So a rule of thumb while invoking methods on the driver is if the method did not return a status greater then zero when called, then you can safely assume the command was successful even if nothing was returned by the method.

Details

remoteDriver is a generator object. To define a new remoteDriver class method ‘new‘ is called. The slots (default value) that are user defined are: remoteServerAddr(localhost), port(4444), browserName(firefox), version(""), platform(ANY), javascript(TRUE). See examples for more information on use.

Fields

`remoteServerAddr` Object of class `"character"`, giving the ip of the remote server. Defaults to localhost

`port` Object of class `"numeric"`, the port of the remote server on which to connect

`browserName` Object of class `"character"`. The name of the browser being used; should be one of chrome|firefox|htmlunit| internet explorer|iphone.

`path` base URL path prefix for commands on the remote server. Defaults to "/wd/hub"

`version` Object of class `"character"`. The browser version, or the empty string if unknown.

`platform` Object of class `"character"`. A key specifying which platform the browser is running on. This value should be one of WINDOWS|XP|VISTA|MAC|LINUX|UNIX. When requesting a new session, the client may specify ANY to indicate any available platform may be used.

`javascript` Object of class `"logical"`. Whether the session supports executing user supplied JavaScript in the context of the current page.

`nativeEvents` Object of class `"logical"`. Whether the session supports native events. n WebDriver advanced user interactions are provided by either simulating the Javascript events directly (i.e. synthetic events) or by letting the browser generate the Javascript events (i.e. native events). Native events simulate the user interactions better.

`serverURL` Object of class `"character"`. Url of the remote server which JSON requests are sent to.

`sessionInfo` Object of class `"list"`. A list containing information on sessions.

Methods

`acceptAlert()` Accepts the currently displayed alert dialog. Usually, this is equivalent to clicking the 'OK' button in the dialog.

`addCookie(name, value, path = "/", domain = NULL, httpOnly = NULL, expiry = NULL, secure = FALSE)` Set a cookie on the domain. The inputs are required apart from those with default values.

`buttondown(buttonId = 0)` Click and hold the given mouse button (at the coordinates set by the last moveto command). Note that the next mouse-related command that should follow is buttondown . Any other mouse command (such as click or another call to buttondown) will yield undefined behaviour. buttonId - any one of 'LEFT'/0 'MIDDLE'/1 'RIGHT'/2. Defaults to 'LEFT'

`buttonup(buttonId = 0)` Releases the mouse button previously held (where the mouse is currently at). Must be called once for every buttondown command issued. See the note in click and buttondown about implications of out-of-order commands. buttonId - any one of 'LEFT'/0 'MIDDLE'/1 'RIGHT'/2. Defaults to 'LEFT'

`click(buttonId = 0)` Click any mouse button (at the coordinates set by the last mouseMoveToLocation() command). buttonId - any one of 'LEFT'/0 'MIDDLE'/1 'RIGHT'/2. Defaults to 'LEFT'

`close()` Close the current session.

`closeServer()` Closes the server in practice terminating the process. This is useful for linux systems. On windows the java binary operates as a seperate shell which the user can terminate.

`closeWindow()` Close the current window.

`deleteAllCookies()` Delete all cookies visible to the current page.

`deleteCookieNamed(name)` Delete the cookie with the given name. This command will be a no-op if there is no such cookie visible to the current page.

`dismissAlert()` Dismisses the currently displayed alert dialog. For comfirm() and prompt() dialogs, this is equivalent to clicking the 'Cancel' button. For alert() dialogs, this is equivalent to clicking the 'OK' button.

`doubleclick(buttonId = 0)` Double-Click any mouse button (at the coordinates set by the last mouseMoveToLocation() command). buttonId - any one of 'LEFT'/0 'MIDDLE'/1 'RIGHT'/2. Defaults to 'LEFT'

`executeAsyncScript(script, args = list())` Inject a snippet of JavaScript into the page for execution in the context of the currently selected frame. The executed script is assumed to be asynchronous and must signal that is done by invoking the provided callback, which is always provided as the final argument to the function. The value to this callback will be returned to the client. Asynchronous script commands may not span page loads. If an unload event is fired while waiting for a script result, an error should be returned to the client.

`executeScript(script, args = list(""))` Inject a snippet of JavaScript into the page for execution in the context of the currently selected frame. The executed script is assumed to be synchronous and the result of evaluating the script is returned to the client. The script argument defines the script to execute in the form of a function body. The value returned by that function will be returned to the client. The function will be invoked with the provided args array and the values may be accessed via the arguments object in the order specified. Arguments may be any JSON-primitive, array, or JSON object. JSON objects that define a WebElement reference will be converted to the corresponding DOM element. Likewise, any WebElements in the script result will be returned to the client as WebElement JSON objects.

`findElement(using = c("xpath", "css selector", "id", "name", "tag name", "class name", "link text",`
Search for an element on the page, starting from the document root. The located element will be returned as an object of webElement class.The inputs are:

- `using`: Locator scheme to use to search the element, available schemes: Defaults to 'xpath'. Partial string matching is accepted.
  - **"class name" :** Returns an element whose class name contains the search value; compound class names are not permitted.
  - **"css selector" :** Returns an element matching a CSS selector.
  - **"id" :** Returns an element whose ID attribute matches the search value.
  - **"name" :** Returns an element whose NAME attribute matches the search value.
  - **"link text" :** Returns an anchor element whose visible text matches the search value.
  - **"partial link text" :** Returns an anchor element whose visible text partially matches the search value.
  - **"tag name" :** Returns an element whose tag name matches the search value.
  - **"xpath" :** Returns an element matching an XPath expression.
- `value`: The search target. See examples.

`findElements(using = c("xpath", "css selector", "id", "name", "tag name", "class name", "link text"`
Search for multiple elements on the page, starting from the document root. The located elements will be returned as an list of objects of class WebElement. The inputs are:

- `using`: Locator scheme to use to search the element, available schemes: "class name", "css selector", "id", "name", "link text", "partial link text", "tag name", "xpath" . Defaults to 'xpath'. Partial string matching is accepted. See the findElement method for details
- `value`: The search target. See examples.

`getActiveElement()` Get the element on the page that currently has focus. The located element will be returned as a WebElement id.

`getAlertText()` Gets the text of the currently displayed JavaScript alert(), confirm() or prompt() dialog.

`getAllCookies()` Retrieve all cookies visible to the current page. Each cookie will be returned as a list with the following name and value types:

- `name`: character
- `value`: character
- `path`: character
- `domain`: character
- `secure`: logical

`getCurrentUrl()` Retrieve the url of the current page.

`getCurrentWindowHandle()` Retrieve the current window handle.

`getLogTypes()` Get available log types. Common log types include 'client' = Logs from the client, 'driver' = Logs from the webdriver, 'browser' = Logs from the browser, 'server' = Logs from the server. Other log types, for instance, for performance logging may also be available. phantomjs for example returns a har log type which is a single-entry log, with the HAR (HTTP Archive) of the current webpage, since the first load (it's cleared at every unload event)

`getPageSource(...)` Get the current page source.

`getSessions()` Returns a list of the currently active sessions. Each session will be returned as a list containing amongst other items:

- `id`: The session ID
- `capabilities`: An object describing session's capabilities

`getStatus()` Query the server's current status. All server implementations should return two basic objects describing the server's current platform and when the server was built.

`getTitle(url)` Get the current page title.

`getWindowHandles()` Retrieve the list of window handles used in the session.

`getWindowPosition(windowId = "current")` Retrieve the window position. 'windowid' is optional (default is 'current' window). Can pass an appropriate 'handle'

`getWindowSize(windowId = "current")` Retrieve the window size. 'windowid' is optional (default is 'current' window). Can pass an appropriate 'handle'

`goBack()` Equivalent to hitting the back button on the browser.

`goForward()` Equivalent to hitting the forward button on the browser.

`log(type)` Get the log for a given log type. Log buffer is reset after each request.

- `type`: The log type. Typically 'client', 'driver', 'browser', 'server'

`maxWindowSize(winHand = "current")` Set the size of the browser window to maximum. The windows handle is optional. If not specified the current window in focus is used.

`mouseMoveToLocation(x = NA_integer_, y = NA_integer_, webElement = NULL)` Move the mouse by an offset of the specificed element. If no element is specified, the move is relative to the current mouse cursor. If an element is provided but no offset, the mouse will be moved to the center of the element. If the element is not visible, it will be scrolled into view.

`navigate(url)` Navigate to a given url.

`open(silent = FALSE)` Send a request to the remote server to instantiate the browser.

`phantomExecute(script, args = list())` This API allows you to send a string of JavaScript via 'script', written for PhantomJS, and be interpreted within the context of a WebDriver Page. In other words, for the given script then this variable is initialized to be the current Page. See https://github.com/ariya/phantomjs/wiki/API-Reference-WebPage and the example in this help file. NOTE: Calling the PhantomJS API currently only works when PhantomJS is driven directly via phantom

`quit()` Delete the session & close open browsers.

`refresh()` Reload the current page.

`screenshot(display = FALSE, useViewer = TRUE, file = NULL)` Take a screenshot of the current page. The screenshot is returned as a base64 encoded PNG. If display is TRUE the screenshot is displayed locally. If useViewer is TRUE and RStudio is in use the screenshot is displayed in the RStudio viewer panel. If file is not NULL and display = FALSE the screenshot is written to the file denoted by file.

`sendKeysToActiveElement(sendKeys)` Send a sequence of key strokes to the active element. This command is similar to the send keys command in every aspect except the implicit termination: The modifiers are not released at the end of the call. Rather, the state of the modifier keys is kept between calls, so mouse interactions can be performed while modifier keys are depressed. The key strokes are sent as a list. Plain text is enter as an unnamed element of the list. Keyboard entries are defined in ‘selKeys‘ and should be listed with name ‘key‘. See the examples.

`sendKeysToAlert(sendKeys)` Sends keystrokes to a JavaScript prompt() or alert() dialog. The key strokes are sent as a list. Plain text is enter as an unnamed element of the list. Keyboard entries are defined in ‘selKeys‘ and should be listed with name ‘key‘. See the examples.

`setAsyncScriptTimeout(milliseconds = 10000)` Set the amount of time, in milliseconds, that asynchronous scripts executed by execute_async_script() are permitted to run before they are aborted and a |Timeout| error is returned to the client.

`setImplicitWaitTimeout(milliseconds = 10000)` Set the amount of time the driver should wait when searching for elements. When searching for a single element, the driver will poll the page until an element is found or the timeout expires, whichever occurs first. When searching for multiple elements, the driver should poll the page until at least one element is found or the timeout expires, at which point it will return an empty list. If this method is never called, the driver will default to an implicit wait of 0ms.

`setTimeout(type = "page load", milliseconds = 10000)` Configure the amount of time that a particular type of operation can execute for before they are aborted and a |Timeout| error is returned to the client.

`type`**:** The type of operation to set the timeout for. Valid values are: "script" for script timeouts, "implicit" for modifying the implicit wait timeout and "page load" for setting a page load timeout. Defaults to "page load"

`milliseconds`**:** The amount of time, in milliseconds, that time-limited commands are permitted to run. Defaults to 10000 milliseconds.

`setWindowPosition(x, y, winHand = "current")` Set the position (on screen) where you want your browser to be displayed. The windows handle is optional. If not specified the current window in focus is used.

`setWindowSize(width, height, winHand = "current")` Set the size of the browser window. The windows handle is optional. If not specified the current window in focus is used.

`switchToFrame(Id)` Change focus to another frame on the page. Id can be string|number|null|WebElement Object. If the Id is null, the server should switch to the page’s default content.

`switchToWindow(windowId)` Change focus to another window. The window to change focus to may be specified by its server assigned window handle, or by the value of its name attribute.

Examples

```
## Not run:
# start the server if one isnt running
startServer()

# use default server initialisation values
remDr <- remoteDriver$new()
```

```
# send request to server to initialise session
remDr$open()

# navigate to R home page
remDr$navigate("http://www.r-project.org")

# navigate to www.bbc.co.uk notice the need for http://
remDr$navigate("http://www.bbc.co.uk")

# go backwards and forwards
remDr$goBack()

remDr$goForward()

remDr$goBack()

# Examine the page source
frontPage <- remDr$getPageSource()

# The R homepage contains frames
webElem <- remDr$findElements(value = "//frame")
sapply(webElem, function(x) {x$getElementAttribute('name')})

# The homepage contains 3 frames: logo, contents and banner
# switch to the `contents` frame
webElem <- remDr$findElement(using = 'name', value = 'contents')
remDr$switchToFrame(webElem$elementId)

# re-examine the page source

contentPage <- remDr$getPageSource()
identical(contentPage, frontPage) # false we hope!!

# Find the link for the search page on R homepage. Use xpath as default.
webElem <- remDr$findElement(value = '//a[@href = "search.html"]')
webElem$getElementAttribute('href')
# http://www.r-project.org/search.html

# click the search link
webElem$clickElement()

# FILL OUT A GOOGLE SEARCH FORM
remDr$navigate("http://www.google.com")

# show different methods of accessing DOM components

webElem1 <- remDr$findElement(using = 'name', value = 'q')
webElem2 <- remDr$findElement(
  using = 'id',
  value = webElem1$getElementAttribute('id')[[1]])
webElem3 <- remDr$findElement(using = 'xpath',
                              value = '//input[@name = "q"]')
```

```
# Enter some text in the search box

webElem1$sendKeysToElement(list('RSelenium was here'))

# clear the text previously entered

webElem1$clearElement()

# show an example of sending a key press
webElem1$sendKeysToElement(list('R', key = 'enter'))

# Collate the results for the `R` search
googLinkText <- remDr$findElements(value = "//h3[@class = 'r']")
linkHeading <- sapply(googLinkText, function(x) x$getElementText())
googLinkDesc <- remDr$findElements(value = "//div[@class = 's']")
linkDescription <- sapply(googLinkDesc, function(x) x$getElementText())
googLinkHref <- remDr$findElements(value = "//h3[@class = 'r']/a")
linkHref <- sapply(googLinkHref,
                   function(x) x$getElementAttribute('href'))

data.frame(heading = linkHeading,
           description = linkDescription, href = linkHref)

# Example of javascript call
remDr$executeScript("return arguments[0] + arguments[1];", args = 1:2)
# Example of javascript async call
jsscript <-
  "arguments[arguments.length - 1](arguments[0] + arguments[1]);"
remDr$executeAsyncScript(jsscript, args = 1:2)

# EXAMPLE INJECTING INTO PHANTOMJS using phantomExecute
require(RSelenium)
pJS <- wdman::phantomjs(port = 4932L)
remDr <- remoteDriver(browserName = "phantomjs", port = 4932L)
remDr$open(silent = TRUE)
remDr$navigate("http://ariya.github.com/js/random/")
# returns a set of random numbers
remDr$findElement("id", "numbers")$getElementText()[[1]]
result = remDr$phantomExecute("var page = this;
                               page.onInitialized = function () {
                               page.evaluate(function () {
                               Math.random = function() {return 42/100}
                               })
                               }", list());
remDr$navigate("http://ariya.github.com/js/random/")
# Math.random returns our custom function
remDr$findElement("id", "numbers")$getElementText()[[1]]
remDr$close()
pJS$stop()

## End(Not run)
```

rsDriver — *Start a selenium server and browser*

Description

Start a selenium server and browser

Usage

```
rsDriver(port = 4567L, browser = c("chrome", "firefox", "phantomjs",
  "internet explorer"), version = "latest", chromever = "latest",
  geckover = "latest", iedrver = NULL, phantomver = "2.1.1",
  verbose = TRUE, check = TRUE, ...)
```

Arguments

`port`	Port to run on
`browser`	Which browser to start
`version`	what version of Selenium Server to run. Default = "latest" which runs the most recent version. To see other version currently sourced run binman::list_versions("seleniumserver")
`chromever`	what version of Chrome driver to run. Default = "latest" which runs the most recent version. To see other version currently sourced run binman::list_versions("chromedriver"), A value of NULL excludes adding the chrome browser to Selenium Server.
`geckover`	what version of Gecko driver to run. Default = "latest" which runs the most recent version. To see other version currently sourced run binman::list_versions("geckodriver"), A value of NULL excludes adding the firefox browser to Selenium Server.
`iedrver`	what version of IEDriverServer to run. Default = "latest" which runs the most recent version. To see other version currently sourced run binman::list_versions("iedriverserver"), A value of NULL excludes adding the internet explorer browser to Selenium Server. NOTE this functionality is Windows OS only.
`phantomver`	what version of PhantomJS to run. Default = "2.1.1" which runs the most recent stable version. To see other version currently sourced run binman::list_versions("phantomjs"), A value of NULL excludes adding the PhantomJS headless browser to Selenium Server.
`verbose`	If TRUE, include status messages (if any)
`check`	If TRUE check the versions of selenium available and the versions of associated drivers (chromever, geckover, phantomver, iedrver). If new versions are available they will be downloaded.
`...`	Additional arguments to pass to `remoteDriver`

Details

This function is a wrapper around `selenium`. It provides a "shim" for the current issue running firefox on Windows. For a more detailed set of functions for running binaries relating to the Selenium/webdriver project see the `wdman` package. Both the client and server are closed using a registered finalizer.

Value

A list containing a server and a client. The server is the object returned by `selenium` and the client is an object of class `remoteDriver`

Examples

```
## Not run:
# start a chrome browser
rD <- rsDriver()
remDr <- rD[["client"]]
remDr$navigate("http://www.google.com/ncr")
remDr$navigate("http://www.bbc.com")
remDr$close()
# stop the selenium server
rD[["server"]]$stop()

# if user forgets to stop server it will be garbage collected.
rD <- rsDriver()
rm(rD)
gc(rD)

## End(Not run)
```

`selKeys` *Selenium key mappings*

Description

This data set contains a list of selenium key mappings. selKeys is used when a sendKeys variable is needed. sendKeys is defined as a list. If an entry is needed from selKeys it is denoted by key.

Usage

```
selKeys
```

Format

A named list. The names are the descriptions of the keys. The values are the "UTF-8" character representations.

Source

https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol#sessionsessionidelementidvalue

startServer — *Start the standalone server.*

Description

Defunct. Please use rsDriver

Usage

```
startServer(dir = NULL, args = NULL, javaargs = NULL, log = TRUE,
  ...)
```

Arguments

dir	A directory in which the binary is to be placed.
args	Additional arguments to be passed to Selenium Server.
javaargs	arguments passed to JVM as opposed to the Selenium Server jar.
log	Logical value indicating whether to write a log file to the directory containing the Selenium Server binary.
...	arguments passed system2. Unix defaults wait = FALSE, stdout = FALSE, stderr = FALSE. Windows defaults wait = FALSE, invisible = TRUE.

Details

`startServer` A utility function to start the standalone server. Return two functions see values.

Value

Returns a list containing two functions. The 'getpid' function returns the process id of the started Selenium binary. The 'stop' function stops the started Selenium server using the process id.

Detail

By default the binary is assumed to be in the RSelenium package /bin directory. The log argument is for convience. Setting it to FALSE and stipulating args = c("-log /user/etc/somePath/somefile.log") allows a custom location. Using log = TRUE sets the location to a file named sellog.txt in the directory containing the Selenium Server binary.

Examples

```
## Not run:
selServ <- startServer()
# example of commandline passing
selServ <- startServer(
  args = c("-port 4455"),
  log = FALSE,
  invisible = FALSE
)
remDr <- remoteDriver(browserName = "chrome", port = 4455)
remDr$open()
# get the process id of the selenium binary
selServ$getpid()
# stop the selenium binary
selServ$stop()

## End(Not run)
```

webElement-class *CLASS webElement*

Description

Selenium Webdriver represents all the HTML elements as WebElements. This class provides a mechanism to represent them as objects & perform various actions on the related elements. Typically, the findElement method in `remoteDriver` returns an object of class webElement.

Details

webElement is a generator object. To define a new webElement class method ‘new‘ is called. When a webElement class is created an elementId should be given. Each webElement inherits from a remoteDriver. webElement is not usually called by the end-user.

Fields

`elementId` Object of class `"character"`, giving a character representation of the element id.

Methods

`clearElement()` Clear a TEXTAREA or text INPUT element’s value.

`clickElement()` Click the element.

`compareElements(otherElem)` Test if the current webElement and an other web element refer to the same DOM element.

`describeElement()` Describe the identified element.

`findChildElement(using = c("xpath", "css selector", "id", "name", "tag name", "class name", "link t`
Search for an element on the page, starting from the node defined by the parent webElement. The located element will be returned as an object of webElement class. The inputs are:

- `using`**:** Locator scheme to use to search the element, available schemes: "class name", "css selector", "id", "name", "link text", "partial link text", "tag name", "xpath" . Defaults to 'xpath'. Partial string matching is accepted.
- `value`**:** The search target. See examples.

`findChildElements(using = c("xpath", "css selector", "id", "name", "tag name", "class name", "link`
Search for multiple elements on the page, starting from the node defined by the parent webElement. The located elements will be returned as an list of objects of class WebElement. The inputs are:

- `using`: Locator scheme to use to search the element, available schemes: "class name", "css selector", "id", "name", "link text", "partial link text", "tag name", "xpath" . Defaults to 'xpath'. Partial string matching is accepted.
- `value`: The search target. See examples.

`getElementAttribute(attrName)` Get the value of an element's attribute. See examples.

`getElementLocation()` Determine an element's location on the page. The point (0, 0) refers to the upper-left corner of the page.

`getElementLocationInView()` Determine an element's location on the screen once it has been scrolled into view. Note: This is considered an internal command and should only be used to determine an element's location for correctly generating native events.

`getElementSize()` Determine an element's size in pixels. The size will be returned with width and height properties.

`getElementTagName()` Query for an element's tag name.

`getElementText()` Get the innerText of the element.

`getElementValueOfCssProperty(propName)` Query the value of an element's computed CSS property. The CSS property to query should be specified using the CSS property name, not the JavaScript property name (e.g. background-color instead of backgroundColor).

`highlightElement(wait = 75/1000)` Utility function to highlight current Element. Wait denotes the time in seconds between style changes on element.

`isElementDisplayed()` Determine if an element is currently displayed.

`isElementEnabled()` Determine if an element is currently enabled. Obviously to enable an element just preform a click on it.

`isElementSelected()` Determine if an OPTION element, or an INPUT element of type checkbox or radiobutton is currently selected.

`selectTag()` Utility function to return options from a select DOM node. The option nodes are returned as webElements. The option text and the value of the option attribute 'value' and whether the option is selected are returned also. If this method is called on a webElement that is not a select DOM node an error will result.

`sendKeysToElement(sendKeys)` Send a sequence of key strokes to an element. The key strokes are sent as a list. Plain text is enter as an unnamed element of the list. Keyboard entries are defined in ‘selKeys‘ and should be listed with name ‘key‘. See the examples.

`setElementAttribute(attributeName, value)` Utility function to set an elements atrributes.

`submitElement()` Submit a FORM element. The submit command may also be applied to any element that is a descendant of a FORM element.

Index