

Package: QuickJSR (via r-universe)

October 11, 2026

Title Interface for the 'QuickJS-NG' Lightweight 'JavaScript' Engine

Version 1.11.9000

Description An 'R' interface to the 'QuickJS' portable 'JavaScript' engine. The engine and all 'R' to 'JavaScript' interoperability is bundled within the package, requiring no dependencies beyond a 'C' compiler.

License MIT + file LICENSE

URL <https://github.com/andrjohns/QuickJSR>,
<https://github.com/quickjs-ng/quickjs>,
<https://andrjohns.github.io/QuickJSR/>

BugReports <https://github.com/andrjohns/QuickJSR/issues>

Suggests knitr, rmarkdown, tinytest (>= 1.4.1)

Encoding UTF-8

Language en-AU

NeedsCompilation yes

Roxygen list(markdown = TRUE)

SystemRequirements GNU make

VignetteBuilder knitr

Config/build/compilation-database true

Config/roxygen2/version 8.1.0

Config/pak/sysreqs make

Repository <https://andrjohns.r-universe.dev>

Date/Publication 2026-10-11 10:21:16 UTC

RemoteUrl <https://github.com/andrjohns/QuickJSR>

RemoteRef HEAD

RemoteSha 7f020071a5c6b685462d93593506c6ff669b49b5

Contents

QuickJSR-package	2
from_json	3
js_columnar_data_frame	3
js_masked_typed_array	4
js_mutable_view	5
js_readonly_view	6
js_ref_to_altprep	7
js_typed_array	8
JSCompiledScript	8
JSContext	9
JSContext-method-assign	9
JSContext-method-call	10
JSContext-method-eval	11
JSContext-method-get	11
JSContext-method-source	12
JSContext-method-validate	12
JSContext-new	13
JSContext-reference-methods	14
JSValueRef	14
qjs_eval	15
quickjs_flags	16
quickjs_version	16
to_json	17
Index	18

QuickJSR-package	<i>The QuickJSR package.</i>
------------------	------------------------------

Description

An interface to the QuickJS lightweight Javascript engine

Author(s)

Maintainer: Andrew R. Johnson <andrew.johnson@arjohnsonau.com> ([ORCID](#))

Authors:

- Andrew R. Johnson <andrew.johnson@arjohnsonau.com> ([ORCID](#))

Other contributors:

- QuickJS Authors (QuickJS sources and headers) [copyright holder]
- QuickJS-NG Authors (QuickJS-NG sources and headers) [copyright holder]

See Also

Useful links:

- <https://github.com/andrjohns/QuickJSR>
- <https://github.com/quickjs-ng/quickjs>
- Report bugs at <https://github.com/andrjohns/QuickJSR/issues>

from_json

from_json

Description

Use the QuickJS C API to convert a JSON string to an R object

Usage

```
from_json(json)
```

Arguments

json JSON string to convert to an R object

Value

R object

js_columnar_data_frame

Transfer a data frame as an object of columns

Description

Transfers a data frame as one JavaScript object whose properties are columns. This avoids creating one object per row. With `typed = TRUE`, raw, integer, and double columns use contiguous typed arrays. Integer columns with missing values, factors, and date-time columns use ordinary arrays. Double NA and NaN values become JavaScript NaN in typed columns. Explicit character row names are included as the `_row` column.

Usage

```
js_columnar_data_frame(x, typed = TRUE)
```

Arguments

x A data frame.

typed Use typed arrays for eligible raw, integer, and double columns.

Value

The data frame marked for column-oriented conversion.

Examples

```
ctx <- JSContext$new()
ctx$source(code = "function rows(x) { return x.value.length; }")
df <- data.frame(value = 1:4, label = letters[1:4])
ctx$call("rows", js_columnar_data_frame(df))
```

js_masked_typed_array *Transfer missing typed values with a validity mask*

Description

Transfers an R vector as a branded JavaScript object containing contiguous values and validity typed arrays. The validity byte is zero for a missing value and one otherwise.

Usage

```
js_masked_typed_array(x)
```

Arguments

x A logical, integer, or double vector without factor or date classes.

Details

Logical values use a Uint8Array, integers use an Int32Array, and doubles use a Float64Array. Every representation uses a Uint8Array validity mask and exposes the original length as length.

R integer and logical NA values have validity zero and a placeholder value of zero. Double NA is treated the same way, while ordinary NaN remains a valid Float64 value. This preserves the R distinction between NA_real_ and NaN.

JavaScript can modify the contents of both typed arrays. Returning the branded object to R reconstructs the original R vector type using the current values and validity bytes. A nonzero validity byte is treated as valid. Removing the brand by constructing a plain object returns the two buffers separately under the ordinary object conversion rules.

The mask uses one byte per element. Use js_typed_array() when a mask is not required. Factors and dates are rejected because their class semantics cannot be represented by the numeric values and validity buffers alone.

Value

A lightweight wrapper around the input vector.

Examples

```
ctx <- JSContext$new(profile = "bare")
ctx$source(code = paste(
  "function fillMissing(x) {",
  "  x.values[1] = 20;",
  "  x.validity[1] = 1;",
  "  return x;",
  "}")
))
values <- js_masked_typed_array(c(10L, NA_integer_, 30L))
ctx$call("fillMissing", values)
```

js_mutable_view

*Expose owned R vector storage as a mutable JavaScript view***Description**

Creates a fixed-length array-like JavaScript object whose indexed writes update R-owned vector storage. The wrapper makes one copy when it is created; passing it to JavaScript does not copy or convert the complete vector.

Usage

```
js_mutable_view(x)
```

Arguments

x An unclassed raw, logical, integer, double, or character vector.

Details

JavaScript reads convert elements on demand and writes update `view$value`. The caller's original vector is never modified. Multiple JavaScript views of the same wrapper share its owned storage.

The view has `length`, indexed access, enumeration, iteration, inherited `Array` methods, and `toArray()`. Its length cannot change, properties cannot be added, and elements cannot be deleted. Methods such as `reverse`, `sort`, `fill`, and `copyWithin` work because they only replace existing elements.

Assignments are type checked. JavaScript `null` and `undefined` write R missing values except for raw vectors. JavaScript `NaN` remains an ordinary R `NaN`. Factors, dates, and other classed vectors are rejected.

Normal R copy-on-write rules still apply. Replacing or modifying `view$value` in R after JavaScript retains the view can detach the R binding from the retained storage.

Value

A mutable-view wrapper containing an independent copy in `value`.

Examples

```
ctx <- JSContext$new(profile = "bare")
ctx$source(code = "function increment(x) { x[0] += 1; return x; }")
values <- js_mutable_view(c(1, 2, 3))
ctx$call("increment", values)
values$value
```

js_readonly_view

*Expose an R vector as a read-only JavaScript view***Description**

Creates an array-like JavaScript object that reads directly from retained R vector storage. Creating the view does not copy or convert its elements. Elements are converted only when JavaScript accesses them.

Usage

```
js_readonly_view(x)
```

Arguments

x A raw, logical, integer, double, or character vector.

Details

The view has length, indexed access, array iteration, and inherited non-mutating Array methods. Use `view.toArray()` or `Array.from(view)` when a mutable JavaScript array is required. Attempts to write or delete indexed values throw an error.

The wrapper does not copy the vector. The view retains the wrapper and vector until JavaScript releases them. Normal R assignment follows R's copy-on-write rules, so modifying an R binding can detach it from the storage retained by an existing view. Returning the view to R recovers the wrapper without materialising a JavaScript array.

Missing values use the ordinary conversion rules: logical, integer, double, and character NA become JavaScript null; NaN remains NaN. Factors yield their labels, and dates yield JavaScript dates.

Value

A lightweight wrapper around the input vector.

Examples

```
ctx <- JSContext$new(profile = "bare")
ctx$source(code = paste(
  "function endpoints(x) { return [x[0], x[x.length - 1]]; }",
  "function total(x) { return x.reduce((a, b) => a + b, 0); }"
))
```

```
values <- js_readonly_view(as.double(1:1000))
ctx$call("endpoints", values)
ctx$call("total", values)
```

js_ref_to_altrep	Create an R ALTREP view over a JavaScript typed array
------------------	---

Description

Creates a zero-copy R view over the storage of a retained JavaScript typed array. Uint8Array and Uint8ClampedArray produce raw vectors, Int32Array produces an integer vector, and Float64Array produces a double vector.

Usage

```
js_ref_to_altrep(ref)
```

Arguments

ref	A JSValueRef containing a supported typed array.
-----	--

Details

The backing ArrayBuffer is made permanently immutable before its address is exposed to R. This affects every JavaScript alias of that buffer. Indexed JavaScript writes are ignored, while mutating typed-array methods, DataView writes, resizing, and transfer operations throw an error.

The ALTREP object retains the reference and its context while it reads the JavaScript buffer. When R requests writable storage, the values are copied into an ordinary R vector, the R modification affects only that copy, and the JavaScript reference can be released.

Shared, detached, unsupported, and out-of-bounds typed-array buffers are rejected. An Int32Array value of -2147483648 is R's NA_integer_ sentinel. Float64 NaN payloads are preserved, including an R NA_real_ payload originally transferred to JavaScript.

ALTREP-aware R operations can read the buffer without copying. Other R code may request writable or materialised storage and therefore trigger a copy.

Value

An ALTREP raw, integer, or double vector backed by the JavaScript ArrayBuffer.

Examples

```
ctx <- JSContext$new(profile = "bare")
ref <- ctx$eval_ref("new Float64Array([1, 2, 3, 4])")
values <- js_ref_to_altrep(ref)
sum(values)
values[1]
```

js_typed_array	<i>Transfer an R vector as a JavaScript typed array</i>
----------------	---

Description

Marks a vector for contiguous transfer to JavaScript. Raw vectors become Uint8Array, integer vectors become Int32Array, and double vectors become Float64Array. Integer vectors containing NA are rejected. Double NA and NaN values both become JavaScript NaN. On return, the JavaScript Int32Array value -2147483648 becomes R's NA_integer_ sentinel.

Usage

```
js_typed_array(x)
```

Arguments

x	A raw, integer, or double vector.
---	-----------------------------------

Value

The input vector marked for typed-array conversion.

Examples

```
ctx <- JSContext$new()
ctx$source(code = "function total(x) { return x.reduce((a, b) => a + b, 0); }")
ctx$call("total", js_typed_array(as.double(1:4)))
```

JSCompiledScript	<i>Compile and repeatedly execute JavaScript</i>
------------------	--

Description

Compiled scripts avoid parsing source on every execution. The bytecode remains owned by its original context and can be executed repeatedly. The script keeps that context alive.

Usage

```
js_script_run(script)

js_script_run_ref(script)
```

Arguments

script	A script created by ctx\$compile(code).
--------	---

Details

Compilation does not execute the source. Compiled scripts cannot be passed as ordinary JavaScript values. For parameterised work, evaluate a JavaScript function with `ctx$eval_ref()` and call it repeatedly with `js_ref_call()`.

Value

`js_script_run()` converts the result to R. `js_script_run_ref()` returns a `JSValueRef` owned by the script's context.

Examples

```
ctx <- JSContext$new(profile = "bare")
ctx$eval("globalThis.value = 0")
script <- ctx$compile("value += 1; value")
js_script_run(script)
js_script_run(script)
```

JSContext

JSContext object

Description

An initialised context within which to evaluate Javascript scripts or commands.

Usage

```
JSContext
```

Value

A `JSContext` object containing an initialised JavaScript context for evaluating scripts/commands

JSContext-method-assign

Assign a value to a variable in the current context

Description

Assign a value to a variable in the current context

Usage

```
assign(var_name, value)
```

Arguments

var_name	The name of the variable to assign
value	The value to assign to the variable

Value

No return value, called for side effects

Examples

```
## Not run:
ctx <- JSContext$new()
ctx$assign("a", 1)
ctx$get("a")

## End(Not run)
```

JSContext-method-call *Call a JS function in the current context*

Description

Call a specified function in the JavaScript context with the provided arguments.

Usage

```
call(function_name, ...)
```

Arguments

function_name	The function to be called
...	The arguments to be passed to the function

Value

The result of calling the specified function

Examples

```
## Not run:
ctx <- JSContext$new()
ctx$source(code = "function add(a, b) { return a + b; }")
ctx$call("add", 1, 2)

## End(Not run)
```

JSContext-method-eval *Evaluate code in a persistent JavaScript context*

Description

Evaluates source code in an existing JSContext. Global state remains available to later evaluations, calls, and compiled scripts. Use `ctx$eval(code)` to invoke the method.

Usage

```
eval(code)
```

Arguments

code	JavaScript source code to evaluate.
------	-------------------------------------

Value

The evaluation result converted to R.

JSContext-method-get *Get a variable from the current context*

Description

Get the value of a variable from the current context

Usage

```
get(var_name)
```

Arguments

var_name	The name of the variable to retrieve
----------	--------------------------------------

Value

The value of the variable

Examples

```
## Not run:  
ctx <- JSContext$new()  
ctx$source(code = "var a = 1;")  
ctx$get("a")  
  
## End(Not run)
```

JSContext-method-source*Evaluate JS string or file in the current context*

Description

Evaluate a provided JavaScript file or string within the initialised context. Note that this method should only be used for initialising functions or values within the context, no values are returned from this function. See the `$call()` method for returning values.

Usage

```
source(file = NULL, code = NULL)
```

Arguments

<code>file</code>	A path to the JavaScript file to load
<code>code</code>	A single string of JavaScript to evaluate

Value

No return value, called for side effects

Examples

```
## Not run:  
ctx <- JSContext$new()  
ctx$source(file = "path/to/file.js")  
ctx$source(code = "1 + 2")  
  
## End(Not run)
```

JSContext-method-validate*Assess validity of JS code without evaluating*

Description

Checks whether JS code string is valid code in the current context

Usage

```
validate(code_string)
```

Arguments

<code>code_string</code>	The JS code to check
--------------------------	----------------------

Value

A boolean indicating whether code is valid

Examples

```
## Not run:
ctx <- JSContext$new()
ctx$validate("1 + 2")

## End(Not run)
```

JSContext-new

Create a JavaScript context

Description

Creates a persistent QuickJS runtime and context.

Usage

```
new_JSContext(
  stack_size = NULL,
  profile = c("host", "standard", "bare")
)
```

Arguments

stack_size	Optional maximum JavaScript stack size in bytes.
profile	Context facilities to install. See Details.

Details

Create contexts through `JSContext$new()`.

The host profile preserves the complete API with std, os, console helpers, and the global R bridge. The standard profile installs std, os, and console helpers without the global R bridge. The bare profile installs only the core JavaScript runtime. R values can still be passed through context methods in every profile.

Value

A persistent JSContext.

Examples

```
ctx <- JSContext$new(profile = "bare")
ctx$source(code = "function add(a, b) { return a + b; }")
ctx$call("add", 2, 3)
```

JSContext-reference-methods

Create persistent JavaScript references

Description

Evaluate, retrieve, or call JavaScript while retaining the result inside the context instead of eagerly converting it to R.

Usage

```
call_ref(function_name, ...)
```

```
get_ref(var_name)
```

```
eval_ref(code)
```

Arguments

function_name	The global function or dotted method path to call.
...	Arguments passed to the JavaScript function.
var_name	The global value or dotted property path to retrieve.
code	JavaScript source code to evaluate.

Value

A JSValueRef owned by the context.

JSValueRef

Persistent JavaScript value references

Description

Retain values inside their JavaScript context so chained operations do not convert intermediate values to R. References keep their context alive and can only be passed back to that same context.

Usage

```
js_ref_get(ref, name)
```

```
js_ref_call(ref, ...)
```

```
js_ref_to_r(ref)
```

Arguments

ref	A JSValueRef created by a JSContext.
name	A property name or dotted property path.
...	Arguments passed to the referenced JavaScript function.

Value

js_ref_get() and js_ref_call() return a JSValueRef. js_ref_to_r() materialises the referenced value as an R object.

Examples

```
ctx <- JSContext$new()
ref <- ctx$eval_ref("{ value: 2, add(x) { return this.value + x; } })")
result <- js_ref_call(js_ref_get(ref, "add"), 3)
js_ref_to_r(result)
```

qjs_eval

qjs_eval

Description

Evaluate a single Javascript expression.

Usage

```
qjs_eval(eval_string)
```

Arguments

eval_string	A single string of the expression to evaluate
-------------	---

Value

The result of the provided expression

Examples

```
# Return the sum of two numbers:
qjs_eval("1 + 2")

# Concatenate strings:
qjs_eval("'1' + '2'")

# Create lists from objects:
qjs_eval("var t = {'a' : 1, 'b' : 2}; t")
```

quickjs_flags	<i>QuickJS-NG Compiler and Linker Flags</i>
---------------	---

Description

Return the flags needed for compiling against the bundled quickjs-ng headers and library

Usage

```
cppflags()
```

```
ldflags()
```

Value

A character string of flags

quickjs_version	<i>Get the version of the bundled QuickJS library</i>
-----------------	---

Description

Get the version of the bundled QuickJS library

Usage

```
quickjs_version()
```

Value

Character string of the version of the bundled QuickJS library

*to_json**to_json*

Description

Use the QuickJS C API to convert an R object to a JSON string

Usage

```
to_json(arg, auto_unbox = FALSE)
```

Arguments

<code>arg</code>	Argument to convert to JSON
<code>auto_unbox</code>	Automatically unbox single element vectors

Value

JSON string

Index

`assign (JSContext-method-assign)`, [9](#)

`call (JSContext-method-call)`, [10](#)

`call_ref (JSContext-reference-methods)`,
[14](#)

`cppflags (quickjs_flags)`, [16](#)

`eval (JSContext-method-eval)`, [11](#)

`eval_ref (JSContext-reference-methods)`,
[14](#)

`from_json`, [3](#)

`get (JSContext-method-get)`, [11](#)

`get_ref (JSContext-reference-methods)`,
[14](#)

`js_columnar_data_frame`, [3](#)

`js_masked_typed_array`, [4](#)

`js_mutable_view`, [5](#)

`js_readonly_view`, [6](#)

`js_ref_call (JSValueRef)`, [14](#)

`js_ref_get (JSValueRef)`, [14](#)

`js_ref_to_altptr`, [7](#)

`js_ref_to_r (JSValueRef)`, [14](#)

`js_script_run (JSCompiledScript)`, [8](#)

`js_script_run_ref (JSCompiledScript)`, [8](#)

`js_typed_array`, [8](#)

`JSCompiledScript`, [8](#)

`JSContext`, [9](#)

`JSContext-method-assign`, [9](#)

`JSContext-method-call`, [10](#)

`JSContext-method-call_ref`
(JSContext-reference-methods),
[14](#)

`JSContext-method-eval`, [11](#)

`JSContext-method-eval_ref`
(JSContext-reference-methods),
[14](#)

`JSContext-method-get`, [11](#)

`JSContext-method-get_ref`
(JSContext-reference-methods),
[14](#)

`JSContext-method-source`, [12](#)

`JSContext-method-validate`, [12](#)

`JSContext-new`, [13](#)

`JSContext-reference-methods`, [14](#)

`JSValueRef`, [14](#)

`ldflags (quickjs_flags)`, [16](#)

`new_JSContext (JSContext-new)`, [13](#)

`qjs_eval`, [15](#)

`quickjs_flags`, [16](#)

`quickjs_version`, [16](#)

`QuickJSR (QuickJSR-package)`, [2](#)

`QuickJSR-package`, [2](#)

`source (JSContext-method-source)`, [12](#)

`to_json`, [17](#)

`validate (JSContext-method-validate)`, [12](#)