Package 'RserveTS'

September 24, 2026

Title Typed Application Contracts for 'Rserve'

Version 0.8.3

Description Defines a typed application contract between R backends and 'TypeScript' clients over 'Rserve'. Users specify the API architecture with typed object capability (Ocap) functions and compile matching 'TypeScript' schemas and deployment scripts for the server. The companion library 'rserve-ts', available on npm <https://www.npmjs.com/package/rserve-ts>, provides the client-side runtime that consumes those schemas.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Config/testthat/edition 3

Depends R (>= 4.1.0), objectProperties, objectSignals

Imports methods, rlang, Rserve, stats

Suggests testthat (>= 3.0.0), withr

URL https://tomelliott.co.nz/RserveTS/, https://github.com/tmelliott/RserveTS

BugReports https://github.com/tmelliott/RserveTS/issues

NeedsCompilation no

Author Tom Elliott [aut, cre] (ORCID: <https://orcid.org/0000-0002-7815-6318>)

Maintainer Tom Elliott <tom@inzight.co.nz>

Repository CRAN

Date/Publication 2026-09-24 04:40:02 UTC

Contents

`createWidget` *Create 'TypeScript'-compatible widgets*

Description

Widgets are stateful reference classes shared between R and 'TypeScript'. Use `createWidget()` to define one, wrap reactive methods with `observer()`, and optionally declare typed actions with `widgetActions()`. Instances inherit from the `tsWidget` reference class.

Usage

```
widgetActions(..., strict = "warn", enabled = TRUE)

createWidget(
  name,
  properties = list(),
  initialize = NULL,
  methods = list(),
  actions = FALSE,
  auto_flush = TRUE,
  .env = parent.frame(),
  ...
)

observer(props, fn)
```

Arguments

`...`	For `createWidget()`: passed to the underlying `ts_function()` constructor. For `widgetActions()`: named `ts_function()` action definitions.
`strict`	For `widgetActions()`: unknown action handling (`"off"`, `"warn"`, or `"strict"`).
`enabled`	For `widgetActions()`: whether action support is enabled.
`name`	Widget class name (character).
`properties`	Named list of typed properties (`ts_*()` objects, or nested widget constructors).
`initialize`	Optional function run after defaults are applied; receives the widget instance when it has a parameter.

methods	Named list of ts_function() methods and/or observer() reactive methods.
actions	FALSE/TRUE, a list with enabled/types/strict, or a widgetActions() object.
auto_flush	If TRUE (default), methods flush state to 'TypeScript' after they return; if FALSE, call updateState() manually.
.env	Environment for the ref class definition (default parent.frame()).
props	For observer(): property names that trigger the method.
fn	For observer(): method body (function or ts_function()).

Details

createWidget() returns a ts_function()-like constructor (class ts_widget) that 'JavaScript' calls with a state setter. Locally you can inspect or compile it with ts_compile() without a live 'Rserve' session; calling $call() needs an out-of-band 'JavaScript' setter.

Properties and methods:

Each entry in properties is a ts_*() type (optionally with default). Child widgets can be nested by passing another createWidget() result as a property value.

Each entry in methods is usually a ts_function() exported to 'JavaScript'. Use observer() to run a method when properties change (internal-only if the body is a plain function).

Actions:

Pass actions = widgetActions(...) to enable typed, named actions dispatched from 'JavaScript'. Each action must be a named ts_function() with exactly one payload argument. strict controls unknown action handling ("off", "warn", or "strict").

tsWidget **reference class:**

All widget instances inherit tsWidget and include:

- set(prop, value), get(prop) – field access with change tracking
- updateState(all = FALSE) – push changed properties to 'TypeScript'
- addPropHandler(prop, fn) – react to property changes
- batch(props, expr) – batch updates into one state flush
- add_child(property, widget_def) – attach a nested widget
- create_dynamic_child(widget_def) – runtime child widget
- destroy() – tear down the widget

Client apps ('rserve-ts' / React):

Compile widgets with ts_compile() and import the generated schema into a client app that connects to 'Rserve' via the 'rserve-ts' library. Obtain widget Ocaps from the compiled app schema (for example app.histogram after connecting with useRserve() in React).

In React, useWidget() from @tmelliott/react-rserve wraps a compiled widget constructor and keeps 'JavaScript' state in sync with R:

- state – current property values (from R updateState())
- set – update properties from the client
- methods – call exported ts_function() methods on the widget
- children – nested widget connectors when properties include child widgets

Action-enabled widgets (actions = widgetActions(...)) also expose capabilities, dispatchAction, undo, and redo on the hook return value.

Value

`createWidget()` returns a `ts_widget` constructor. `widgetActions()` returns a `ts_widget_actions` object. `observer()` returns a `ts_observer` object. `tsWidget` is the base reference class generator.

See Also

`ts_function()`, `ts_compile()`, type_objects, Package '@tmelliott/react-rserve'

Examples

```
Counter <- createWidget(
    name = "Counter",
    properties = list(count = ts_integer(1L, default = 0L)),
    methods = list(
        increment = ts_function(function(by = ts_integer(1L)) {
            .self$count <- as.integer(.self$count + by)
            .self$count
        }, result = ts_integer(1)),
        on_count = observer("count", function() NULL)
    )
)
inherits(Counter, "ts_widget")
ts_compile(Counter)
```

---

js_function *'JavaScript' functions callable from R*

Description

If `result` is `NULL`, it will be an oobSend (R process will continue); otherwise the R process will wait for a response (oobMessage).

Usage

```
js_function(..., result = NULL)
```

Arguments

`...`	arguments passed to the function
`result`	the type of value returned from 'JavaScript' to R

Value

A ts object that accepts 'JavaScript' functions as input. Using 'JavaScript' functions as output (R to 'JavaScript') is not supported yet.

Examples

```
# Fire-and-forget callback from 'JavaScript' (oobSend)
cb <- js_function(ts_character(1))

# Callback that returns a value to R (oobMessage)
ask <- js_function(ts_integer(1), result = ts_logical(1))
```

`ts_app` *Generate an 'Rserve' app from a* `ts_function()`

Description

Anything that is not a function simply returns itself. However, functions are wrapped with `Rserve::ocap()`, and the result is subsequently wrapped with `ts_app()`.

Usage

```
ts_app(x)
```

Arguments

`x`	A `ts_function()` object

Value

An object of class 'OCref', see `Rserve::ocap()`

Examples

```
f <- ts_function(function(x = ts_integer(1), y = ts_character(1)) {
    x + nchar(y)
}, result = ts_integer(1))
app <- ts_app(f) # class of 'OCref'
# this can now be used in an 'Rserve' application, for example
```

`ts_compile` *Compile R functions*

Description

Generates 'TypeScript' schema for the given R function or file path. If a path, the R app is also generated.

Usage

```
ts_compile(f, ...)
```

Arguments

f	A function or file path (length-one character string for file compilation).
...	Additional arguments. For the **file path** method, named arguments passed to ts_deploy() (e.g. init, port, run). For ts_function() / ts_widget objects, format and prettier_cmd are supported (see details); other arguments are ignored.

Details

ts_function() **method:** name defaults to deparse(substitute(f)) and sets the generated export const symbol.

Character (file) method: filename is the base path for output; .R and .ts extensions are appended. When omitted, output goes under the default compile directory (see below) as {basename(f)}.rserve. Arguments filename, format, and prettier_cmd must be passed by name; they are not part of ....

Default output directory (CRAN-safe; does not write beside the source by default):

1. option RserveTS.compile_dir if set to a non-empty path;
2. else environment variable RSERVETS_COMPILE_DIR if set;
3. else tempdir().

For local development, set e.g. options(RserveTS.compile_dir = ".") or RSERVETS_COMPILE_DIR=. so ts_compile("app.R") writes app.rserve.ts / app.rserve.R in the working directory; or pass filename explicitly.

- format – If TRUE, run 'Prettier' (or a compatible CLI) on the generated 'TypeScript' (returned string for functions; written .ts for files). Defaults to getOption("RserveTS.format", FALSE) so you can enable it once for a session (e.g. 'pkgdown' site builds) without changing call sites.
- prettier_cmd – Character vector argv (executable first). Defaults to getOption("RserveTS.prettier_cmd") (NULL until you set it, e.g. options(RserveTS.prettier_cmd = c("prettier", "--parser", "typescript"))). When still NULL, falls back to environment variable RserveTS_PRETTIER_CMD (space-separated tokens), then prettier or npx prettier on PATH. A temporary .ts copy of the generated source is appended as the last argument (as with prettier --parser typescript path/to/file.ts). Another formatter (e.g. 'Biome') can be used if it accepts that invocation pattern.

Value

For a ts_function() / ts_widget, a character string of 'TypeScript'. For a file path, writes .ts / .R beside filename and returns the output base path invisibly.

Examples

```
# Compile a typed function to a 'TypeScript' schema string (no files written)
f <- ts_function(function(x = ts_integer(1)) x + 1L, result = ts_integer(1))
ts_compile(f)

# File compilation writes under tempdir() by default (or RSERVETS_COMPILE_DIR)
```

```
src <- tempfile(fileext = ".R")
writeLines(
   "add <- ts_function(function(x = ts_integer(1)) x + 1L, result = ts_integer(1), export = TRUE)",
    src
)
out <- ts_compile(src)
file.exists(paste0(out, ".ts"))
file.exists(paste0(out, ".R"))
```

`ts_deploy` *Deploy a typed 'Rserve' app*

Description

Writes an 'Rserve' launcher script for an app source file. By default the script is written under the same directory as `ts_compile()` file output (`RserveTS.compile_dir` / `RSERVETS_COMPILE_DIR` / `tempdir()`) as `{basename(f)}.rserve.R`. Pass `file` explicitly to choose another path.

Usage

```
ts_deploy(
  f,
  file = NULL,
  init = NULL,
  port = 6311,
  run = c("no", "here", "background"),
  silent = FALSE
)
```

Arguments

`f`	The path to the application files
`file`	The file to write the deployment script to. When `NULL` (default), uses `{basename(f)}.rserve.R` under the default compile directory (see `ts_compile()`).
`init`	Names of `ts_function()` objects to make available to the initialisation function
`port`	The port to deploy the app on
`run`	Whether to run the deployment script, takes values "no", "here", "background"
`silent`	Whether to print the deployment script

Value

The path written to (`file`), invisibly. With `run = "here"` or `"background"`, also starts 'Rserve' as requested.

Examples

```
src <- tempfile(fileext = ".R")
writeLines(
   "add <- ts_function(function(x = ts_integer(1)) x, result = ts_integer(1), export = TRUE)",
    src
)
out <- ts_deploy(src, silent = TRUE, run = "no")
file.exists(out)
```

`ts_function` *Define a typed function*

Description

Define a typed function

Usage

```
ts_function(f, ..., result = ts_void(), export = FALSE)
```

Arguments

`f`	an R function
`...`	argument definitions (only required if f does not specify these in its formals)
`result`	return type (ignored if overloads are provided)
`export`	if `TRUE`, and defined in the global namespace of the app at compile time, the function will be part of the initial functions available to 'Rserve'; otherwise it will need to be sent as the result of another Ocap.

Details

Defining functions is the core of writing 'Rserve' apps. Functions are referred to as *object capabilities* (Ocaps), as they are 'objects' that allow 'JavaScript' to access capabilities of R with a restricted interface. Only arguments can be adjusted.

`ts_function()` objects can be defined using existing (named) or anonymous functions. Anonymous functions are useful in that the arguments to the functions can explicitly be defined with their types as formal arguments:

```
ts_function(function(x = ts_integer(), y = ts_string()) { ... })
```

Value

a `ts_function()` object which has a `call()` method that will call the function with the given arguments, which will be checked for type correctness.

Examples

```
f <- ts_function(function(x = ts_integer(1), y = ts_character(1)) {
    x + nchar(y)
}, result = ts_integer(1))
f$call(1, "hello")
```

`ts_object` *Typed object*

Description

This is the base type for all typed objects, and can be used to define custom types.

Usage

```
ts_object(
  input_type = "any",
  return_type = "any",
  default = NULL,
  check = function() stop("Not implemented"),
  generic = FALSE
)

is_ts_object(x)

get_type(x, which = c("input", "return"))

check_type(type, x)
```

Arguments

`input_type`	The type of the object that 'TypeScript' expects to send to R.
`return_type`	The type of the object that 'TypeScript' expects to receive from R.
`default`	The default value of the object.
`check`	A function that checks the object and returns it if it is valid. This operates on the R side and is mostly for development and debugging purposes. It is up to the developer to ensure that all functions return the correct type of object always.
`generic`	logical, if `TRUE` then the object is a generic type.
`x`	An object
`which`	Which type to get, either "input" or "return"
`type`	A ts object

Value

A `ts_object` environment with 'Zod' input/return schema strings and a `check()` helper. `is_ts_object()` returns a logical; `get_type()` returns a character schema string; `check_type()` returns `x` when valid (or errors).

Functions

- `is_ts_object()`: Check if an object is a ts object
- `get_type()`: Get the input type of a ts object
- `check_type()`: Check if an object has the correct type

Examples

```
x <- ts_numeric(1)
is_ts_object(x)
get_type(x, "input")
```

`ts_recursive_list` *Recursive list*

Description

For complex recursive lists — objects that can contain subcomponents of the same (parent) type. For example, a Person with `name` and optional `children` that are themselves Person objects.

Usage

```
ts_recursive_list(values, recur)

ts_self(n = -1L)
```

Arguments

`values`	properties that define the base schema of the list; must be a named list.
`recur`	a named list of properties that are added. These can use `ts_self()`.
`n`	For `ts_self()`: number of elements — `n = 1` for a single nested object, or `n != 1` (default `-1`) for an array of the parent type.

Details

Use `ts_self()` inside `recur` to mark those self-referential fields. By default `ts_self()` means an array of the parent type; use `ts_self(1)` for a single nested object.

Defining this type in 'Zod' is currently complicated, as the type has to be pre-defined, and then extended after manually defining the Type. In an upcoming version of 'zod' 4, this should be simplified. For now, it's tricky.

Value

`ts_recursive_list()` returns a ts object that accepts recursive lists. `ts_self()` returns a marker used in `recur`.

See Also

Other type documentation: type_objects

Examples

```
person <- ts_recursive_list(
    list(name = ts_character(1)),
    list(children = ts_self())
)
echo_person <- ts_function(function() person, result = person)
ts_compile(echo_person, name = "echo_person")
```

---

type_objects *Types in R and 'TypeScript'*

Description

Constructors for typed values used in 'RserveTS' app contracts. Each `ts_*()` helper returns a `ts_object` that describes the 'zod' / `Robj` schema 'TypeScript' clients should expect for inputs and returns.

Usage

```
ts_union(..., default = NULL)

ts_optional(type)

ts_array(type)

ts_logical(n = -1L, default = NULL)

ts_integer(n = -1L, default = NULL)

ts_numeric(n = -1L, default = NULL)

ts_character(n = -1L, default = NULL)

ts_factor(levels = NULL, default = NULL)

ts_list(..., default = NULL)

ts_record(value_type, default = NULL)

ts_dataframe(..., default = NULL)

ts_null()
```

```
ts_void()

ts_undefined()
```

Arguments

`...`	For `ts_list()` / `ts_dataframe()`: member types (named or unnamed for lists; named for data frames). For `ts_union()`: type objects to merge.
`default`	Default value for the type (optional).
`type`	For `ts_optional()` / `ts_array()`: the inner type. For `ts_array()`, may also be a 'zod'-style string such as `"z.number()"`.
`n`	Length of the vector for atomic types. If `n = 1`, a single value is expected; if `n = 0`, any length; if `n > 1`, a vector of that length. The default (`-1`) accepts scalar or array form.
`levels`	For `ts_factor()`: character vector of allowed levels (optional).
`value_type`	For `ts_record()`: a single ts type for all values (e.g. `ts_character(1)`).

Value

A `ts_object` describing the type (except `ts_array()` on a character 'Zod' fragment, which returns a character schema string).

TS objects

The basic object in 'RserveTS' is a `ts_object`. It carries an input type, a return type, an optional default, and a `check()` helper used on the R side during development.

Input types describe the 'zod' schema of objects that 'TypeScript' can pass to 'Rserve' functions. Return types describe the 'zod' schema of objects that 'Rserve' functions return; most utilise the `Robj` helpers in the 'rserve-ts' library (with `r_type` and `r_attributes`).

Scalar versus array ("vector") types

In R, almost all types are vectors. In the 'rserve-js' library, primitive arrays of length one are converted into scalars, which leads to type checking issues when a return value has unknown length (e.g. `which(x > 5)`).

For vectors that support this distinction, pass `n`:

- `n = 1` – scalar form
- `n != 1` (including `0`) – array form
- default (`n = -1`) – union of scalar and array forms

This applies to logicals, integers, numerics, and characters.

Atomic types

- `ts_logical()`: logical / boolean. Array form: `Int8Array` / `Uint8Array`.
- `ts_integer()`: integer. Array form: `Int32Array`. 'JavaScript' has no native integer type, so scalars are numbers (same as `ts_numeric()`).
- `ts_numeric()`: numeric / double. Array form: `Float64Array`.
- `ts_character()`: character / string. Array form: `string[]`.
- `ts_factor()`: factor. Always a string array on the 'JavaScript' side (even for a single value); optional `levels` constrain the labels.

Structured types

- `ts_list()`: a list (named object or array in 'JavaScript'). Forms:
  1. Unknown list – `ts_list()`
  2. Known named list – `ts_list(x = ts_integer(), y = ts_character())` (object in 'JavaScript')
  3. Known unnamed list – `ts_list(ts_integer(), ts_character())` (array in 'JavaScript')
  4. Named list of one value type with unknown keys – use `ts_record(value_type)` (`Record<string, type>` in 'TypeScript')
  5. Unnamed list of one value type with unknown length – `ts_list(ts_integer())`-style homogeneous arrays (`Array<type>`)
- `ts_record()`: named list whose values share one type (`Record<string, value_type>`).
- `ts_dataframe()`: data frame; named columns of equal length.

Nullish and combinators

- `ts_null()`: only `NULL`.
- `ts_void()`: return type for functions that return nothing (prefer this over `ts_null()` for return types).
- `ts_undefined()`: 'JavaScript' `undefined`.
- `ts_union()`: union of several types.
- `ts_optional()`: type or undefined (wrapper around `ts_union()` / `ts_undefined()`).
- `ts_array()`: array of a typed element (or a 'zod' fragment string).

See `ts_recursive_list()` for self-referential list schemas, and `js_function()` for 'JavaScript' callbacks callable from R.

See Also

`ts_object()`, `ts_recursive_list()`, `js_function()`

Other type documentation: `ts_recursive_list()`

Examples

```
(x <- ts_numeric(1))
(person <- ts_list(name = ts_character(1), age = ts_integer(1)))
(df <- ts_dataframe(a = ts_integer(1), b = ts_character(1)))
(labels <- ts_record(ts_character(1)))
(ts_union(ts_numeric(1), ts_character(1)))
```

Index