Package: qupflowR (via r-universe)

November 3, 2026

Title Connect R and 'Shiny' to 'QuPath' Through a Local Bridge

Version 0.0.0.9000

Description Development version of an interface between R and the 'QuPath' 0.7.0 platform for digital pathology image analysis. The planned design lets R talk to a local 'Java' bridge (a 'QuPath' extension and a headless worker) over an authenticated loopback protocol, adds a 'Shiny' app that is built on the same R functions, and packages analysis workflows so they can be shared and run again. This version contains only the package foundations: classified error conditions, a message catalogue, protocol JSON helpers and a JSON Schema subset validator with the protocol schemas, plus a hash-pinned external-method execution boundary. The bridge client, the app and the workflow engine are not implemented yet.

License GPL (>= 3)

URL https://github.com/CTTIR/qupflowR

BugReports https://github.com/CTTIR/qupflowR/issues

Depends R (>= 4.5.0)

Imports cli, jsonlite, stats, tibble, utils

Suggests bslib, callr, chromote, covr, data.table, digest, DT, fs, htmltools, htmlwidgets, httr2, hyperannotatR, knitr, lintr, plotly, png, processx, promises, R6, rmarkdown, sf, shiny, shinytest2, testthat (>= 3.0.0), tiff, withr, zip

Remotes CTTIR/hyperannotatR

VignetteBuilder knitr

SystemRequirements QuPath (0.7.0) and a Java runtime (25) for the optional QuPath bridge; neither is needed to install or load the package.

Config/Needs/website CTTIR/themakR

Config/testthat/edition 3

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.0.0

Config/pak/sysreqs default-jdk

Repository https://cttir.r-universe.dev

Date/Publication 2026-10-28 07:29:13 UTC

RemoteUrl https://github.com/CTTIR/qupflowR

RemoteRef HEAD

RemoteSha cc4eea950910d8e98bc60d629904b7a132ae3132

Contents

qp_abort . . . 2
qp_condition_code . . . 4
qp_error_codes . . . 4
qp_error_from_response . . . 5
qp_json_canonical . . . 6
qp_json_encode . . . 7
qp_msg . . . 9
qp_redact . . . 10
qp_run_external . . . 11
qp_scalar . . . 12
qp_schema_assert . . . 13
qp_schema_path . . . 14
qp_schema_validate . . . 14
qp_value_state . . . 15
qp_warn . . . 16
qp_warnings_from_response . . . 17

Index **19**

`qp_abort` *Signal a classified qupflowR error*

Description

`qp_abort()` is the single way qupflowR raises errors. It looks up `code` in the protocol error registry, builds the condition classes `c(class, <registry class>, "qp_error")` and formats the message from the message catalogue in the language chosen by the option `qupflowR.language` (see `qp_msg()`).

Usage

```
qp_abort(
  code,
  ...,
  class = NULL,
  details = list(),
  request_id = NULL,
  action = NULL,
  message_key = NULL,
  bullets = NULL,
  parent = NULL,
  call = parent.frame()
)
```

Arguments

code	Error code, a single string of upper-case letters, digits and underscores such as "HOST_BUSY". Codes that are not in the registry (qp_error_codes()) are kept, but the condition then only has the class qp_error.
...	Named values for the placeholders of the message template, for example arg = "x" for error.invalid_argument. Values are inserted as text and never evaluated.
class	Additional condition classes placed before the registry class.
details	Named list with machine-readable context. It is passed through qp_redact(), so tokens are removed and long values truncated.
request_id	Protocol request ID (UUID string) or NULL.
action	Short machine-readable suggested action (for example "reconnect") or NULL.
message_key	Message key that replaces the registry message of the code, or NULL to use the registry message.
bullets	Optional character vector of additional message lines. Names are cli bullet types ("x", "i", "*", "!", "v"); the text is shown literally.
parent	Parent condition or NULL.
call	Environment of the function reported as the error call.

Value

Does not return; signals a condition inheriting from qp_error with the fields code, message_key, details, request_id, action, retryable (logical, NA if unknown) and http_status (integer, NA for codes that are never sent over HTTP). Its class is the registry class of code, for example qp_busy_error for HOST_BUSY. An invalid code, class, details, request_id, action, message_key or bullets argument signals a qp_validation_error with code INVALID_ARGUMENT instead.

See Also

qp_error_codes(), qp_condition_code(), qp_redact()

Examples

```
cnd <- tryCatch(
  qp_abort("HOST_BUSY", details = list(reason = "connection_limit")),
  qp_error = function(e) e
)
class(cnd)
cnd$code
cnd$retryable
```

`qp_condition_code` *Code of a qupflowR condition*

Description

Code of a qupflowR condition

Usage

```
qp_condition_code(cnd)
```

Arguments

`cnd`	A condition object.

Value

The condition's `code` as a single string, or `NA_character_` when `cnd` is not a qupflowR condition with a code. It signals no condition of its own, whatever `cnd` is.

Examples

```
cnd <- tryCatch(qp_abort("REVISION_CONFLICT"), error = function(e) e)
qp_condition_code(cnd)
qp_condition_code(simpleCondition("plain"))
```

`qp_error_codes` *Protocol error code registry*

Description

The registry lists every error code used by the bridge protocol and by the R-side workflow, storage, JSON and schema layers. The same file (`inst/schema/protocol/v1/error-codes.json`) is read by the Java bridge.

Usage

```
qp_error_codes()
```

Value

A tibble with one row per code and the columns `code` (character), `http_status` (integer; `NA` for codes that are never sent over HTTP), `r_class` (character), `retryable` (logical), `message_key` (character) and `source` (character, the contract that defines the code). It takes no arguments and signals no condition with the registry that is installed with the package.

Examples

```
codes <- qp_error_codes()
codes[codes$retryable, c("code", "http_status")]
```

`qp_error_from_response` *Map a protocol error response to a classified condition*

Description

Converts the `error` object of a decoded protocol v1 error envelope into the matching qupflowR condition. Known codes get their registry class; unknown codes keep their code with the class `qp_error` only. The host's own message is added as an extra line, and `details` are redacted.

Usage

```
qp_error_from_response(envelope, signal = TRUE, call = parent.frame())
```

Arguments

`envelope`	Decoded error envelope, a named list as returned by `qp_json_decode()` with the fields `request_id` and `error`.
`signal`	If `TRUE` (default) the condition is signalled; if `FALSE` it is returned.
`call`	Environment reported as the call of the condition.

Value

With `signal = FALSE` the condition object; otherwise the function does not return. A malformed envelope signals a `qp_version_error` with code `PROTOCOL_VIOLATION`.

Examples

```
envelope <- qp_json_decode(paste0(
  '{"protocol_version":"1.0",',
  '"request_id":"0d9a3c5e-8f7b-4a21-9c3d-2e1f0a9b8c7d",',
  '"instance_id":"5b1f8a2c-3d4e-4f60-8a9b-0c1d2e3f4a5b",',
  '"error":{"code":"HOST_BUSY","message_key":"error.host_busy",',
  '"message":"Too many open connections; retry later",',
  '"details":{"reason":"connection_limit"},"retryable":true},',
  '"warnings":[]}'
```

```
))
cnd <- qp_error_from_response(envelope, signal = FALSE)
class(cnd)
cnd$retryable
```

`qp_json_canonical` *Canonical JSON*

Description

Produces the canonical JSON text used as digest input. Two documents with the same JSON value have byte-identical canonical forms in R and in the Java bridge.

Usage

```
qp_json_canonical(x)
```

Arguments

x	An R value following the conventions of `qp_json_encode()`, or JSON text marked with `qp_as_json()` (the result of `qp_json_encode()` is already marked). Plain character strings are values, not JSON text.

Details

Rules:

1. No insignificant whitespace and no trailing newline.
2. Object members are sorted by their names compared as sequences of Unicode code points (the same order as comparing the UTF-8 bytes). This differs from RFC 8785, which compares UTF-16 code units, for names that mix characters above U+FFFF with characters from U+E000 to U+FFFF. Names must be unique and non-empty.
3. Strings are UTF-8. `"` and `\` are escaped as `\"` and `\\`; U+0008, U+0009, U+000A, U+000C and U+000D as `\b`, `\t`, `\n`, `\f` and `\r`; other characters below U+0020 as `\u00xx` with lowercase hex digits. Every other character, including non-ASCII, is written literally.
4. Numbers must be integers with magnitude at most 2^53 - 1, written as decimal digits with an optional minus sign, without exponent, fraction or leading zeros; negative zero is written as `0`. Non-integer numbers are rejected (`JSON_UNREPRESENTABLE`, reason `non_integer_number`) because shortest round-trip formatting of doubles is not guaranteed to agree between languages; digested documents store fractional values as decimal strings. In JSON text input every number token with a fraction or exponent is rejected, even `1.0`.
5. Arrays keep their order; `true`, `false` and `null` are literals.

Value

A single UTF-8 string with class `json`. Use `charToRaw(qp_json_canonical(x))` for the digest input bytes. A value that breaks rules 2 or 4 (or cannot be JSON at all) signals a `qp_serialization_error` with code `JSON_UNREPRESENTABLE`; invalid JSON text signals a `qp_serialization_error` with code `JSON_INVALID`. Both carry the rule in `details$reason`.

Examples

```
qp_json_canonical(qp_as_json('{"b": 2, "a": [true, null, "\\u00e9"]}'))
qp_json_canonical(list(z = qp_scalar(1L), a = qp_scalar("x")))
```

`qp_json_encode` *Encode and decode protocol JSON*

Description

`qp_json_encode()` turns R values into JSON text and `qp_json_decode()` turns JSON text into R values, following one explicit convention so that `qp_json_encode(qp_json_decode(txt))` gives back the same JSON value.

Usage

```
qp_json_encode(x, schema_hint = NULL)

qp_json_decode(txt)
```

Arguments

`x`	R value to encode.
`schema_hint`	`NULL`, or a schema name for `qp_schema_path()` or the path of a schema file. With a hint, unmarked length-1 atomic values are unboxed wherever the schema only allows a single value, and stay arrays where the schema allows arrays.
`txt`	JSON text: a single string, a raw vector of UTF-8 bytes, or text marked with `qp_as_json()`.

Value

`qp_json_encode()` returns a single UTF-8 string with class `json`. `qp_json_decode()` returns the decoded R value.

Encoding convention

- `NULL` becomes `null`.
- A `qp_scalar()` value (class `scalar`, as made by `jsonlite::unbox()`) becomes a single number, string, `true` or `false`; `NA` becomes `null`.
- Every other atomic vector becomes an array, at length 1 and 0 as well; `NA` elements become `null`.
- An unnamed list, including `list()`, becomes an array.
- A named list, including `qp_empty_object()`, becomes an object.
- Length-1 vectors are never unboxed silently: mark scalars with `qp_scalar()` or pass `schema_hint`.

- Doubles that hold whole numbers with magnitude up to 2^53 are written as plain digits (`5000000000`, never `5e+09`). Other doubles use the shortest of 15, 16 or 17 significant digits that reads back to the same double.
- `NaN`, `Inf` and `-Inf` are rejected with a `qp_serialization_error` (code `JSON_UNREPRESENTABLE`); JSON has no such literals. Measurement values use `value` plus `value_state`, see `qp_value_state()`.
- Strings are written as UTF-8. Only `"`, `\` and control characters are escaped; invalid UTF-8 is rejected.
- Object names must be non-empty where given, unique and valid UTF-8.
- Factors, dates, matrices, data frames, named atomic vectors, raw vectors, functions and environments are rejected instead of being guessed.

Decoding convention

- Objects become named lists (`{}` becomes `qp_empty_object()`), `null` becomes `NULL`.
- Single values become length-1 vectors marked with `qp_scalar()`.
- Arrays whose elements are all strings, all numbers or all booleans (with optional `null`s) become atomic vectors with `NA` for `null`; other arrays become unnamed lists. `[]` becomes `list()`.
- Numbers without fraction or exponent that fit a 32-bit integer become integers, all others doubles. Integers beyond 2^53 lose precision like in any double-based reader; the protocol does not use them.
- Duplicate or empty object keys, a byte-order mark, invalid UTF-8, escaped NUL characters, unpaired surrogate escapes, numbers outside the double range and nesting deeper than 128 levels are rejected (`JSON_INVALID` or `JSON_UNREPRESENTABLE`). So are comments, white space other than space, tab, line feed and carriage return between tokens, and an unterminated string, also after a complete value as in `[1]"` (`JSON_INVALID`, reason `syntax`), as RFC 8259 and the Java bridge require. R cannot tell an empty member name from a missing one, so empty names are not supported in either direction.
- The checks before parsing take time proportional to the length of the text.

See Also

`qp_json_canonical()`, `qp_schema_validate()`

Examples

```
qp_json_encode(list(
  image_id = qp_scalar("1"),
  object_ids = "8f6589c0-c587-4bfa-b740-8cab06bc8ce1",
  size_bytes = qp_scalar(5e9),
  name = NULL,
  metadata = qp_empty_object()
))
x <- qp_json_decode('{"plane":{"c":null,"z":0,"t":1},"ids":["a"]}')
x$ids
qp_json_encode(x)
```

qp_msg *Localised message text*

Description

Looks up key in the message catalogue (inst/i18n/messages.json) and fills the placeholders of the template with the named values in .... The language comes from .lang, which defaults to the option qupflowR.language ("de" when unset). A key missing in that language falls back to English, and a key missing in the catalogue is returned unchanged.

Usage

```
qp_msg(key, ..., .lang = getOption("qupflowR.language", "de"))
```

Arguments

key	Message key such as "error.host_busy".
...	Named values for the placeholders.
.lang	Language code. Region suffixes are ignored ("de_DE" uses "de"); unknown languages fall back to English.

Details

Templates use cli inline markup, for example {.arg {arg}}. Values are inserted as text: braces inside a value are shown literally and are never evaluated. A placeholder without a value is shown as <name>. The result is plain text without ANSI styling, so it can be used in the Shiny app.

Value

A single string. A key that is not a single message key, or unnamed values in ..., signal a qp_validation_error with code INVALID_ARGUMENT; an unknown key is not an error.

See Also

qp_abort()

Examples

```
qp_msg("error.host_busy", .lang = "en")
qp_msg("error.invalid_argument", arg = "image_id",
       expected = "a single string", .lang = "en")
qp_msg("error.host_busy", .lang = "de")
```

`qp_redact` *Redact and bound condition details*

Description

Removes credentials and bounds the size of values before they are stored in a condition, a log or a diagnostic report.

Usage

```
qp_redact(x, max_chars = 500L, max_items = 50L, max_depth = 5L)
```

Arguments

`x`	Value to redact, usually a named list.
`max_chars`, `max_items`, `max_depth`	Positive whole numbers bounding the string length, the number of elements and the nesting depth.

Details

Rules:

- Elements whose name contains `token`, `secret`, `password`, `passwd`, `authorization`, `cookie`, `credential(s)`, `api_key` or `private_key` as a word part are replaced by `"<redacted>"`. Names ending in `_file`, `_path` or `_dir` (for example `token_file`) name locations, not secrets, and are kept.
- In every string, `Bearer <credential>` and `token=`, `access_token=`, `api_key=`, `password=` or `secret=` query values are replaced, and a string that consists only of a 43-character base64url value (the form of a 256-bit bridge token) becomes `"<redacted>"`.
- Strings longer than `max_chars` characters are truncated with a note.
- Vectors and lists longer than `max_items` keep their first `max_items` elements and gain the attribute `truncated_from` (original length).
- Lists nested deeper than `max_depth` and raw vectors, functions, environments and other non-data objects are replaced by a short description.
- JSON scalar marks (`qp_scalar()`) are removed, so decoded protocol details become plain R values.

Value

The redacted value with the same structure where possible. Bounds that are not positive whole numbers signal a `qp_validation_error` with code `INVALID_ARGUMENT`.

Examples

```
qp_redact(list(
  token = "abc",
  header = "Authorization: Bearer abc.def",
  token_file = "/run/user/1000/qupflowr/token"
))
```

`qp_run_external` *Run a pinned external method in an isolated process*

Description

Execute an explicitly trusted method without rewriting its script or analytical settings. This compatibility boundary does not use the live Java bridge and does not imply support for a different QuPath version. No program starts until trust and all file hashes have been checked.

Usage

```
qp_run_external(
  program,
  args,
  pins,
  outputs,
  workspace,
  timeout = 600,
  trust = FALSE,
  env = character()
)
```

Arguments

`program`	Absolute path to the executable, also included in `pins`.
`args`	Character argument vector. Literal `{{workspace}}` tokens expand to the new output directory; no shell interprets arguments.
`pins`	Data frame with `path` and `sha256` character columns. Pin the executable and every script, model, runtime and input used by the method. Hashes are checked before and after execution. The function cannot infer undeclared transitive dependencies of an arbitrary executable.
`outputs`	Non-empty vector of relative output file paths. Files must be non-empty and resolve inside the workspace. Their hashes are recorded. Presence does not validate analytical content: the pinned method must perform its own semantic validation before returning successfully.
`workspace`	A new, not-yet-existing directory in an existing parent. It is retained on both success and failure, with logs and a run receipt.
`timeout`	Finite positive time limit in seconds.

`trust`	Exactly `TRUE` to authorize execution of the pinned code. Hash integrity is not a substitute for deciding to trust that code.
`env`	Named character vector of additional environment variables. Home, display, temporary-directory and Java-option overrides are forbidden. Do not supply secrets: the full explicit child environment is recorded.

Details

The child receives a stripped environment, private home/XDG/Java preference and temporary directories, no display, and a bounded timeout. Private temporary directories are removed after the child exits. This is process hygiene, not a sandbox for malicious trusted code. Only processes started by this call are cleaned up. Existing output directories are never reused. No downloads, dependency installation or silent runtime upgrade occur. A nonzero exit or missing output never counts as completion. The initial execution profile supports Linux only; other platforms fail before creating a workspace. In particular, no Windows atomic-replacement or process-cleanup behavior is claimed without platform qualification.

Value

A list with `status = "execution_complete"`, command, pinned file and output hashes, process exit status, timestamps and workspace. This proves execution and output presence, not scientific reproduction, successful project saving or semantic validity of the outputs.

Examples

```
# Configure executable, script/model/input SHA-256 pins and output paths
# before explicitly authorizing a method with trust = TRUE.
```

---

qp_scalar *Mark JSON values*

Description

`qp_scalar()` marks a length-1 atomic value as a JSON scalar (class `scalar`, compatible with `jsonlite::unbox()`); `NA` encodes as `null`. `qp_empty_object()` returns the empty named list that encodes as `{}`. `qp_as_json()` marks a string as JSON text for `qp_json_canonical()` and `qp_schema_validate()`.

Usage

```
qp_scalar(x)

qp_empty_object()

qp_as_json(x)
```

Arguments

x	For qp_scalar(), an atomic vector of length 1 or NULL. For qp_as_json(), a single string or a raw vector of UTF-8 bytes.

Value

qp_scalar(): x with class scalar (NULL stays NULL). qp_empty_object(): structure(list(), names = character(0)). qp_as_json(): a single UTF-8 string with class json. An x of the wrong type or length signals a qp_validation_error with code INVALID_ARGUMENT; text that is not UTF-8, contains NUL or starts with a byte order mark signals a qp_serialization_error with code JSON_INVALID.

Examples

```
qp_json_encode(list(a = qp_scalar(1L), b = 1L, c = qp_empty_object()))
qp_as_json('{"a": 1}')
```

---

qp_schema_assert *Assert that JSON matches a schema*

Description

Assert that JSON matches a schema

Usage

```
qp_schema_assert(x, schema, call = parent.frame())
```

Arguments

x	Value to validate: an R value (it is encoded with qp_json_encode() first) or JSON text marked with qp_as_json().
schema	Schema name for qp_schema_path() or the path of a schema file.
call	Environment reported as the call of the error.

Value

x, invisibly, when it is valid. Otherwise a qp_validation_error with code VALIDATION_FAILED is signalled; its details hold schema and up to 20 errors (each with path, keyword and message).

Examples

```
try(qp_schema_assert(list(status = qp_scalar("ready")), "health"))
```

qp_schema_path *Locate an installed JSON Schema*

Description

Locate an installed JSON Schema

Usage

```
qp_schema_path(name, set = "protocol", version = "v1")
```

Arguments

name	Schema name without extension, for example "health" or "patch-request".
set	Schema set, a directory below inst/schema (default "protocol").
version	Major version directory (default "v1").

Value

The absolute path of <set>/<version>/<name>.schema.json inside the installed package. A missing schema signals a qp_validation_error with code SCHEMA_INVALID.

Examples

```
basename(qp_schema_path("envelope-error"))
```

qp_schema_validate *Validate JSON against a schema*

Description

Validates a value with the project's Draft-07 subset validator, the same rules the Java bridge applies to the shared fixture corpus.

Usage

```
qp_schema_validate(x, schema)
```

Arguments

x	Value to validate: an R value (it is encoded with qp_json_encode() first) or JSON text marked with qp_as_json().
schema	Schema name for qp_schema_path() or the path of a schema file.

Details

Supported keywords: `$schema`, `$id`, `$ref` (local only), `definitions`, `title`, `description`, `type`, `properties`, `required`, `additionalProperties`, `items` (single schema), `minItems`, `maxItems`, `uniqueItems`, `enum`, `const`, `minimum`, `maximum`, `exclusiveMinimum`, `exclusiveMaximum`, `minLength`, `maxLength`, `pattern`, `format`, `oneOf`, `anyOf`, `allOf` and `not`. Formats are asserted: `uuid` (lower-case 8-4-4-4-12 hex), `sha256` (64 lower-case hex digits), `date-time` (RFC 3339 with upper-case `T` and `Z`, seconds 00-59) and `uri-reference` (no white space, control characters, `<`, `>`, `"`, `{`, `}`, `|`, backslash, `^` or backtick, and valid percent escapes).

Reporting rules, shared with Java: every failing keyword is reported with the JSON pointer of the instance location. `required` is reported at the object, `additionalProperties: false` at the extra member and a `false` schema with the keyword `false`. `oneOf`, `anyOf` and `not` report a single error at their location and none of their branch errors. Errors of `$ref` targets and `allOf` branches are reported as they are. Keywords that do not apply to the instance type are ignored, as in Draft-07.

Value

A `qp_validation_report`: a list with `valid` (logical), `schema` (file name) and `errors`, a tibble with the character columns `path` (JSON pointer, `""` for the root), `keyword`, `message` and `schema_path` (file and JSON pointer of the failing keyword), sorted by `path` and `keyword`. An unsupported or unresolvable schema signals a `qp_validation_error` with code `SCHEMA_INVALID`.

See Also

`qp_schema_assert()`

Examples

```
health <- qp_as_json(paste0(
  '{"status":"ready","host_type":"worker","qupath_version":"0.7.0",',
  '"bridge_version":"0.1.0-SNAPSHOT","protocol_version":"1.0",',
  '"fx_thread_blocked":true}'
))
report <- qp_schema_validate(health, "health")
report$valid
report$errors
```

---

qp_value_state *Special measurement values*

Description

The protocol carries a measurement as `value` plus `value_state`. Only the state `finite` has a JSON number; the states `missing`, `nan`, `pos_inf` and `neg_inf` have `value = null`. In R they correspond to `NA_real_`, `NaN`, `Inf` and `-Inf`.

Usage

```
qp_value_state(x)

qp_value_from_state(value, value_state)

qp_value_fields(x)
```

Arguments

x	Numeric vector (integer or double).
value	Numeric vector (with NA where the state is not finite), a logical vector of NA, or NULL for a single non-finite state.
value_state	Character vector of states, parallel to value.

Value

qp_value_state(): character vector of states parallel to x. qp_value_from_state(): double vector. qp_value_fields(): named list with the scalar-marked fields value (or NULL) and value_state, ready for qp_json_encode(). Non-numeric input, unknown states, a state that contradicts its value or vectors of different lengths signal a qp_validation_error with code INVALID_ARGUMENT.

Examples

```
x <- c(1.5, NA, NaN, Inf, -Inf)
state <- qp_value_state(x)
state
value <- ifelse(state == "finite", x, NA_real_)
identical(qp_value_from_state(value, state), x)
qp_json_encode(qp_value_fields(NaN))
```

---

qp_warn *Signal a classified qupflowR warning*

Description

Signal a classified qupflowR warning

Usage

```
qp_warn(
  code,
  ...,
  message_key = NULL,
  class = NULL,
  details = list(),
  bullets = NULL,
  call = parent.frame()
)
```

Arguments

code	Warning code, a single string of upper-case letters, digits and underscores.
...	Named values for the placeholders of the message template.
message_key	Message key; by default "warning." followed by the lower-case code.
class	Additional condition classes placed before qp_warning.
details	Named list with machine-readable context, redacted with qp_redact().
bullets	Optional character vector of additional message lines shown literally (see qp_abort()).
call	Environment of the function reported as the warning call.

Value

NULL, invisibly. Signals a warning inheriting from qp_warning with the fields code, message_key and details. An invalid code, message_key, class, details or bullets argument signals a qp_validation_error with code INVALID_ARGUMENT instead.

Examples

```
w <- tryCatch(
  qp_warn("SNAPSHOT_NEAR_EXPIRY", message_key = "warning.unknown_code"),
  qp_warning = function(w) w
)
w$code
```

---

qp_warnings_from_response

Signal the warnings of a protocol response

Description

Signal the warnings of a protocol response

Usage

```
qp_warnings_from_response(envelope, signal = TRUE, call = parent.frame())
```

Arguments

envelope	Decoded protocol envelope (named list) whose warnings field is an array of warning objects with code, message_key, message and optional details.
signal	If TRUE (default) each warning is signalled as a qp_warning; if FALSE the conditions are returned instead.
call	Environment reported as the call of the warnings.

Value

A list of qp_warning conditions (invisibly when signalled). A malformed warning entry becomes a warning with code PROTOCOL_VIOLATION. An envelope that is not a named list, or an invalid signal, signals a qp_validation_error with code INVALID_ARGUMENT.

Examples

```
envelope <- list(warnings = list(list(
  code = qp_scalar("SNAPSHOT_NEAR_EXPIRY"),
  message_key = qp_scalar("warning.snapshot_near_expiry"),
  message = qp_scalar("Snapshot expires soon")
)))
ws <- qp_warnings_from_response(envelope, signal = FALSE)
ws[[1]]$code
```

Index

jsonlite::unbox(), *7*, *12*

qp_abort, 2
qp_abort(), *9*, *17*
qp_as_json *(*qp_scalar*)*, 12
qp_as_json(), *6*, *7*, *13*, *14*
qp_condition_code, 4
qp_condition_code(), *3*
qp_empty_object *(*qp_scalar*)*, 12
qp_empty_object(), *7*, *8*
qp_error_codes, 4
qp_error_codes(), *3*
qp_error_from_response, 5
qp_json_canonical, 6
qp_json_canonical(), *8*, *12*
qp_json_decode *(*qp_json_encode*)*, 7
qp_json_decode(), *5*
qp_json_encode, 7
qp_json_encode(), *6*, *13*, *14*, *16*
qp_msg, 9
qp_msg(), *2*
qp_redact, 10
qp_redact(), *3*, *17*
qp_run_external, 11
qp_scalar, 12
qp_scalar(), *7*, *8*, *10*
qp_schema_assert, 13
qp_schema_assert(), *15*
qp_schema_path, 14
qp_schema_path(), *7*, *13*, *14*
qp_schema_validate, 14
qp_schema_validate(), *8*, *12*
qp_value_fields *(*qp_value_state*)*, 15
qp_value_from_state *(*qp_value_state*)*, 15
qp_value_state, 15
qp_value_state(), *8*
qp_warn, 16
qp_warnings_from_response, 17