Package: shinyOAuth (via r-universe)

October 31, 2026

Title OIDC Authentication and OAuth Authorization for 'shiny' Applications

Version 0.7.0

Description Provides a simple, configurable framework for 'OpenID Connect' (OIDC) authentication and 'OAuth 2.0' authorization in 'shiny' applications using 'S7' classes. Defines providers, clients, and tokens, as well as various supporting functions and a 'shiny' module. Features include cross-site request forgery (CSRF) protection, state encryption, 'Proof Key for Code Exchange' (PKCE) handling, validation of OIDC identity tokens (nonces, signatures, claims), automatic user info retrieval for OIDC and supported 'OAuth' providers, asynchronous flows, and hooks for audit logging.

License MIT + file LICENSE

Encoding UTF-8

Imports S7 (>= 0.2.0), R6 (>= 2.0), rlang (>= 1.0.0), shiny (>= 1.7.0), jsonlite (>= 1.6), openssl (>= 2.1.2), httr2 (>= 1.1.0), curl, urltools (>= 1.7.3), cachem (>= 1.1.0), callr (>= 3.7.6), jose (>= 1.2.0), lifecycle (>= 1.0.0), cli (>= 3.0.0), htmltools (>= 0.5.0), otel (>= 0.2.0)

Suggests bslib, ggplot2, purrr, dplyr, testthat (>= 3.2.2), DT, knitr, rmarkdown, webfakes, promises (>= 1.3.3), mirai (>= 2.5.1), future, withr, later, processx, pkgload, chromote, httpuv, sodium, shinytest2, xml2, otelsdk

Depends R (>= 4.1.0)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://github.com/lukakoning/shinyOAuth, https://lukakoning.github.io/shinyOAuth/

BugReports https://github.com/lukakoning/shinyOAuth/issues

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Luka Koning [aut, cre, cph]

Maintainer Luka Koning <koningluka@gmail.com>

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

Date/Publication 2026-10-31 17:50:28 UTC

RemoteUrl https://github.com/cran/shinyOAuth

RemoteRef HEAD

RemoteSha 22e291c4522ee97dd5c08a2c8171560a041213b3

Contents

check_oauth21 . . . 3
client_bearer_req . . . 5
custom_cache . . . 6
get_userinfo . . . 10
handle_callback . . . 11
introspect_token . . . 13
is_ok_host . . . 15
oauth_browser_owner . . . 16
oauth_client . . . 19
oauth_client_mtls_registration . . . 30
oauth_client_secret_apple . . . 32
oauth_connection . . . 33
oauth_connection_store_memory . . . 35
oauth_connections . . . 37
oauth_connections_server . . . 40
oauth_connections_ui . . . 43
oauth_form_post_ui . . . 45
oauth_module_server . . . 48
oauth_provider . . . 57
oauth_provider_apple . . . 67
oauth_provider_auth0 . . . 68
oauth_provider_authentik . . . 69
oauth_provider_cognito . . . 70
oauth_provider_github . . . 71
oauth_provider_gitlab . . . 72
oauth_provider_globus . . . 73
oauth_provider_google . . . 74
oauth_provider_huggingface . . . 74
oauth_provider_keycloak . . . 75
oauth_provider_microsoft . . . 76
oauth_provider_oidc . . . 80
oauth_provider_oidc_discover . . . 82
oauth_provider_okta . . . 86
oauth_provider_orcid . . . 87

oauth_provider_slack . . . 88
oauth_provider_spotify . . . 88
oauth_provider_surfconext . . . 89
oauth_token_target . . . 90
oauth_ui . . . 92
OAuthClient . . . 93
OAuthConnection . . . 106
OAuthProvider . . . 113
OAuthToken . . . 123
perform_client_bearer_req . . . 126
perform_resource_req . . . 128
prepare_authorization_request . . . 131
prepare_call . . . 132
refresh_token . . . 134
resource_req . . . 136
revoke_token . . . 139
smart_client . . . 141
smart_context . . . 144
smart_discover . . . 146
smart_launch_route . . . 147
use_shinyOAuth . . . 149

Index **151**

`check_oauth21` *Review an OAuth client configuration before deployment*

Description

Check a configured OAuthClient for missing settings and recommendations. The result includes a table of findings and suggested fixes.

Usage

```
check_oauth21(config, context = list(), draft = "draft-ietf-oauth-v2-1-16")
```

Arguments

`config`	An OAuthClient or OAuthProvider. A provider-only report is partial because it cannot check your client registration settings.
`context`	Optional list. `operations` selects additional `"userinfo"`, `"introspection"` or `"revocation"` checks. The advanced `nonce_exception` declaration is described below and is unset by default.
`draft`	Target specification revision. Currently only `"draft-ietf-oauth-v2-1-16"` is supported; this is an Internet-Draft, not an RFC.

Details

The returned `checks` table contains individual findings. `configuration_compliant` is `FALSE` for a known mandatory failure, `NA` for unresolved mandatory settings, and `TRUE` when the applicable mandatory configuration checks pass. Recommendations and unknown external behavior do not change that result.

This assessment uses OAuth 2.1 draft 16, ruleset `1.1.1`. It is not certification or a test of the running provider. It assesses the supplied configuration.

Value

A `shinyOAuth_oauth21_assessment` list. `checks` is a data frame with `id`, `scope`, `status`, `requirement`, `message`, `remediation`, `reference`, `evidence_source`, `requirement_source` and `affects_verdict` columns. Statuses are `pass`, `fail`, `warn`, `unknown` and `not_applicable`. Only rows with `affects_verdict = TRUE` affect `configuration_compliant`.

The report also records `draft`, `ruleset_version`, `package_version`, `assessed_at`, `assessment_scope` and `operations`.

Advanced assessment scope

Checks cover code exchange, refresh, enabled PAR, required UserInfo and introspection, selected extra operations and applicable signing-key retrieval. Optional DPoP, mTLS, PAR, JAR and JARM are not required together. Provider enforcement, registered redirects, secret storage, browser/proxy TLS and future API requests need separate deployment checks.

The checker does not open mTLS files, so applicable `mtls.credentials.*` checks report `unknown` when paths are present; missing required paths fail.

`requirement_source` distinguishes protocol requirements from security guidance and package policy. Callback capacity thresholds are package recommendations, not numeric limits from the draft. Reports exclude credentials, keys, endpoint URLs and the configuration objects. Cache and state-store methods are not run.

Advanced PKCE exception

`context = list(nonce_exception = TRUE)` declares independently established server assurance for the draft's confidential-client nonce exception (section 7.5.1.1).

References

https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-16

https://datatracker.ietf.org/doc/html/draft-ietf-oauth-rfc7523bis-11

https://www.rfc-editor.org/rfc/rfc9207.html

Examples

```
provider <- oauth_provider(
  name = "Example",
  auth_url = "https://auth.example/authorize",
  token_url = "https://auth.example/token",
```

```
  token_auth_style = "public", use_pkce = TRUE, pkce_method = "S256"
)
client <- oauth_client(
  provider, client_id = "example-client",
  redirect_uri = "https://app.example/callback"
)
assessment <- check_oauth21(client)
assessment[["checks"]][assessment[["checks"]][["status"]] != "pass", ]
```

`client_bearer_req` *Alias for* `resource_req()`

Description

[Deprecated]

Deprecated alias for `resource_req()`. Use `resource_req()` for Bearer, DPoP, and mTLS-protected resource requests instead.

Usage

```
client_bearer_req(
  token,
  url,
  method = "GET",
  headers = NULL,
  query = NULL,
  follow_redirect = FALSE,
  check_url = TRUE,
  client = NULL,
  token_type = NULL,
  dpop_nonce = NULL,
  resource_hosts = NULL,
  oauth_client = NULL
)
```

Arguments

`token`	Either an OAuthToken object or a raw access token string.
`url`	The absolute URL to call. Resource URLs must not exceed 8192 bytes, including the query and fragment, both before and after applying `query`. This limit also applies when `check_url = FALSE`.
`method`	Optional HTTP method (character). Defaults to "GET". When the effective token type is `DPoP`, this must be the final request method because the proof is signed against it. `TRACE` and the nonstandard `TRACK` method are rejected because authenticated requests could be reflected by the server and disclose credentials.

headers	Optional named list or named character vector of extra headers to set on the request. Header names are case-insensitive. Names must be unique HTTP field-name tokens, and values must not contain prohibited control characters. DPoP requests reject Host overrides. This function manages the Authorization and DPoP headers and ignores user-supplied replacements.
query	Optional named list of query parameters to append to the URL.
follow_redirect	Logical or NULL. FALSE (the default) disables HTTP redirects even when shinyOAuth.allow_redirect is enabled. NULL inherits that global option (disabled by default). TRUE enables redirects. DPoP requests reject enabled redirects because each target needs a new proof.
check_url	Logical. If TRUE (the default), validates url against is_ok_host() before attaching the access token. This rejects relative URLs, plain HTTP to non-loopback hosts, and when options(shinyOAuth.allowed_hosts) is set, hosts outside the allowlist. Without an allowlist this performs HTTPS and URL-syntax validation only (with the configured non-HTTPS exceptions); any HTTPS host is accepted. FALSE skips this policy check; URL length and resource-host restrictions still apply.
client	Optional OAuthClient; required for DPoP or mTLS-bound tokens so the request can use the matching signing key or certificate and check any token-binding information.
token_type	Optional override for the access token type when token is supplied as a raw string. Supported values are Bearer and DPoP. Invalid or multi-valued inputs are rejected. When omitted, shinyOAuth preserves OAuthToken@token_type, and may infer DPoP from explicit OAuthToken@cnf[["jkt"]] metadata. Raw access-token strings default to Bearer unless you pass token_type = "DPoP" explicitly.
dpop_nonce	Optional DPoP nonce to embed in the proof for this request. This is primarily useful after a resource server challenges with DPoP-Nonce.
resource_hosts	Optional non-empty character vector of trusted resource host patterns, using is_ok_host() matching rules. This call-scoped allowlist adds to the global policy and is enforced even if check_url is FALSE. It constrains the initial URL, not redirect destinations or resolved IPs. NULL adds no resource-specific policy.
oauth_client	Compatibility alias for client. Supply only one spelling.

Value

Same value as resource_req().

---

custom_cache *Create a custom state store or signing-key cache*

Description

Connect shinyOAuth to a shared storage backend, such as Redis or a database, by wrapping your R functions in a cachem-like interface. Use the result as `state_store` in `oauth_client()` to share pending login state, or as `jwks_cache` in `oauth_provider()` to share provider signing keys (JWKS).

A shared state store is needed when a login can start on one R process and its callback can arrive at another, for example in a multi-worker deployment without sticky routing. With a shared signing-key cache, workers reuse keys fetched by other workers.

Usage

```
custom_cache(
  get,
  set,
  remove,
  take = NULL,
  info = NULL,
  set_if_absent = NULL,
  worker_safe = FALSE
)
```

Arguments

`get`	A function(key, missing = NULL) -> value. Required. Should return the stored value, or the `missing` argument if the key is not present. The `missing` parameter is required because shinyOAuth passes it explicitly.
`set`	A function(key, value) -> invisible(NULL). Required. Should store the value under the given key.
`remove`	A function(key) -> any. Required. Delete the entry for `key`. shinyOAuth does not depend on the return value. State consumption uses atomic `take` where available; a process-local fallback checks that removal succeeded.
`take`	A function(key, missing = NULL) -> value. Optional. Read and remove the entry as one indivisible operation, returning its value or the `missing` argument when absent. Required for shared state stores: separate reads and deletes can let two requests use the same login record. Redis `GETDEL` and SQL `DELETE ... RETURNING` are examples. The process-local memory store has a safe fallback; custom/shared stores without `take` fail by default when login state is consumed.
`info`	Function() -> list(max_age = seconds, ...). Optional `oauth_module_server()` uses the returned `max_age` to align the browser cookie's maximum age with the store's retention period.
`set_if_absent`	A function(key, value, ttl = NULL) -> logical. Optional. An atomic set-if-missing operation for shared JWKS caches and callback bridge slots. It must store `value` and return `TRUE` only when `key` did not already exist; otherwise it must leave the existing value unchanged and return `FALSE`. When

`ttl` is supplied, the claimed key must expire after that many seconds. Map this to a native backend primitive such as Redis `SET ... NX EX` or a database uniqueness constraint with expiry. shinyOAuth uses this operation to make forced JWKS-refresh throttling safe across workers. Without it, forced refresh is disabled for shared/custom caches; `cachem::cache_mem()` keeps its process-local serialized fallback. Callback bridges require it for custom/shared stores to avoid overwriting concurrent candidates; full partitions reject callbacks until slots expire or are consumed. The process-local memory store uses a serialized fallback. The existing `shinyOAuth.allow_non_atomic_state_store = TRUE` option also permits non-atomic callback insertion, with no protection against concurrent writers replacing a candidate. Other state-store writes still use `set`.

`worker_safe` Logical, default `FALSE`. Explicitly declares that this backend can be serialized and used in another R process. Applies when the cache is used as `jwks_cache`; see the worker-safety requirements above.

Details

Backend functions must preserve stored R values and expire entries after their configured lifetime.

Value

An R6 object exposing cachem-like `[["get"]]`/`[["set"]]`/`[["remove"]]`/`[["info"]]` methods and the optional `[["take"]]` and `[["set_if_absent"]]` atomic methods.

Shared login state in multi-worker deployments

The default `cachem::cache_mem()` state store belongs to one R process. If a load balancer sends the returning callback to another worker, that worker cannot find the pending login and validation fails. Shared storage allows either worker to resume it, provided both use the same `state_key` and provider/client settings.

A shared `state_store` requires atomic `take`: reading and deleting a pending login in one operation so two requests cannot use it. Redis `GETDEL` and SQL `DELETE ... RETURNING` provide this operation. Separate reads and deletes, including those in `cachem::cache_disk()`, can let two workers consume the same state. See the deployment guidance. For the `form_post` and `form_post.jwt` callback bridges, also implement `set_if_absent` to reserve callback slots without replacing pending responses.

The backend preserves stored R values and enforces access controls, expiry and atomic consumption. Pending login records are encrypted. An upgrade can invalidate pending records; affected logins then restart.

Hosted Request Objects (`request_object_mode = "request_uri"`) use separate records. Signed objects can contain readable authorization details; configure Request Object encryption if these details must be confidential in storage.

For a state store, returning `max_age` in seconds from `info()` also lets `oauth_module_server()` align the browser cookie lifetime with the store. Reporting this value does not expire entries; your backend must enforce it.

Shared provider signing keys

A shared `jwks_cache` can reduce repeated key downloads when several R workers use the same provider. This is independent of sharing login state: sharing signing keys alone does not let another worker resume a login.

Key caching uses `get` and `set`. Also implement `set_if_absent` to coordinate rate-limited forced key refreshes across workers, for example when the provider rotates its signing keys. Without that atomic operation, forced refresh is disabled for custom/shared caches. Use separate stores or key namespaces for login state and signing keys when they require different expiry policies.

A custom signing-key cache defaults to synchronous fallback for asynchronous login and token operations. `worker_safe = TRUE` declares that its functions and captured values remain usable in a new R process. That requires per-operation connections rather than captured database/file handles; serialization alone cannot establish worker safety. Standard `cachem::cache_mem()` and `cachem::cache_disk()` backends support worker dispatch automatically. Other cache implementations can expose a scalar logical `worker_safe` member with the same contract. State stores are not used in callback workers.

Examples

```
# This in-memory example illustrates the cache interface in one R process.
# It does not share entries across workers or implement timed expiry.
# A production shared store must implement both itself.
mem <- new.env(parent = emptyenv())

my_cache <- custom_cache(
  get = function(key, missing = NULL) {
    base::get0(key, envir = mem, ifnotfound = missing, inherits = FALSE)
  },

  set = function(key, value) {
    assign(key, value, envir = mem)
    invisible(NULL)
  },

  remove = function(key) {
    if (exists(key, envir = mem, inherits = FALSE)) {
      rm(list = key, envir = mem)
    }
    invisible(NULL)
  },

  # In a shared store, replace this with the backend's atomic get-and-delete
  # operation, such as Redis GETDEL. This R environment is process-local.
  take = function(key, missing = NULL) {
    val <- base::get0(key, envir = mem, ifnotfound = missing, inherits = FALSE)
    if (exists(key, envir = mem, inherits = FALSE)) {
      rm(list = key, envir = mem)
    }
    val
  },
```

```
  info = function() list(max_age = Inf)
)
```

`get_userinfo` *Fetch a user's profile (UserInfo)*

Description

Retrieve profile information using the user's access token. Call this when fetching a profile on demand, reloading profile fields, or managing tokens outside the Shiny module. It returns the provider's profile as an R list. The Shiny module fetches and stores this information during login when `userinfo_required = TRUE`; that result is available as `auth[["token"]]@userinfo`.

Usage

```
get_userinfo(
  client,
  token,
  token_type = NULL,
  shiny_session = NULL,
  oauth_client = NULL
)
```

Arguments

`client`	OAuthClient object. The client must have a `userinfo_url` configured in its OAuthProvider.
`token`	Either an OAuthToken object or a raw access token string.
`token_type`	Optional override for the access token type when `token` is provided as a raw string. Supported values are `Bearer` and `DPoP`.
`shiny_session`	Optional captured Shiny session details for audit events, normally supplied by the module.
`oauth_client`	Compatibility alias for `client`. Supply only one spelling.

Details

Pass the client and complete OAuthToken from login, for example `get_userinfo(client, auth[["token"]])`. The provider needs a `userinfo_url`. For OIDC, shinyOAuth checks that the returned profile belongs to the validated login. A raw access-token string cannot provide that identity check.

`userinfo_required = FALSE` disables automatic profile retrieval.

Value

A list containing the user information returned by the provider. For JSON responses, arrays are simplified to vectors or data frames where possible. Signed JWT responses retain arrays as lists.

Advanced response validation

Normal JSON profiles and signed JWT profiles are supported; encrypted UserInfo is not. `userinfo_signed_jwt_required` requires a signed response using `userinfo_allowed_algs`. `userinfo_jwt_required_time_claims` can require expiry and other time claims; present time claims are always checked.

By default, OIDC providers with a UserInfo endpoint require a validated identity token, including for manual retrieval. Explicit `userinfo_id_token_match = FALSE` allows retrieval without one, but when both are available their `sub` values must always match. Standard profile fields are checked for the expected JSON types; extension fields retain their provider-defined values.

For DPoP or mTLS-bound tokens, the client supplies the matching signing key or certificate. See the advanced security vignette for those configurations.

Examples

```
# get_userinfo(), introspect_token(), and refresh_token() are typically
# called by oauth_module_server() according to your provider/client and
# module settings, rather than directly by application code. The module
# also calls revoke_token() during logout when the provider supports it.
# These helpers are exported for custom login flows, on-demand profile or
# token checks, and applications that manage token lifetime themselves.
#
# The examples below require a real token from a completed login.
# Inside a reactive expression in server(), after creating auth with
# oauth_module_server() and confirming auth[["authenticated"]]:
if (interactive()) {
  token <- auth[["token"]]
  user_info <- get_userinfo(client, token)

  # Requires an introspection endpoint. NA means activity is unknown.
  result <- introspect_token(client, token)
  isTRUE(result[["active"]])

  # Returns replacement credentials using the refresh token.
  token <- refresh_token(client, token)

  # Requires a revocation endpoint to invalidate the token at the provider.
  result <- revoke_token(client, token, token_kind = "refresh")
}
```

---

handle_callback *Validate an OAuth 2.0 callback and retrieve tokens*

Description

Check a returning login request and exchange the provider's temporary authorization code for an OAuthToken (OAuth 2.0 Authorization Code flow). Use this in a custom callback handler after starting authorization with `prepare_call()`. It applies shinyOAuth's state, token, and configured identity checks while your application manages the HTTP callback and stores the returned token. `oauth_module_server()` handles these responsibilities for Shiny sessions.

Usage

```
handle_callback(
  client,
  code,
  state,
  browser_token,
  shiny_session = NULL,
  iss = NULL,
  oauth_client = NULL,
  payload = NULL
)
```

Arguments

client	An OAuthClient object.
code	Authorization code received from the provider on a classic direct callback.
state	Encrypted state payload returned by the provider on a classic direct callback, unchanged from the value sent by prepare_call().
browser_token	Browser token present in the user's session. This is usually managed by oauth_module_server().
shiny_session	Optional captured Shiny session details for audit events, normally supplied by the module.
iss	Optional RFC 9207 callback issuer (iss) from the authorization response. Pass this when one callback URL can receive responses from more than one authorization server. If client@enforce_callback_issuer is TRUE, this parameter is required and must match the configured provider issuer before any token exchange occurs. This low-level API cannot verify which redirect URI received the response. Clients configured with authorization_server_mode = "multi_redirect_uri" must use oauth_module_server() instead.
oauth_client	Compatibility alias for client. Supply only one spelling.
payload	Compatibility alias for state. Supply only one spelling.

Details

Pass the returned code, the callback's state, and the browser token saved for this login. This helper accepts direct code/state callbacks only. For signed responses using JWT Secured Authorization Response Mode (JARM; "jwt", "query.jwt", or "form_post.jwt"), use oauth_module_server() and, for POST responses, oauth_form_post_ui(). There is no public JARM resume API.

Value

An OAuthToken object. If callback validation, token exchange, or token verification fails, the function raises an error.

Examples

```
# Advanced example: your code supplies browser redirects and callback handling.
# For a Shiny app, oauth_module_server() manages these steps for you.

if (interactive()) {
  # Define client
  client <- oauth_client(
    provider = oauth_provider_github(),
    client_id = Sys.getenv("GITHUB_OAUTH_CLIENT_ID"),
    client_secret = Sys.getenv("GITHUB_OAUTH_CLIENT_SECRET"),
    redirect_uri = "http://127.0.0.1:8100"
  )

  # Get the login URL and store state in client's state store
  # `<browser_token>` must be unpredictable and persisted for this transaction
  # in storage bound to the application's exact origin (scheme, host, port).
  # The module combines origin-scoped storage with an independent marker cookie
  # and checks both on return. A cookie alone does not provide this boundary:
  # cookies can be shared by applications on different ports of the same host.
  # Shiny applications should use oauth_module_server() for the complete flow.
  authorization_url <- prepare_call(client, "<browser_token>")

  # Redirect user to authorization URL; retrieve code & state from the query;
  # recover this transaction's `<browser_token>` through the origin-bound flow
  # and verify its independent marker before calling handle_callback().
  code <- "..."
  state <- "..."
  browser_token <- "..."

  # Handle callback, exchanging code for token and validating state
  token <- handle_callback(client, code, state, browser_token)
}
```

`introspect_token` *Introspect an OAuth 2.0 token*

Description

Ask the provider to check an access or refresh token. This is called token introspection and requires a configured `introspection_url`. A token may be revoked before its expiry; introspection checks its current status. To require it automatically during login and refresh, set `introspect = TRUE` on `oauth_client()`.

Usage

```
introspect_token(
  client,
  token,
  token_kind = c("access", "refresh"),
```

```
  async = FALSE,
  shiny_session = NULL,
  oauth_client = NULL,
  oauth_token = NULL,
  which = NULL
)
```

Arguments

client	OAuthClient object
token	OAuthToken object to introspect
token_kind	Which token to introspect: "access" (default) or "refresh".
async	If TRUE, return a promise resolving to the result, using configured mirai daemons or a future plan; mirai takes priority. A non-sequential backend moves work outside the main R process. Default FALSE waits and returns the result directly.
shiny_session	Optional captured Shiny session details for audit events, normally supplied by the module.
oauth_client	Compatibility alias for client. Supply only one spelling.
oauth_token	Compatibility alias for token. Supply only one spelling.
which	Compatibility alias for token_kind. Supply only one spelling.

Details

result[["active"]] is TRUE for an active token, FALSE for an inactive token, and NA when the result is unknown.

Unsupported endpoints, missing tokens, unsuccessful HTTP responses, and unusable response bodies return a descriptive status. The provider must return active as a JSON boolean. Other types return "invalid_active". Encoded or decoded body limits return "body_too_large"; unsupported compression returns "unsupported_encoding". Both leave active = NA. Other transport failures and decoding errors raise conditions (or reject the asynchronous promise) instead of returning a status result. Requests use the client's configured credentials and token_auth_style.

Value

A list with fields:

- supported: logical, TRUE when an introspection endpoint is configured.
- active: logical or NA, where NA means the provider did not return a usable RFC 7662 active value.
- raw: parsed introspection response list, or NULL when the endpoint is unsupported or the response could not be parsed.
- status: machine-readable status such as "ok", "introspection_unsupported", "missing_token", "invalid_json", "missing_active", "invalid_active", or "http_<code>".

Examples

```
# get_userinfo(), introspect_token(), and refresh_token() are typically
# called by oauth_module_server() according to your provider/client and
# module settings, rather than directly by application code. The module
# also calls revoke_token() during logout when the provider supports it.
# These helpers are exported for custom login flows, on-demand profile or
# token checks, and applications that manage token lifetime themselves.
#
# The examples below require a real token from a completed login.
# Inside a reactive expression in server(), after creating auth with
# oauth_module_server() and confirming auth[["authenticated"]]:
if (interactive()) {
  token <- auth[["token"]]
  user_info <- get_userinfo(client, token)

  # Requires an introspection endpoint. NA means activity is unknown.
  result <- introspect_token(client, token)
  isTRUE(result[["active"]])

  # Returns replacement credentials using the refresh token.
  token <- refresh_token(client, token)

  # Requires a revocation endpoint to invalidate the token at the provider.
  result <- revoke_token(client, token, token_kind = "refresh")
}
```

`is_ok_host` *Check a URL against the package's host policy*

Description

Test whether a URL is allowed by shinyOAuth's ordinary scheme and host rules. HTTPS is accepted by default. HTTP is limited to local development hosts unless you change `allowed_non_https_hosts`. Supply `allowed_hosts` to restrict which services your app may contact.

Provider and API request helpers apply these checks internally. A direct call reports the policy result without making a network request.

Usage

```
is_ok_host(
  url,
 allowed_non_https_hosts = getOption("shinyOAuth.allowed_non_https_hosts", default =
    c("localhost", "127.0.0.1", "::1", "[::1]")),
  allowed_hosts = getOption("shinyOAuth.allowed_hosts", default = NULL)
)
```

Arguments

`url`	Single URL or vector of URLs (character; length 1 or more)
`allowed_non_https_hosts`	Character vector of hostnames that are allowed to use HTTP instead of HTTPS. Defaults to localhost equivalents. Supports globs
`allowed_hosts`	Optional allowlist of hosts/domains; if supplied (length > 0), only these hosts are permitted. Supports globs

Details

Both host lists support `*` (any characters), `?` (one character), and a leading dot: `".example.com"` matches the domain and its subdomains. `"*"` permits every host. If `allowed_hosts` is empty, only the scheme rules apply. Missing values, empty strings, and malformed URLs return `FALSE`.

If the scheme is absent, this helper tries HTTP, then HTTPS. Request helpers can impose additional requirements, including an absolute URL. OIDC discovery has a separate HTTPS policy and requires an explicit loopback development opt-in; a `TRUE` result here does not override it.

Defaults come from `shinyOAuth.allowed_hosts` and `shinyOAuth.allowed_non_https_hosts`.

Value

Logical indicator (TRUE if all URLs pass all checks; FALSE otherwise)

Examples

```
# HTTPS allowed by default
is_ok_host("https://example.com")

# HTTP allowed for localhost
is_ok_host("http://localhost:8100")

# Restrict to a specific domain (allowlist)
is_ok_host("https://api.example.com", allowed_hosts = c(".example.com"))

# A catch-all pattern allows any host.
is_ok_host("https://anywhere.example", allowed_hosts = c("*"))
```

`oauth_browser_owner` *Configure ownership of retained OAuth connections*

Description

Defines ownership of saved connections for `oauth_connections()`: either the browser that created them or an account authenticated by the application. The manager applies the policy when its matching retention mode is enabled; these functions create configuration only.

Usage

```
oauth_browser_owner(
  idle_timeout = 1800,
  absolute_timeout = 28800,
  same_site = c("Lax", "Strict"),
  allow_http_loopback = FALSE,
  max_entries = 1000L
)

oauth_account_owner(
  resolver,
  idle_timeout,
  absolute_timeout,
  reauth_after_seconds,
  max_entries = 1000L
)

## S3 method for class 'OAuthOwnerPolicy'
print(x, ...)
```

Arguments

idle_timeout	Maximum owner inactivity in seconds. Resource and status reads do not count as activity. The manager's touch() records explicit activity; connecting, explicit refresh and disconnect also count.
absolute_timeout	Maximum owner lifetime in seconds, independent of activity. Must be at least idle_timeout.
same_site	Owner-cookie policy, "Lax" for top-level authorization navigation or "Strict". Embedded cross-site ownership is not supported. With "Strict", oauth_connections_ui() may show a brief intermediate page when returning from another site, so the browser can restore the saved session. See Deployment details below if a proxy changes browser headers.
allow_http_loopback	Explicit development-only exception for HTTP on localhost or a loopback address. Default FALSE requires HTTPS. The exception cannot provide a Secure, host-prefixed owner cookie.
max_entries	Maximum owner-registry entries per manager, a positive whole number. Browser mode counts browsers that authorize a service. A separate provisional visitor pool has the same limit; its oldest entries may be replaced and expire after at most five minutes, unless a live authorization transaction protects them until its expiry. When all visitor entries have pending authorizations, new visitors are rejected. Account mode counts authentication generations, including retired generations until their local reauthentication deadline. This limit is independent of the connection store's max_entries.
resolver	Trusted application function accepting the current Shiny session. On every call it must validate the application's local login and return NULL if unauthen-

ticated, otherwise a plain list containing `subject`, `session_id`, `generation`, `authenticated_at` and `expires_at`. The first three are non-empty strings; the timestamps are finite Unix seconds. Subject is the stable local account ID. Session ID and generation identify the current local authentication session, as established by the application's verified local login rather than the external OAuth authorization.

`reauth_after_seconds`
: Maximum age of the verified local authentication, in seconds. Required for account retention; refresh cannot reset this age.

`x`
: An `OAuthOwnerPolicy` to print.

`...`
: Unused print arguments.

Details

Browser retention identifies an authorized browser session, not a verified person. The owner cookie is HttpOnly, host-only, has root path and uses Secure and a `__Host-` name on HTTPS. It contains no OAuth token or patient data. Server-side idle and absolute limits are authoritative. A cookie is never accepted as an owner without a matching live server record for this application origin.

Account retention requires finite idle, absolute and local reauthentication lifetimes. The internal session registry re-runs `resolver` when resolving or validating an owner and checks the intended subject/session generation. Expiry or logout retires that generation until a fresh local authentication session is supplied. A resolver error fails closed with a redacted error. Local-session validation belongs to the application's resolver, including signature, expiry, revocation and account changes.

Value

An `OAuthOwnerPolicy` configuration object. Browser policies use an opaque server-issued cookie and server-side owner registry. Account policies use the trusted local-session resolver described below.

Deployment details

Browser cookies are shared across ports on a hostname, so another HTTPS service receiving the owner cookie could reuse it. Cookie names do not establish separate trust boundaries for those services. With `same_site = "Strict"`, external-entry recovery needs same-origin Fetch Metadata, `Referer` or `Origin` headers; a proxy must preserve that information.

Cookie rotation invalidates the previous session generation immediately and preserves the original absolute lifetime. Local logout removes the live owner session. The manager checks that generation before code exchange and credential commit, with no grace period for pending authorization, and handles credential cleanup after logout. An external provider login cannot establish a local owner or implicitly link browser connections to an account.

A full owner registry rejects new retained owners without evicting live sessions or retirement records. Browser session end does not release ownership; logout or idle/absolute expiry does. Account logout retains its retired generation until `authenticated_at + reauth_after_seconds` so it cannot be enrolled again. Retired generations count toward `max_entries` throughout that window.

See Also

oauth_connection_store_memory()

Examples

```
# Browser connections expire after 15 minutes idle or 8 hours in total.
oauth_browser_owner(idle_timeout = 15 * 60, absolute_timeout = 8 * 3600)

# Supply your application's trusted local-login validator for account retention.
# It must revalidate the session on each call and return the documented fields.
account_policy <- function(validate_local_session) {
  oauth_account_owner(
    resolver = validate_local_session,
    idle_timeout = 15 * 60,
    absolute_timeout = 8 * 3600,
    reauth_after_seconds = 8 * 3600
  )
}
```

---

oauth_client *Configure OAuth/OIDC client credentials and login settings*

Description

Create a client with the credentials assigned by your provider, the URL where users return after login, and the permissions your app needs. Pass the result to oauth_module_server().

Usage

```
oauth_client(
  provider,
  client_id,
  client_secret = character(0),
  redirect_uri,
  enforce_callback_issuer = NULL,
  scopes = token_target_default_scopes(token_targets, provider),
  resource = character(0),
  claims = NULL,
  state_store = cachem::cache_mem(max_age = 300),
  state_payload_max_age = 300,
  state_entropy = 64,
  state_key = random_urlsafe(128),
  client_assertion_private_key = NULL,
  client_assertion_private_key_kid = NULL,
  client_assertion_alg = NULL,
  client_assertion_audience = NULL,
  mtls_client_cert_file = NULL,
  mtls_client_key_file = NULL,
```

```
  mtls_client_key_password = NULL,
  mtls_client_ca_file = NULL,
  mtls_certificate_bound_access_tokens = FALSE,
  request_object_mode = c("parameters", "request", "request_uri"),
  response_mode = NULL,
  request_object_signing_alg = NULL,
  request_object_audience = NULL,
  request_object_encryption_alg = NULL,
  request_object_encryption_enc = NULL,
  request_object_encryption_kid = NULL,
  request_object_ttl = 45,
  request_object_nbf_skew = NULL,
  dpop_private_key = NULL,
  dpop_private_key_kid = NULL,
  dpop_signing_alg = NULL,
  dpop_require_access_token = NULL,
  scope_validation = c("warn", "strict", "none"),
  claims_validation = c("none", "warn", "strict"),
  userinfo_jwt_required_time_claims = character(0),
  required_acr_values = character(0),
  introspect = FALSE,
  introspection_checks = character(0),
  authorization_server_mode = c("single", "multi_issuer", "multi_redirect_uri"),
  authorization_server_redirect_uris = character(0),
  dpop_require_observed_cnf = FALSE,
  jarm_signed_response_alg = NULL,
  jarm_encrypted_response_alg = NULL,
  jarm_encrypted_response_enc = NULL,
  jarm_decryption_private_key = NULL,
  jarm_decryption_private_key_kid = NULL,
  jarm_max_lifetime = 600,
  endpoint_auth = list(),
  mtls_require_observed_cnf = TRUE,
  trusted_id_token_audiences = character(0),
  compare_callback_issuer = NULL,
  client_assertion_typ = "JWT",
  authorization_method = "GET",
  resource_bases = character(),
  required_scopes = character(),
  label = default_client_label(provider),
  ...,
  introspect_elements = NULL,
  token_targets = list(),
  token_target_default = if (length(token_targets) == 1L) names(token_targets) else
    character()
)
```

Arguments

provider	The service configuration, created with a provider helper such as oauth_provider_google() or oauth_provider_oidc_discover().
client_id	The identifier assigned when you register your app with the provider.
client_secret	The secret issued for your app, preferably read with Sys.getenv(). Omit it for registrations that do not use a secret. It is required for token_auth_style = "header". With "body" and PKCE, an empty secret is omitted. With "public" (alias "none"), it is never sent for client authentication. HMAC-signed ID token validation still requires a non-empty secret, regardless of the client authentication method.
redirect_uri	The URL where users return after login. It must match the callback URL registered with your provider, including scheme, host, port, and path. Use HTTPS in production.
enforce_callback_issuer	Logical or NULL. When TRUE, enforce that authorization responses handled through this client include an RFC 9207 iss parameter and reject callbacks unless it exactly matches provider@issuer. This is recommended when one callback URL can receive responses from more than one authorization server. Requires the provider to have a configured issuer. When NULL (the oauth_client() helper default), shinyOAuth auto-enables this check for providers that advertise authorization_response_iss_parameter_supported = TRUE and have a configured issuer, such as OIDC discovery providers that expose RFC 9207 support. Set FALSE to opt out explicitly.
scopes	Character vector of permissions to request. The provider defines the available names. For OIDC (issuer set and infer_oidc_from_issuer = TRUE), shinyOAuth adds "openid" automatically if absent. The resulting set is used in the request and subsequent scope checks. When omitted with token_targets, API scopes are inferred from those declarations. Explicit scopes must include every target's API scopes; they are never expanded from declarations. With no targets, the default requests no API scopes. OIDC adds "openid"; request further identity and refresh-consent scopes explicitly.
resource	Optional RFC 8707 resource indicator(s). Supply a character vector of absolute URIs to request audience-restricted tokens for one or more protected resources. Each value is sent as a repeated resource parameter on the authorization request, initial token exchange, and token refresh requests. Default is character(0).
claims	Optional request for specific OIDC user information, beyond scopes. Default NULL sends no request. Supply a list with userinfo and/or id_token members, for example list(userinfo = list(email = list(essential = TRUE))). Use claims_validation = "strict" if an unmet request must stop login. Use NULL to request a claim without a constraint, value for one acceptable value, or values for several. Lists are converted to JSON for the provider; a pre-encoded JSON object is also accepted. The provider must support claims requests. See Claims requests below for validation and numeric limits.
state_store	Storage for pending logins. The default cachem::cache_mem(max_age = 300) is suitable for one R process. For multiple app processes, supply a shared

custom_cache() with atomic [["take"]]() and use the same state_key on every process. Plain cachem::cache_disk() is unsafe for shared login state because its separate read and delete operations do not prevent simultaneous reuse. See custom_cache() for method and stored-value requirements.

state_payload_max_age
: Maximum age of a pending login's encrypted state, in seconds. Default 300. This is checked separately from the state store's entry lifetime; both must allow the returning login.

state_entropy
: Length in characters of the random state identifier, from 22 to 128. Default 64.

state_key
: Secret used to encrypt and protect pending login details. A random key is generated when omitted. This is separate from client_secret and is also used for public clients.

  Deployments with multiple R processes use the same key and state_store across processes. The key is a character string or raw vector containing at least 32 cryptographically random bytes. State uses AES-GCM authenticated encryption.

client_assertion_private_key
: Optional private key for private_key_jwt client authentication at the token endpoint. Can be an openssl::key or a PEM string containing a private key. Required when the provider's token_auth_style = 'private_key_jwt'. Also used to sign JAR Request Objects, regardless of the token auth style. Current outbound private-key JWT signing supports RSA, EC, and Ed25519 private keys. RSA keys support RS256 and explicitly selected RS384; RS512 and RSA-PSS (PS256, PS384, PS512) are not supported. Ed25519 keys support Ed25519 (RFC 9864) and legacy EdDSA (the default for compatibility); Ed448 is not supported.

client_assertion_private_key_kid
: Optional key identifier (kid) to include in the JWT header for private_key_jwt assertions and JAR Request Objects. Useful when the authorization server uses kid to select the correct verification key.

client_assertion_alg
: Optional JWT signing algorithm to use for client assertions. When omitted, defaults to HS256 for client_secret_jwt. For private_key_jwt, a compatible default is selected based on the private key type/curve (e.g., RS256 for RSA or ES256/ES384/ES512 for EC P-256/384/521, or EdDSA for Ed25519). If an explicit value is provided but incompatible with the key, validation fails early with a configuration error. When the provider advertises token_endpoint_auth_signing_alg_values_supp both explicit values and inferred defaults must be included in that set. Supported values are HS256, HS384, HS512 for client_secret_jwt and asymmetric algorithms supported for outbound signing (RS256, RS384, ES256, ES384, ES512, and Ed25519 or legacy EdDSA with Ed25519 keys) for private keys. RS512, PS256, PS384, and PS512 are not currently supported for outbound client assertions. Reusing an RSA key or HMAC secret requires the same signing algorithm across enabled profiles and endpoint overrides. Use separate keys or secrets when algorithms differ; see the advanced security vignette.

client_assertion_audience
: Optional override for the aud claim used when building JWT client assertions (client_secret_jwt / private_key_jwt). By default, shinyOAuth uses the

active token, introspection, or revocation request URL. PAR uses the issuer when configured, otherwise the canonical PAR URL, including when the request uses an mTLS alias. Set an explicit value when required by the provider's registration agreement.

`mtls_client_cert_file`
: Optional path to the PEM-encoded client certificate (or certificate chain) used for RFC 8705 mutual TLS (mTLS) client authentication and certificate-bound protected-resource requests. Required when `provider@token_auth_style` is `"tls_client_auth"` or `"self_signed_tls_client_auth"`. The certificate matching the private key must appear first, followed by its issuers in chain order. CA-first bundles are rejected.

`mtls_client_key_file`
: Optional path to the PEM-encoded private key used with `mtls_client_cert_file`. Must be supplied together with `mtls_client_cert_file`, and is required for RFC 8705 mTLS client authentication.

`mtls_client_key_password`
: Optional password used to decrypt an encrypted PEM private key referenced by `mtls_client_key_file`.

`mtls_client_ca_file`
: Optional path to a PEM CA bundle used to validate the remote HTTPS server certificate when making mTLS requests. This is mainly useful for local or test environments that use self-signed server certificates.

`mtls_certificate_bound_access_tokens`
: Logical. Whether this client intends to request RFC 8705 certificate-bound access tokens when the provider advertises that capability. Default is `FALSE`.

  When `TRUE`, authorization-server requests prefer discovered `mtls_endpoint_aliases`, even when `token_auth_style` is not an mTLS auth style. Token and protected-resource requests present the certificate.

  Requires `mtls_client_cert_file` and `mtls_client_key_file`, and the provider must be configured with `mtls_client_certificate_bound_access_tokens = TRUE`. By default, `mtls_require_observed_cnf = TRUE` also requires locally observable confirmation of the certificate binding. For opaque tokens whose binding is enforced only by the servers, keep `mtls_certificate_bound_access_tokens = TRUE` and set `mtls_require_observed_cnf = FALSE`.

`request_object_mode`
: Controls how the authorization request is transported to the provider.

  - `"parameters"` (default): send OAuth parameters directly on the browser redirect URL.
  - `"request"`: send a signed JWT-secured authorization request (JAR; RFC 9101) via the `request` parameter.
  - `"request_uri"`: publish a signed Request Object by reference and send its URL via the `request_uri` parameter.

  If the provider has a `par_url`, `"parameters"` and `"request"` are sent to that endpoint first using Pushed Authorization Requests (PAR). The browser then receives the provider-issued `request_uri` handle. Caller-published `"request_uri"` mode is separate from PAR and cannot be used when the provider requires PAR.

Use a signed Request Object when the provider requires JAR or when it must verify the integrity of the authorization parameters. `"request_uri"` lets the provider fetch the object from a published URL instead of carrying the JWT in the browser redirect. Both modes require signing material on the client. shinyOAuth prefers `client_assertion_private_key` when present; otherwise it falls back to HMAC signing with `client_secret`. When Request Object encryption is configured, shinyOAuth signs first and then wraps the signed Request Object in a JWE. Caller-managed `request_uri` publication requires HTTPS; HTTP URLs are rejected even when another configured host policy would otherwise allow them, as required by RFC 9101 Section 5.2. If the provider advertises `request_uri_registration_required = TRUE`, caller-managed `request_uri` publication still depends on the provider having that URI or a matching wildcard prefix registered for the client; shinyOAuth cannot verify that server-side registration automatically.

`response_mode`
: How the provider returns the login result. `NULL` (default) omits the parameter, using the provider's normal query callback. `"query"` explicitly requests URL parameters; `"form_post"` requests an HTTP POST. POST callbacks require `oauth_form_post_ui()`.

  Signed responses (JWT Secured Authorization Response Mode, JARM) use `"jwt"`, `"query.jwt"`, or `"form_post.jwt"` and require `oauth_module_server()`. `"jwt"` uses the query transport for this authorization-code flow. `"form_post.jwt"` also needs `oauth_form_post_ui()`. `handle_callback()` does not handle JARM. Requested modes must be in `response_modes_supported` when advertised; fragment modes are not supported.

`request_object_signing_alg`
: Optional JWS algorithm override for signed authorization requests when `request_object_mode` uses a Request Object (`"request"` or `"request_uri"`). When omitted, shinyOAuth chooses `HS256` for HMAC-based signing or a compatible asymmetric default based on `client_assertion_private_key` (for example `RS256`, `RS384`, `ES256`, `ES384`, `ES512`, or `EdDSA` for Ed25519). `RS512`, `PS256`, `PS384`, and `PS512` are not currently supported for outbound signed authorization requests.

`request_object_audience`
: Optional override for the `aud` claim used in signed authorization requests. By default, shinyOAuth uses the provider issuer when available. When `request_object_mode = "request"` or `"request_uri"`, the provider needs either a configured issuer or this override to identify the authorization server in the signed Request Object.

`request_object_encryption_alg`
: Optional JWE key-management algorithm override for encrypted Request Objects. Current outbound support is limited to `RSA-OAEP`. Encryption requires both this value and `request_object_encryption_enc`.

`request_object_encryption_enc`
: Optional JWE content-encryption algorithm override for encrypted Request Objects. Current outbound support is limited to the AES-CBC-HMAC family (`A128CBC-HS256`, `A192CBC-HS384`, `A256CBC-HS512`). Encryption also requires `request_object_encryption_alg`.

`request_object_encryption_kid`
: Optional key identifier (`kid`) used to select one provider encryption key and emit

the outer JWE kid header. This is mainly useful when the provider publishes more than one Request Object encryption key.

request_object_ttl

Positive number of seconds to keep signed authorization request objects (request JWTs) valid. When request_object_mode = "request_uri", shinyOAuth also uses this value as the default publication window for the referenced Request Object URI. Default is 45.

request_object_nbf_skew

Optional non-negative number of seconds. When provided, shinyOAuth adds an nbf claim set to iat - request_object_nbf_skew so deployments can tolerate small clock skew while still emitting bounded request-object validity windows. NULL (the default) omits nbf. Request-object nbf is reserved by shinyOAuth and cannot be supplied through extra authorization parameters.

dpop_private_key

Private key for tying tokens to this app's requests using Demonstrating Proof of Possession (DPoP). Only needed when your provider/API supports DPoP. Accepts an openssl::key or PEM private-key string, using RSA, EC, or Ed25519. oauth_client() then defaults dpop_require_access_token to TRUE. Supported signing algorithms are RS256, RS384, ES256, ES384, ES512, and Ed25519 or legacy EdDSA with Ed25519 keys; RSA-PSS and other RSA signing algorithms are not supported for outgoing proofs. See dpop_signing_alg and the advanced security vignette.

dpop_private_key_kid

Optional key identifier (kid) to include in the JOSE header of DPoP proofs. Useful when the authorization or resource server expects a stable key identifier alongside the embedded public JWK.

dpop_signing_alg

Optional JWT signing algorithm to use for DPoP proofs. When omitted, a compatible asymmetric default is selected based on the private key type/curve (for example RS256, ES256, ES384, or ES512, or EdDSA for Ed25519). RS512, PS256, PS384, and PS512 are not currently supported for outbound DPoP proofs. If an explicit value is provided but incompatible with the key, validation fails early with a configuration error. When the provider advertises dpop_signing_alg_values_supported, both explicit values and inferred defaults must be included in that set.

dpop_require_access_token

Logical or NULL. When TRUE and dpop_private_key is configured, shinyOAuth requires the authorization server to return token_type = "DPoP" for access tokens and rejects other token types, independently of the access token's representation. Observed binding data must match the configured key; requiring its presence is a separate policy (dpop_require_observed_cnf). In oauth_client(), the default NULL resolves to TRUE when dpop_private_key is configured and to FALSE otherwise. Explicit FALSE also permits Bearer access tokens, including deployments where DPoP binds only the refresh token.

scope_validation

Controls how scope discrepancies are handled when the authorization server grants fewer scopes than requested. RFC 6749 Section 3.3 permits servers to issue tokens with reduced scope, and Section 5.1 allows token responses to omit scope when it is unchanged from the requested scope.

- "warn" (default): Emits a warning but continues authentication if scopes are missing.
- "strict": Throws an error if any requested scope is missing from the granted scopes. Omitted scope is treated as unchanged, not as an error.
- "none": Skips scope validation entirely.

claims_validation
: What to do if requested claims are missing or have unexpected values: "warn" continues with a warning, "strict" stops login, and "none" skips the check. When omitted, oauth_client() uses "warn" if claims includes essential = TRUE, value, or values requirements, and "none" otherwise. Checks on claims[["id_token"]] require ID token validation (id_token_validation = TRUE or use_nonce = TRUE). Checks on claims[["userinfo"]] require userinfo_required = TRUE and a configured UserInfo endpoint on the provider.

userinfo_jwt_required_time_claims
: Optional character vector of temporal JWT claims that must be present when the UserInfo response is a signed JWT (application/jwt). Allowed values are "exp", "iat", and "nbf".

  The default, character(0), checks these claims only when present. Set, for example, userinfo_jwt_required_time_claims = "exp" to require an expiry on signed UserInfo JWTs; multiple values require the corresponding time claims.

required_acr_values
: Optional character vector of acceptable login requirements, such as a provider's multi-factor authentication (MFA) policy. Use the provider's Authentication Context Class Reference (ACR) identifiers. The validated ID token must contain a matching acr or login fails. The request also sends acr_values as a hint to the provider. Requires id_token_validation = TRUE and an issuer. Default character(0) imposes no requirement.

introspect
: If TRUE, ask the provider to confirm the access token is active before completing login and module refreshes. Requires introspection_url; an unsuccessful check or a response other than active = TRUE stops the operation. Default FALSE.

introspection_checks
: Optional character vector of additional requirements to enforce on the introspection response when introspect = TRUE. Supported values:

  - "sub": require the introspected sub to match the session subject (from a validated ID token sub when available, else from userinfo sub).
  - "client_id": require the introspected client_id to match your OAuth client id.
  - "scope": validate introspected scope against requested scopes (respects the client's scope_validation mode).
  - "token_type": require introspection to return token_type. This is useful for sender-constrained deployments such as DPoP, where introspection can authoritatively report token_type = "DPoP". Default is character(0). Providers may omit these fields from introspection responses.

authorization_server_mode
: Declares whether this client is part of an application that can interact with more

than one authorization server, and which RFC 9700 mix-up defense it uses. One of:

- `"single"` (default): the application uses only one authorization server, so RFC 9700 does not require a mix-up defense.
- `"multi_issuer"`: authorization responses identify their issuer. JARM response modes satisfy this requirement through their validated `iss` claim. Direct response modes require the provider to advertise `authorization_response_iss_parameter = TRUE`; shinyOAuth then requires and validates the RFC 9207 `iss` response parameter. Missing support metadata is treated as absence of this defense.
- `"multi_redirect_uri"`: each authorization server uses a distinct redirect URI. Supply the complete set through `authorization_server_redirect_uris`. This mode is supported by `oauth_module_server()`, which compares the browser-visible canonical scheme, authority, and path before parsing callback values.

`authorization_server_redirect_uris`

Complete character vector of redirect URIs used by the application for its authorization servers when `authorization_server_mode = "multi_redirect_uri"`. It must contain at least two canonically distinct scheme/authority/path routes and include this client's `redirect_uri`. Query and fragment components do not make routes distinct.

`dpop_require_observed_cnf`

Logical. When `TRUE`, shinyOAuth rejects `token_type = "DPoP"` access tokens unless it can observe `cnf[["jkt"]]` locally, from the token response, introspection, or optional JWT access-token inspection. Set `options(shinyOAuth.access_token_cnf = "opaque")` to disable access-token decoding for both DPoP and mTLS; the compatibility default `"jwt"` inspects JWT `cnf` without treating it as signature validation. Opaque tokens then need binding metadata in the token response or introspection result. Default is `FALSE`.

`jarm_signed_response_alg`

Optional expected JWS algorithm for signed JWT Secured Authorization Responses (JARM). When omitted and the effective response mode is JARM, shinyOAuth defaults to `RS256`. This value is not sent dynamically on the authorization request; it must match the client metadata and provider behavior configured out-of-band for that client. Current inbound support accepts `HS256`, `HS384`, `HS512`, `RS256`, `RS384`, `RS512`, `ES256`, `ES384`, `ES512`, `Ed25519`, and `EdDSA`. RSA-PSS (`PS256`, `PS384`, `PS512`) and unsecured `none` are not accepted for inbound JARM.

`jarm_encrypted_response_alg`

Optional expected JWE key-management algorithm for encrypted JARM responses. Current inbound support is limited to `RSA-OAEP`. Like `jarm_signed_response_alg`, this reflects out-of-band client metadata and expected provider behavior rather than an authorization request parameter emitted by shinyOAuth.

`jarm_encrypted_response_enc`

Optional expected JWE content-encryption algorithm for encrypted JARM responses. Current inbound support is limited to the AES-CBC-HMAC family (`A128CBC-HS256`, `A192CBC-HS384`, `A256CBC-HS512`). When omitted while `jarm_encrypted_response_alg` is set, shinyOAuth defaults to `A128CBC-HS256`.

This must also match the provider-side JARM client metadata when encrypted responses are enabled.

`jarm_decryption_private_key`
: Optional private key used to decrypt encrypted JARM responses. Can be an `openssl::key` or a PEM string containing a private key. Required when encrypted JARM is enabled.

`jarm_decryption_private_key_kid`
: Optional key identifier (`kid`) associated with `jarm_decryption_private_key`.

`jarm_max_lifetime`
: Positive number of seconds. Maximum accepted lifetime for a JARM response JWT. Default is 600 seconds, matching JARM's recommended 10-minute upper bound for authorization response JWTs. When a JARM payload includes `iat`, shinyOAuth enforces `exp - iat <= jarm_max_lifetime`; otherwise it falls back to the remaining `exp` window at validation time. Applies only when `response_mode` uses JARM.

`endpoint_auth`
: Named list of authentication overrides for `par`, `introspection`, and `revocation`. Token exchange and refresh use the top-level client/provider authentication settings. Each entry may supply `token_auth_style`, `client_secret`, `client_assertion_private_key`, `client_assertion_private_key_kid`, `client_assertion_alg`, `client_assertion_audience`, `client_assertion_typ`, `extra_headers` (named character vector), and the `mtls_client_*` certificate/key/CA fields. Introspection and revocation may also use a separate `client_id`. Unspecified credentials inherit the client's settings. Discovered endpoint methods and signing algorithms are checked independently. PAR inherits token authentication. Extra token headers apply only to token exchange and refresh; set `extra_headers` explicitly for every other endpoint that needs them.

`mtls_require_observed_cnf`
: Logical, default `TRUE`. When `mtls_certificate_bound_access_tokens = TRUE`, require `cnf[["x5t#S256"]]` in the token response, JWT access token, or introspection and verify that it matches the configured certificate. `FALSE` allows missing confirmation for server-enforced opaque bindings. Observed confirmation is validated; mismatches and conflicting claims are rejected.

`trusted_id_token_audiences`
: Character vector of additional ID-token audiences explicitly trusted by this client. Defaults to `character(0)`, which permits only `client_id`. The token must always include `client_id` in `aud`; when `azp` is present it must equal `client_id`. Values are matched exactly and case-sensitively and apply to identity-token recipients, independently of API resource addresses.

`compare_callback_issuer`
: Logical or `NULL`. Compare any supplied callback `iss` exactly with `provider@issuer`, while allowing absence when `enforce_callback_issuer = FALSE`. `NULL` enables comparison when an issuer is configured, except when `enforce_callback_issuer = FALSE` was explicitly supplied. This preserves the existing complete opt-out. Set `compare_callback_issuer = TRUE` with `enforce_callback_issuer = FALSE` to check present values without requiring older providers to send `iss`. Required issuer presence always enables comparison, even when this separate flag is `FALSE`. Validated JARM supplies its own issuer protection without requiring a redundant outer `iss`.

client_assertion_typ
: JWT header typ for client authentication. Defaults to "JWT" for existing providers. Use "client-authentication+jwt" with client_assertion_audience set to the provider's trusted issuer identifier for RFC7523bis-11 / OAuth 2.1 draft 16.

authorization_method
: Browser method for sending the authorization request: "GET" (default) or provider-supported "POST". POST submits form fields instead of a URL query. Use the module's request_login() or prepare_authorization_request(); URL-only helpers reject POST.

resource_bases
: Optional named character vector of approved API base URLs for oauth_connection() and oauth_connections(), default character(). Each resource ID starts with a letter and contains letters, digits, _ or - (at most 64 bytes). Up to 64 bases are supported. HTTPS is required except for loopback development URLs. Requests through a connection stay within the exact scheme, host, effective port and base path; redirects are disabled. Bases exclude user information, query strings, fragments, dot segments, repeated slashes, semicolon parameters and ambiguous encoded characters. This is local request policy, not evidence of token audience, and does not add the OAuth resource authorization parameter.

required_scopes
: Optional requested scopes that every usable connection needs, default character(). Other requested scopes may be absent from a limited grant. Ordinary OAuth clients compare literal scopes; smart_client() selects SMART semantic comparison and also enforces these permissions when validating token responses. Explicit refresh narrowing retains these scopes.

label
: Optional display label used in connection summaries; defaults to the provider name, with control characters replaced by spaces and shortened to 128 UTF-8 bytes if needed. If the provider name is empty, missing or not a single string, the default is "OAuth provider". Explicit labels must be non-empty strings of at most 128 bytes without control characters. Labels contain no credentials or patient context.

...
: Deprecated renamed arguments accepted temporarily for backward compatibility.

introspect_elements
: Compatibility alias for introspection_checks. Supply only one spelling.

token_targets
: Optional named list for obtaining tokens for several APIs through one login. Build entries with oauth_token_target() or plain named lists. Each entry needs a resource identifier and scopes (also included in explicit client scopes). Optional required_scopes set that API's minimum permissions; resource_ids associate it with named resource_bases. Optional Microsoft scope_aliases identify unusual short permission names in responses. Requires provider support and userinfo_required = FALSE; cannot be combined with resource or SMART configuration. See vignette("token-targets", package = "shinyOAuth") (online guide) for examples, provider-specific permission rules and declaration limits.

token_target_default
: Name of the API to use when a connection method omits target. Required with several targets; the sole target is selected automatically.

Details

Create the client outside `server()` so its settings and pending login state remain available when the callback returns. Configure `provider`, `client_id`, `client_secret` (if issued), `redirect_uri`, and `scopes` from the app registration. Use `state_store` and `state_key` for shared login state across workers, and validation arguments to require particular scopes, claims, or authentication context. See the usage vignette for a complete app, or the advanced security vignette for certificate and signed-request settings.

Value

OAuthClient object

Claims requests

Use an R list for most applications. `values` vectors become JSON arrays, including single-element vectors. Each claim entry is `NULL` or a named list; `essential` must be logical, and `value` and `values` cannot be combined. Pre-encoded JSON must use unique member names, object-valued target sections and nonempty arrays for `values`. Invalid requests are rejected even when `claims_validation = "none"` disables checking returned values.

Numbers use R's numeric precision and must be finite and within `+/- (2^53 - 1)`. Use strings for larger exact identifiers or decimal values that R cannot represent accurately. UserInfo claim checks require automatic profile retrieval and a configured UserInfo endpoint.

Examples

```
# Register an app with GitHub and store its credentials in your environment.
# This creates the configuration; it does not start login or contact GitHub.
client <- oauth_client(
  provider = oauth_provider_github(),
  client_id = "your-client-id",
  client_secret = "your-client-secret",
  redirect_uri = "http://127.0.0.1:8100",
  scopes = c("read:user", "user:email")
)

# In a real app, read credentials with Sys.getenv() and create client
# outside server(). Inside server(), start login with:
# auth <- oauth_module_server("auth", client)
```

`oauth_client_mtls_registration` *Prepare client-certificate registration settings (mTLS)*

Description

Build a list of settings to register a certificate-based client with your provider using mutual TLS (mTLS). Use this when preparing metadata for dynamic client registration or when your provider asks for certificate identifiers, public keys, or certificate-bound token settings. It derives those settings from an `oauth_client()` already configured for mTLS.

The result is a metadata list for the provider's registration process. This function does not register the client or upload the certificate.

Usage

```
oauth_client_mtls_registration(
  client,
 tls_client_auth_type = c("subject_dn", "san_dns", "san_uri", "san_ip", "san_email"),
  tls_client_auth_value = NULL,
  jwks_uri = NULL,
  oauth_client = NULL
)
```

Arguments

`client`	OAuthClient configured for RFC 8705 mutual TLS client authentication or for certificate-bound access tokens.
`tls_client_auth_type`	For `tls_client_auth`, which RFC 8705 certificate identifier field to emit. One of `"subject_dn"`, `"san_dns"`, `"san_uri"`, `"san_ip"`, or `"san_email"`.
`tls_client_auth_value`	Optional explicit value for the selected `tls_client_auth_type`. When omitted, shinyOAuth derives the subject DN from the configured client certificate. SAN registration requires an explicit value because the current certificate extractor does not preserve ASN.1 SAN types. Select the type and exact value from the certificate; a numeric-looking DNS name is still a DNS SAN, not an IP SAN. Explicit `subject_dn` values use RFC 4514 syntax and are preserved verbatim, including escaped spaces.
`jwks_uri`	Optional absolute URL of a JWKS document to publish for `self_signed_tls_client_auth`. When omitted, the helper returns an inline `jwks` object with the configured client certificate chain in `x5c`.
`oauth_client`	Compatibility alias for `client`. Supply only one spelling.

Details

For `tls_client_auth`, the result identifies the client certificate using one selected subject or alternative-name field. For `self_signed_tls_client_auth`, it contains an inline `jwks` with the certificate chain (`x5c`), or the supplied `jwks_uri`.

For certificate-bound tokens without mTLS client authentication, the result uses the corresponding registration authentication method (for example, `public` becomes `none`) and sets `tls_client_certificate_bound_access = TRUE`. See the advanced security vignette for when these configurations are useful.

Value

A JSON-ready list of RFC 7591/RFC 8705 client metadata.

Examples

```
# Set these environment variables to your existing certificate and key files.
provider <- oauth_provider(
  name = "Example service",
  auth_url = "https://example.com/authorize",
  token_url = "https://example.com/token",
  token_auth_style = "tls_client_auth"
)
client <- oauth_client(
  provider = provider,
  client_id = "example-client",
  redirect_uri = "http://127.0.0.1:8100/callback",
  mtls_client_cert_file = Sys.getenv("OAUTH_MTLS_CERT_FILE"),
  mtls_client_key_file = Sys.getenv("OAUTH_MTLS_KEY_FILE")
)
oauth_client_mtls_registration(client)
```

---

oauth_client_secret_apple

Create a client secret for Sign in with Apple

Description

Use your Apple developer key to create the client secret expected by Sign in with Apple. Pass the returned string as `client_secret` to `oauth_client()`. The returned secret expires after `expires_in` seconds.

Usage

```
oauth_client_secret_apple(
  client_id,
  team_id,
  key_id,
  private_key,
  expires_in = 15776700,
  issued_at = Sys.time(),
  audience = "https://appleid.apple.com"
)
```

Arguments

`client_id`	Apple Services ID or App ID used as the OAuth client id
`team_id`	Apple Developer Team ID. Apple documents this as a 10-character identifier

key_id	Apple Sign in with Apple private-key identifier (kid). Apple documents this as a 10-character identifier
private_key	Apple private key as an openssl::key or PEM string. The key must be compatible with ES256 (P-256 ECDSA)
expires_in	Positive whole-number lifetime in seconds. Must be at least one second and no more than 15777000 seconds (six months). Defaults to 15776700 seconds, leaving a five-minute margin below Apple's documented maximum
issued_at	Issue time for the JWT. Defaults to Sys.time()
audience	Audience claim. Defaults to "https://appleid.apple.com"

Details

The helper signs a JWT with ES256. It places your Team ID in iss, your client ID in sub, Apple's URL in aud, and the key ID in the header. The lifetime must not exceed 15,777,000 seconds (about six months).

Value

A compact signed JWT string suitable for oauth_client(..., client_secret = ...)

Examples

```
## Not run:
key <- openssl::ec_keygen(curve = "P-256")

oauth_client_secret_apple(
  client_id = "com.example.web",
  team_id = "ABCDEFGHIJ",
  key_id = "ABC123DEFG",
  private_key = key
)

## End(Not run)
```

---

oauth_connection *Wrap an existing reactive token as a connection*

Description

Combine one module's reactive token with its client and approved API addresses configured on oauth_client(). Call [["request"]]() on the connection to send a request with the current token. It reads the reactive token again after refresh or logout and restricts requests to the configured APIs. This optional wrapper expires with its Shiny session; it does not implement refresh itself or retain credentials across redirects. Clients with token_targets must use the module's connection() factory, which retains the authorization's permission limits alongside its tokens.

Usage

```
oauth_connection(
  client,
  token_reactive,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

client	An OAuthClient with non-empty resource_bases, created by oauth_client() or smart_client().
token_reactive	A Shiny reactive expression returning the current OAuthToken or NULL, usually shiny::reactive(auth[["token"]]). It must come from the module using client. This association is supplied by the caller; API address restrictions cannot verify an opaque token's audience.
session	The owning Shiny session; defaults to the current session.

Details

The module's auth$connection() factory also supports token acquisition and refresh for external packages. This wrapper adapts an existing reactive token.

The reference is bound to its creating Shiny session and checks that session's reactive context. [["summary"]]() excludes tokens, identity claims and extension context. [["is_usable"]]() checks local presence, known expiry and required scopes; it cannot guarantee remote authorization. Unknown token expiry is unusable.

Paths are relative to the selected base directory. Absolute and root-relative references (including pagination links) must stay within that same base. Dot segments and ambiguous encodings are rejected; redirects are never followed. Bearer, DPoP and mTLS use the existing transport and the configured client.

Use request-level required_scopes for optional operations. They must be included in the client's requested scopes and covered by the current grant. The package cannot infer arbitrary API permissions from an HTTP method/path. oauth_module_server() still manages refresh and logout. To retain credentials across Shiny sessions, use oauth_connections() and oauth_connections_server().

Value

An OAuthConnection with [["id"]], [["is_usable"]](), [["summary"]]() and [["request"]](resource_id, path = Requests return httr2 responses. configure can add a body and application headers; see OAuthConnection for its contract.

Examples

```
## Not run:
# Configure outside server():
client <- oauth_client(provider, "registered-app",
  redirect_uri = "https://app.example/callback", scopes = "read",
  resource_bases = c(api = "https://api.example/v1"))
# Inside server():
```

```
auth <- oauth_module_server("auth", client)
connection <- oauth_connection(client, shiny::reactive(auth[["token"]]))
data <- shiny::reactive({
  shiny::req(connection[["is_usable"]]())
  connection[["request"]]("api", "records", required_scopes = "read")
})

## End(Not run)
```

`oauth_connection_store_memory` *Create a process-local store for retained OAuth connections*

Description

Stores encrypted credentials for `oauth_connections()` across Shiny sessions with browser or account retention. The store belongs to the manager rather than an individual Shiny session. Records last for the R process's lifetime; the store rejects use from a copied worker process.

Usage

```
oauth_connection_store_memory(
  max_age = 28800,
  refresh_timeout = 60,
  max_entries = 1000L
)

## S3 method for class 'OAuthConnectionStore'
print(x, ...)
```

Arguments

`max_age`	Maximum record and transaction-deduplication lifetime in seconds. Each record must also supply an earlier or equal absolute expiry.
`refresh_timeout`	Maximum time in seconds for a claimed refresh. An abandoned operation becomes uncertain; its old credentials cannot be retried.
`max_entries`	Maximum live record and transaction-reservation count, including tombstones and recently expired transactions. A full store rejects creation instead of evicting another connection.
`x`	An `OAuthConnectionStore` adapter to print.
`...`	Unused print arguments.

Details

These are low-level storage operations, called after the manager validates the current owner session and generation. The adapter scopes records by owner ID; it does not authenticate a caller from an owner or connection ID. Records contain ciphertext only. Encryption keys stay with the manager, outside this adapter.

- `[["create"]](owner, id, transaction, client, fingerprint, sealed, expires_at)` returns a record with a new revision, or `NULL` for a duplicate connection/transaction.
- `[["read"]](owner, id)` returns the owner's record, or `NULL` when absent, expired, or owned by someone else. It includes the sealed envelope for internal use.
- `[["list"]](owner)` returns metadata lists without ciphertext or operation IDs.
- `[["begin_refresh"]](owner, id, revision)` claims an active record and returns its new revision and operation ID. A conflicting claim returns `NULL`.
- `[["commit_refresh"]](owner, id, operation, revision, sealed)` installs credentials only for the current claim, returning the updated record or `NULL`.
- `[["fail_refresh"]](owner, id, operation, revision, outcome)` releases a claim only for `"not_consumed"`; `"possibly_consumed"` and `"consumed"` remove the old envelope and mark the record `"uncertain"`. Returns the record or `NULL`.
- `[["disconnect"]](owner, id, revision)` first installs a credential-free tombstone, then returns `list(record, previous)` for bounded remote cleanup. Conflicts return `NULL`; the caller must reload before retrying.
- `[["disconnect_owner"]](owner)` tombstones all of that owner's records and returns the previous records for bounded cleanup.

Reads expire abandoned refresh claims before returning data. Tombstones and transaction IDs remain at least until the original absolute expiry so late completions cannot restore a disconnected grant. Operations are synchronous and do not yield or perform network calls. Atomicity applies to this R process only. A separate shared backend needs its own verified concurrency contract.

Value

An `OAuthConnectionStore` adapter with the methods described below.

See Also

`oauth_client()`, `oauth_connection()`

Examples

```
# Create once, outside server(), then supply to oauth_connections().
store <- oauth_connection_store_memory(max_age = 8 * 3600, max_entries = 100)
```

oauth_connections *Manage several OAuth connections in one Shiny app*

Description

Let users connect several services or accounts in one Shiny app. Create the manager outside `server()` with named clients from `oauth_client()` or `smart_client()`, then use `oauth_connections_ui()` and `oauth_connections_server()` to display and use their saved connections.

Usage

```
oauth_connections(
  clients,
  app_origin,
  retention = c("shiny", "browser", "account"),
  retention_seconds = 28800,
  owner_policy = NULL,
  store = NULL,
  keys = NULL,
  callback_policy = "distinct_routes"
)

## S3 method for class 'OAuthConnections'
print(x, ...)
```

Arguments

`clients`	Non-empty named list of OAuthClient objects, at most 64. `resource_bases` is required only for connection `$request()` calls; clients used solely to supply tokens to an SDK can omit it. Names select local configurations (for example `hospital_a`), independently of OAuth `client_id`: a letter followed by letters, digits, _ or -, at most 64 characters.
`app_origin`	Public application origin, including a non-default port. HTTPS is required for retained owners, except an explicit browser-owner HTTP loopback exception. Session-only development also permits loopback HTTP.
`retention`	`"shiny"` (default) discards connections at Shiny session end. `"browser"` restores the browser owner's connections after navigation; `"account"` uses a trusted local application login.
`retention_seconds`	Maximum lifetime of each stored grant, in seconds. Positive and finite, no larger than the store's `max_age`. Refresh never resets it. Browser retention is also capped by the owner's absolute expiry.
`owner_policy`	`oauth_browser_owner()` or `oauth_account_owner()` matching the retained mode. Must be `NULL` for session-only retention.

`store` A store from `oauth_connection_store_memory()`. Required for retained modes; session-only mode creates a memory store by default. Only the supplied memory store in one R process is supported.

`keys` Named list with `credentials` and `owner`, each a deployment-held raw vector of 32 bytes. Required for retained modes. Session-only mode creates ephemeral keys if omitted. Keys are held by the manager, separately from the ciphertext-only connection store.

`callback_policy`
`"distinct_routes"` (default) gives each client its own registered callback route. With several clients, configure every client with `authorization_server_mode = "multi_redirect_uri"` and the complete set of routes in `authorization_server_redirect_uris`. Every client's declared set must include all manager callback routes; additional application routes are permitted. Routes are compared by canonical origin and path. `"issuer"` allows shared routes for distinct authorization-server issuers. `"shared_routes"` additionally supports several clients or registrations at one issuer through a protected pending-state index. Both opt-in policies require explicit `authorization_server_mode = "multi_issuer"` clients, with RFC 9207 issuer responses or signed JARM. Encrypted JARM requires distinct routes. Routing never substitutes for callback authentication.

`x` A connection manager to print.

`...` Unused print arguments.

Details

By default, connections last for one Shiny session. Optional browser or local account retention can restore them after navigation. See `vignette("multiple-authorizations", package = "shinyOAuth")` (online guide) for a complete example.

The manager's named clients select the configuration for each login. Connections are separate local records, but the provider may reuse one grant; revoking a connection can therefore affect other tokens from that grant.

Value

An `OAuthConnections` server-side configuration object. Printing shows only the retention mode and client count. It holds client configuration and deployment keys for the UI and server modules.

Advanced deployment and shared credentials

Refreshes using one shared credential are coordinated. Rotation or an uncertain refresh can require other connections sharing that credential to reconnect; one connection's response never replaces another's identity or permissions. Revocation can also affect related provider grants.

A manager is bound to one UI/server module ID and one public origin. The memory store and owner registries survive Shiny sessions, not R restarts. Copies in another R process are rejected.

In session-only mode, a pending authorization survives navigation through the existing single-use OAuth state and browser binding, while existing grants are discarded with the old Shiny session. Browser/account mode additionally binds authorization to the initiating local owner and its session generation.

The manager supports generic OAuth clients and the SMART discovery, scope and launch policies configured by smart_client().

See Also

oauth_connections_ui(), oauth_connections_server()

Examples

```
# Replace this example provider and client ID with your registered application.
provider <- oauth_provider(
  name = "Example service",
  auth_url = "https://example.com/authorize",
  token_url = "https://example.com/token",
  token_auth_style = "public"
)
client <- oauth_client(
  provider = provider,
  client_id = "example-client",
  redirect_uri = "http://127.0.0.1:8100/callback/service",
  resource_bases = c(api = "https://api.example.com")
)

# Create the manager once, outside server(). Default retention is one Shiny session.
manager <- oauth_connections(
  clients = list(service = client),
  app_origin = "http://127.0.0.1:8100"
)
ui <- oauth_connections_ui(
  shiny::fluidPage(
    shiny::actionButton("connect", "Connect to service"),
    shiny::verbatimTextOutput("connections")
  ),
  id = "auth",
  manager = manager
)
server <- function(input, output, session) {
  auth <- oauth_connections_server("auth", manager)
  shiny::observeEvent(input[["connect"]], auth[["connect"]]("service"))
  output[["connections"]] <- shiny::renderPrint(auth[["connections"]]())
}

# Construct the app without launching it or contacting the provider.
app <- shiny::shinyApp(
  ui,
  server,
  uiPattern = ".*",
  options = list(port = 8100)
)
```

`oauth_connections_server` *Connect, restore and use several OAuth authorizations in a Shiny session*

Description

Let users connect several accounts or services, choose a saved connection, and call its API from your Shiny server. Call this once inside `server()` with the manager and ID used by `oauth_connections_ui()`. Connection summaries show status without exposing tokens; server code can obtain a current token from the selected connection when another R package needs it.

Usage

```
oauth_connections_server(
  id,
  manager,
  async = FALSE,
  refresh_proactively = FALSE,
  refresh_lead_seconds = 60,
  refresh_check_interval_ms = 10000
)
```

Arguments

`id`	Module ID shared with `oauth_connections_ui()`.
`manager`	Configuration from `oauth_connections()`.
`async`	Whether authorization and refresh use the existing async worker transport. Owner checks and store mutations stay in the original R process.
`refresh_proactively`	Refresh before expiry when a refresh credential is available. Otherwise the manager attempts refresh at expiry.
`refresh_lead_seconds`	Non-negative number of seconds before expiry used for proactive refresh. Background checks never extend owner inactivity limits.
`refresh_check_interval_ms`	Positive polling interval in milliseconds, at least 100. Safely retryable automatic refresh failures wait at least 30 seconds across all connections sharing the same refresh credential and registration; an uncertain refresh requires reconnecting.

Details

Use `auth$connections()` to populate a selection control, then `connection <- auth$connection(selected_id)` for the chosen account. `connection$request()` calls a configured API; `connection$access_token()` supplies a current token for another R package. See `vignette("multiple-authorizations", package = "shinyOAuth")` (online guide) for a complete app and `vignette("external-integrations", package = "shinyOAuth")` (online guide) for API client and database examples.

Value

A server-side list with:

- `connect(client_name)`: request a new authorization without discarding others. EHR-only clients report `fresh_ehr_launch_required` and return `FALSE`; use their registered `smart_launch_route()` to start authorization.
- `connections()`: reactive list of redacted connection summaries.
- `connection(connection_id = NULL)`: an OAuthConnection for requests, refresh, scope checks and server-side `$access_token()` retrieval for external SDKs.
- `reauthorize(connection_id)`: end this local authorization and start its replacement with the retained scope limit, without upstream revocation. Local state preparation is checked first; rejection preserves current access. The replacement summary includes `replaces_connection_id`. Failed/cancelled replacement leaves the old reference invalid. EHR-only clients require a fresh EHR launch instead.
- `touch()`: record explicit user activity after checking the current owner; returns `TRUE` invisibly.
- `disconnect(connection_id, revoke = TRUE)`: remove local usability first, then return separate `local` and `remote` revocation results. For token targets, `remote$access` summarizes all acquired API tokens; incomplete or failed cleanup takes priority over successful cleanup.
- `disconnect_all(revoke = TRUE)`: cancel pending authorizations and disconnect this owner's stored connections; return a list of results.
- `logout(revoke = TRUE, reload = TRUE)`: end the local owner session, disconnect its connections, and normally reload the UI. This does not log the user out of the external OAuth provider or the app's own account authentication system.
- `errors()`: reactive list of per-client module error codes, with no raw provider text. An ended owner is reported as `owner_unavailable`.

Advanced lifetime and refresh behavior

Each Shiny session needs its own connection references, even when the manager retains the saved authorizations. A new session obtains these after owner verification. The manager coordinates refresh across its connections without changing the original authentication time or retention expiry. Reactive connection reads also recheck expiry at `refresh_check_interval_ms`, including references used without `connections()` or `errors()`. These checks notify reactive expressions when connection status changes; unchanged polling does not rerun application requests or extend owner inactivity limits. Notifications to application code reflect only this owner's record changes. Without an ID, `connection()` returns NULL when no authorization remains, selects the sole retained authorization, or raises `shinyOAuth_access_error` with reason `selection_required` when several remain. Expired, limited and uncertain authorizations count toward ambiguity; disconnected rows do not.

Reactive expressions that read `connection()`, `$access_token()` or `$has_scopes()` do not rerun on token refresh alone if the login and permissions stay the same. Token acquisition never extends owner idle limits or replays application requests. See `vignette("external-integrations", package = "shinyOAuth")` (online guide). Resource requests and status reads do not reset owner inactivity, including reactive reads after automatic refresh. `touch()` explicitly records activity, for example from an input event handler. Connecting, explicitly refreshing and disconnecting also count as activity.

Remote revocation is best effort: at most ten seconds per disconnect/logout batch, at most two seconds and one HTTP attempt per credential. Results are `accepted`, `unsupported`, `missing`, `failed` or `not_attempted` for access and refresh credentials. `accepted` describes the endpoint response, not proof of prior token validity. Local disconnect remains effective if revocation fails. A provider may revoke an entire authorization grant or related credentials (RFC 7009 section 2.1). Separate connection IDs do not establish independent provider grants, even after repeated consent or account selection. Consequently, default `revoke = TRUE` may also end access for sibling connections or other applications covered by the provider's revocation policy. With `revoke = FALSE`, disconnect removes local access without requesting remote revocation. This also applies to credentials returned by work already in flight. Removed credentials remain valid remotely until the provider expires or revokes them. Local summaries and `is_usable()` reflect local state, independently of remote provider changes.

The API checks its owning session's reactive context. Session setup requires a matching HTTP Origin on the Shiny request. Raw HTTP routes cannot import credentials or select an owner. The manager supports one R process.

Examples

```
# Replace this example provider and client ID with your registered application.
provider <- oauth_provider(
  name = "Example service",
  auth_url = "https://example.com/authorize",
  token_url = "https://example.com/token",
  token_auth_style = "public"
)
client <- oauth_client(
  provider = provider,
  client_id = "example-client",
  redirect_uri = "http://127.0.0.1:8100/callback/service",
  resource_bases = c(api = "https://api.example.com")
)

# Create the manager once, outside server(). Default retention is one Shiny session.
manager <- oauth_connections(
  clients = list(service = client),
  app_origin = "http://127.0.0.1:8100"
)
ui <- oauth_connections_ui(
  shiny::fluidPage(
    shiny::actionButton("connect", "Connect to service"),
    shiny::verbatimTextOutput("connections")
  ),
  id = "auth",
  manager = manager
)
server <- function(input, output, session) {
  auth <- oauth_connections_server("auth", manager)
  shiny::observeEvent(input[["connect"]], auth[["connect"]]("service"))
  output[["connections"]] <- shiny::renderPrint(auth[["connections"]]())
}
```

```
# Construct the app without launching it or contacting the provider.
app <- shiny::shinyApp(
  ui,
  server,
  uiPattern = ".*",
  options = list(port = 8100)
)
```

`oauth_connections_ui` *Handle callbacks and establish the connection owner's browser session*

Description

Handles login returns and restores the user's saved connections. The wrapper and `oauth_connections_server()` share the same manager and module ID. Browser setup from `use_shinyOAuth()` is included.

Usage

```
oauth_connections_ui(
  base_ui,
  id,
  manager,
  request_uri_resolver = NULL,
  app_base_path = "/",
  launch_routes = list(),
  additional_clients = list()
)
```

Arguments

`base_ui`	A Shiny UI object or request-dependent UI function.
`id`	Module ID shared with `oauth_connections_server()`.
`manager`	Configuration from `oauth_connections()`.
`request_uri_resolver`	Optional trusted function mapping a Rook request to its public absolute URI. Required when a trusted reverse proxy changes the apparent scheme/host. The custom resolver defines which proxy headers are trusted; the wrapper checks the resolved origin against the manager's origin.
`app_base_path`	Public path at which the Shiny application is hosted, default `"/"`. Must begin and end with `/`. The wrapper inserts a document base before scripts so Shiny dependencies load from the app root even on nested callback pages. The wrapper manages the page's HTML `base` element. Managed JAR Request Object URLs are also published beneath this path.
`launch_routes`	List of `smart_launch_route()` configurations, empty by default. EHR entry requires browser retention and top-level navigation.

additional_clients
: Optional named list of ordinary OAuth/OIDC clients used by separate oauth_module_server() modules. Names are their full module IDs. These clients keep their existing login lifecycle and are not managed connections. This wrapper dispatches their callbacks alongside managed ones. Their callback routes must be distinct from managed callbacks and SMART launch routes, on this app's origin and inside app_base_path. All clients must select an appropriate multi-server authorization mode, even when the manager contains one client.

Details

With shinyApp(..., uiPattern = ".*"), callback paths reach the wrapper. With browser retention, a cookie lets the returning browser find its saved connections.

Value

A request UI function for shinyApp(..., uiPattern = ".*").

Advanced browser and proxy requirements

With Strict cookies, returning from another site may briefly show an intermediate page before Shiny starts. This lets the browser send its existing cookie. Proxies must preserve same-origin Fetch Metadata, Referer or Origin information for recovery. The shinyOAuth_owner_recovery query parameter is reserved for this flow.

A managed login return must still belong to a valid local owner. An ordinary entry or an independently validated additional_clients callback can create an empty browser owner, but cannot recover someone else's connections. Invalid cookies are cleared before a fresh owner is created.

Cookies are shared across ports and paths on the same hostname. Services on that hostname share the owner cookie's trust boundary.

See Also

oauth_browser_owner(), oauth_ui()

Examples

```
# Replace this example provider and client ID with your registered application.
provider <- oauth_provider(
  name = "Example service",
  auth_url = "https://example.com/authorize",
  token_url = "https://example.com/token",
  token_auth_style = "public"
)
client <- oauth_client(
  provider = provider,
  client_id = "example-client",
  redirect_uri = "http://127.0.0.1:8100/callback/service",
  resource_bases = c(api = "https://api.example.com")
)
```

```
# Create the manager once, outside server(). Default retention is one Shiny session.
manager <- oauth_connections(
  clients = list(service = client),
  app_origin = "http://127.0.0.1:8100"
)
ui <- oauth_connections_ui(
  shiny::fluidPage(
    shiny::actionButton("connect", "Connect to service"),
    shiny::verbatimTextOutput("connections")
  ),
  id = "auth",
  manager = manager
)
server <- function(input, output, session) {
  auth <- oauth_connections_server("auth", manager)
  shiny::observeEvent(input[["connect"]], auth[["connect"]]("service"))
  output[["connections"]] <- shiny::renderPrint(auth[["connections"]]())
}

# Construct the app without launching it or contacting the provider.
app <- shiny::shinyApp(
  ui,
  server,
  uiPattern = ".*",
  options = list(port = 8100)
)
```

`oauth_form_post_ui` *Wrap a Shiny UI to enable OAuth 2.0/OIDC form_post callbacks*

Description

Accepts the provider's login callback as an HTTP POST for `response_mode = "form_post"` or `"form_post.jwt"`. These modes keep callback parameters out of the browser URL. The POST arrives before a Shiny session exists; the wrapper receives it and makes the validated callback available to `oauth_module_server()`. `oauth_ui()` handles query-string callbacks.

Usage

```
oauth_form_post_ui(
  base_ui,
  id = NULL,
  client = NULL,
  callback_path = NULL,
  request_uri_resolver = NULL,
  clients = NULL
)
```

Arguments

base_ui	Existing Shiny UI object, or a UI function accepting req.
id	Shiny module id used by oauth_module_server(). This must match the id argument passed to the server module.
client	OAuthClient object used by oauth_module_server().
callback_path	Optional URL path to accept POST callbacks on. Defaults to the path component of client@redirect_uri and must match it when supplied. This is the public callback path; use request_uri_resolver to map a trusted proxy's backend path to the registered public URI.
request_uri_resolver	Optional function accepting the Rook req environment and returning the trusted, public absolute request URI without relying on query parameters. Return NULL to reject the route. This is intended for HTTPS-terminating proxies whose backend request has an HTTP Rook scheme. The function must verify the proxy trust boundary before using forwarded headers; its result is still required to match the configured redirect origin and callback_path. Registered fixed query parameters must also occur unchanged in the incoming request. Continuation URLs preserve only registered application parameters.
clients	Optional client registry as in oauth_ui(). With a registry, omit id, client, and callback_path; each client's redirect URI sets its route, and both query and form-post clients are supported.

Details

The wrapper and oauth_module_server() share the same id and client. The wrapper includes browser setup and response headers. Callback paths below the app root require uiPattern = ".*" in shiny::shinyApp() so Shiny routes those requests to the wrapper.

The wrapper receives the login result before the Shiny session starts, then sends the browser to a clean app URL. Raw callback values stay out of that URL. The server module completes the usual login checks.

Value

A Shiny UI function for shiny::shinyApp().

Advanced callback storage

Responses are held briefly in the client's state store under single-use handles. They expire after at most 120 seconds, or sooner if the state policy requires it. Pending responses are not replaced to make room for new ones; a full callback pool rejects additional responses until space becomes available. Shared stores need atomic take and set_if_absent; see custom_cache(). Pending callbacks created by an older version must be restarted after upgrading. form_post.jwt also verifies the signed authorization response (JARM).

Deployment behind a proxy

The default resolver uses the server-observed address and ignores forwarded headers. For an HTTPS-terminating proxy, a custom `request_uri_resolver` can reconstruct the public address from headers accepted at that proxy's trust boundary. The wrapper checks the result against the configured redirect origin and callback path. See the advanced security vignette.

Callback size limits are documented in the package options reference.

Examples

```
if (
  # Example requires a local or remote Keycloak realm whose client allows
  # http://127.0.0.1:8100/callback as a valid redirect URI.
  nzchar(Sys.getenv("KEYCLOAK_BASE_URL")) &&
    nzchar(Sys.getenv("KEYCLOAK_REALM")) &&
    nzchar(Sys.getenv("KEYCLOAK_CLIENT_ID")) &&
    interactive()
) {
  library(shiny)
  library(shinyOAuth)

  options(shinyOAuth.allow_insecure_oidc_loopback = TRUE)

  provider <- oauth_provider_keycloak(
    base_url = Sys.getenv("KEYCLOAK_BASE_URL"),
    realm = Sys.getenv("KEYCLOAK_REALM")
  )

  client <- oauth_client(
    provider = provider,
    client_id = Sys.getenv("KEYCLOAK_CLIENT_ID"),
    client_secret = Sys.getenv("KEYCLOAK_CLIENT_SECRET"),
    redirect_uri = "http://127.0.0.1:8100/callback",
    scopes = c("openid", "profile", "email"),
    response_mode = "form_post"
  )

  base_ui <- fluidPage(
    uiOutput("login")
  )

  ui <- oauth_form_post_ui(base_ui, id = "auth", client = client)

  server <- function(input, output, session) {
    auth <- oauth_module_server("auth", client, auto_redirect = TRUE)

    output[["login"]] <- renderUI({
      if (auth[["authenticated"]]) {
        user_info <- auth[["token"]]@userinfo
        tagList(
          tags[["p"]]("You are logged in!"),
          tags[["pre"]](paste(capture.output(str(user_info)), collapse = "\n"))
```

```
        )
      } else {
        tags[["p"]]("You are not logged in.")
      }
    })
  }

  runApp(
    shinyApp(ui, server, uiPattern = ".*"),
    port = 8100,
    launch.browser = FALSE
  )
}
```

`oauth_module_server` *OAuth 2.0 authorization and OIDC authentication module for Shiny*

Description

Manage login for a Shiny server module. The module sends the browser to the provider, validates the return, and publishes reactive login status and user information. An `oauth_client()` supplies the registration and policy; `oauth_ui()` handles query callbacks and `oauth_form_post_ui()` handles POST callbacks. The matching UI wrapper includes browser setup.

With an OpenID Connect (OIDC) provider, it also validates the signed-in user's identity.

Usage

```
oauth_module_server(
  id,
  client,
  auto_redirect = TRUE,
  async = FALSE,
  indefinite_session = FALSE,
  reauth_after_seconds = NULL,
  refresh_proactively = FALSE,
  refresh_lead_seconds = 60,
  refresh_check_interval_ms = 10000,
  revoke_on_session_end = FALSE,
  tab_title_cleaning = TRUE,
  tab_title_replacement = NULL,
  request_uri_base_url = NULL,
  browser_cookie_path = NULL,
  browser_cookie_samesite = c("Strict", "Lax", "None"),
  refresh_check_interval = NULL
)
```

Arguments

id	A name for this Shiny module, such as "auth".
client	The app configuration created with oauth_client().
auto_redirect	If TRUE (default), start login automatically for unauthenticated sessions. If FALSE, call auth[["request_login"]]() to start it.
async	If TRUE, run the module's network work through a background backend. Configure mirai daemons or a non-sequential future plan first; mirai takes priority when both are configured. Default FALSE. future::sequential() runs in the main process. See Asynchronous execution for operations that remain synchronous.
indefinite_session	If TRUE, the module will not automatically clear the token due to access-token expiry or the reauth_after_seconds window, and it will not trigger automatic reauthentication when a token expires or a refresh fails.
reauth_after_seconds	Maximum local login age in seconds, or NULL (default) for no forced reauthentication. Refresh does not reset this timer; indefinite_session = TRUE ignores it. When the limit is reached, local access is cleared and a new login is required. With token targets, API-token expiry alone need not end local login; this argument limits the whole authorization. SMART clients using OIDC require NULL unless indefinite_session = TRUE. See Session lifetime for provider behavior.
refresh_proactively	If TRUE, obtain a replacement access token before expiry when a refresh token is available. Default FALSE. The module schedules refresh at approximately expires_at - refresh_lead_seconds.
refresh_lead_seconds	Number of seconds before expiry to attempt proactive refresh (default: 60). An already running refresh, including one requested on demand, has this many seconds plus 5 after access-token expiry to finish before the module clears the authorization. This grace period never extends reauth_after_seconds.
refresh_check_interval_ms	Fallback interval in milliseconds for checking expiry and refresh (default 10000). Known expiry times are scheduled directly; this interval is used as a safety check or when expiry is unknown or infinite.
revoke_on_session_end	If TRUE, attempt to revoke provider tokens when the Shiny session ends or the module clears the login after expiry, the maximum login age or a refresh failure. A retained login keeps its grant. Default FALSE. Requires a provider revocation_url. Revocation is best effort and runs in the background only with async = TRUE; local cleanup still happens if the provider is unavailable.
tab_title_cleaning	If TRUE (default), removes any query string suffix from the browser tab title after the OAuth callback, so titles like "localhost:8100?code=...&state=..." become "localhost:8100"
tab_title_replacement	Optional character string to explicitly set the browser tab title after the OAuth callback. If provided, it takes precedence over tab_title_cleaning

request_uri_base_url
: Optional absolute base URL used when request_object_mode = "request_uri" publishes Request Objects through Shiny. By default (NULL), shinyOAuth derives the base URL from the current browser-visible app origin, but only when options(shinyOAuth.allowed_hosts = ...) pins the permitted public host. Set this when the authorization server must fetch the published Request Object through a different public host or proxy address than the browser uses, or when you prefer to declare the public origin explicitly. The value must use HTTPS and contain no query string or fragment. Caller-published Request Object URLs require HTTPS even when the ordinary is_ok_host() policy permits HTTP for that host (RFC 9101 Section 5.2). Wrap the app in oauth_ui(ui, id, client) or oauth_form_post_ui() to serve these URLs. Handles contain no Shiny session token and expire within 120 seconds. Shared workers require a shared state store with atomic take().

browser_cookie_path
: URL path covered by the login cookie. Default NULL uses "/", covering all app routes. An explicit path, such as "/app", must cover both the starting page and callback, start with /, and contain no semicolons or control characters. On HTTPS the path is always "/" and the cookie always uses the __Host- prefix to prevent sibling-domain cookie injection. Module identifiers isolate cookie names. Custom paths apply only to HTTP development; HTTP cannot provide this protection.

browser_cookie_samesite
: Cookie setting controlling when the browser sends the login cookie on requests from other sites. One of "Strict" (default), "Lax", or "None". "Lax" allows the cookie on top-level cross-site navigations, which some proxy arrangements require. "None" also allows cross-site cookie use in other contexts; it requires HTTPS and sets the cookie's Secure attribute.

refresh_check_interval
: Compatibility alias for refresh_check_interval_ms. Supply only one spelling.

Details

Login starts automatically by default. With auto_redirect = FALSE, auth[["request_login"]]() starts the same flow explicitly. The module publishes auth[["authenticated"]] and the accepted token after its configured checks pass.

After login, auth$connection() returns a session-bound reference to the current credentials. connection$request() calls a configured API, while connection$access_token() supplies a token for another R package. See vignette("external-integrations", package = "shinyOAuth") (online guide) for examples.

See the usage vignette for a complete app and instructions for registration, API calls, and deployment.

Value

A shiny::reactiveValues() object. If you assign it to auth, its main fields are:

- auth[["authenticated"]]: TRUE when a token is present and the configured checks have passed, otherwise FALSE. With indefinite_session = TRUE, the flag stays true while a token

is kept, including after refresh errors. With token targets, it tracks the retained authorization across individual token expiry and recoverable target failures. Connection methods check the selected token's lifetime and permissions before use; the configured authentication-age limit still applies to the whole authorization.

- `auth[["token"]]`: an OAuthToken, or `NULL` before login or after clearing the session. Read properties with `@`, for example `auth[["token"]]@userinfo`. Additional token response parameters are available in `auth[["token"]]@extra_fields`; `auth[["token"]]@initial_extra_fields` preserves the parameters from the initial code exchange across refreshes.
- `auth[["error"]]`, `auth[["error_description"]]`: the error code and available diagnostic detail. Use your own user-facing message; these fields can include sensitive provider information.
- `auth[["error_uri"]]`: an optional provider help URL. Only absolute HTTPS URLs on provider or explicitly allowed hosts are surfaced. Treat it as untrusted navigation input. `NULL` means the provider omitted the URL or supplied a value that did not pass validation.
- `auth[["token_stale"]]`: `TRUE` when a target authorization retains an expired primary token, or an indefinite session keeps an expired token or one whose refresh failed. Resets after successful login, primary-token refresh, or authorization clearing, including logout and maximum authentication age. Refreshing a secondary target does not renew the primary token.

The object also supplies:

- `auth$connection()`: the current OAuthConnection, or `NULL` before login or after the module clears the authorization. The reference and its `$id` survive refresh; logout, reauthorization and session closure invalidate it. Use `connection <- shiny::req(auth$connection())` inside reactive code, then `connection$access_token()` for a server-side SDK or database driver. Accessor-triggered refresh works without `refresh_proactively = TRUE`.
- `auth$reauthorize()`: invalidate the current local authorization and begin replacement with its retained scope limit, without upstream revocation. Local state preparation is checked first; rejection preserves current access. Failed/cancelled replacement after that does not revive old references. With no prior grant, this starts ordinary login. It does not force account selection or a password prompt; configured maximum authentication age still applies.
- `auth[["request_login"]]()`: start login. Waits for browser setup when needed and does nothing if the session is already authenticated. Uses a browser form when the client selects `authorization_method = "POST"`; the app's Content Security Policy `form-action` must permit the provider endpoint.
- `auth[["logout"]]()`: clear the local login and attempt to revoke tokens when supported, following `async`. It does not sign out of the provider account.
- `auth[["build_auth_url"]]()`: advanced helper for a custom login link. Rejects POST clients; use `request_login()` for their form submission. Creates pending login state as well as the URL, so retain the result for the link instead of rebuilding it on every UI update. Rotates and checks the browser binding before creating state. Returns a promise resolving to the URL (or `NA` on failure or an obsolete result); use `promises::then()`. PAR URLs carry `shinyOAuth.par_request_uri`, `shinyOAuth.par_expires_in`, and `shinyOAuth.par_expires_at` attributes to help you decide when to regenerate the link. `request_login()` handles these details for button-based login. A custom observer's promise handler runs after browser acknowledgment. Returning `invisible(NULL)` from that observer lets Shiny process the acknowledgment; awaiting the pending promise in the observer blocks it.

- `auth[["has_browser_token"]]()`: reports browser-binding readiness, independently of authentication state. The normal login methods manage browser setup automatically.
- `auth[["set_browser_token"]]()`: asks the browser to establish its binding when missing. The token becomes available after the browser reports it back to Shiny.
- `auth[["clear_browser_token"]]()`: clears the browser binding. `request_login()` manages cookie setup automatically, and `logout()` handles cookie rotation when ending a session.

Other fields manage the module internally and are not needed in app code.

Asynchronous execution

With `async = TRUE` and configured mirai workers or a non-sequential future plan, provider requests run outside the main R process. Synchronous network waits otherwise delay Shiny sessions sharing that process.

Browser setup

The UI wrapper handles the registered callback URL. IDE viewers and embedded dashboard previews can block the browser navigation used during login. On Posit Connect and Connect Cloud, the direct app URL provides top-level navigation; Connect Cloud exposes it under Settings > URL.

The module establishes browser binding through cookies, tab-scoped session storage and Web Crypto, and checks it on return. The advanced security vignette describes these browser boundaries and their deployment implications.

Session lifetime

Token refresh does not extend the configured maximum login age. A new login after that age expires requests the client's configured permissions again; explicit `auth$reauthorize()` keeps any previously reduced permissions.

For OIDC, age-based reauthentication requests `max_age = 0` and requires a valid `auth_time` in the returned identity token. OAuth-only providers cannot guarantee an active provider login, so the age limit ends local access and starts ordinary authorization. With token targets, `authenticated` describes the retained login, while connection methods check the selected API token.

Advanced worker behavior

Signed-request preparation, pushed authorization requests (PAR) and query JARM verification can run in workers. State-store access and publication of hosted Request Objects stay in the main process. Setup-time discovery, standalone `prepare_call()` and JARM verification in `oauth_form_post_ui()` remain synchronous. Configure timeouts for those operations; see the package options reference.

See Also

`oauth_ui()`, `oauth_client()`, OAuthToken, `oauth_form_post_ui()`

Examples

```
# Register http://127.0.0.1:8100 as the GitHub OAuth App callback URL.
if (
  # Example requires configured GitHub OAuth 2.0 app
  # (go to https://github.com/settings/developers to create one):
  nzchar(Sys.getenv("GITHUB_OAUTH_CLIENT_ID")) &&
    nzchar(Sys.getenv("GITHUB_OAUTH_CLIENT_SECRET")) &&
    interactive()
) {
  library(shiny)
  library(shinyOAuth)

  # Define client
  client <- oauth_client(
    provider = oauth_provider_github(),
    client_id = Sys.getenv("GITHUB_OAUTH_CLIENT_ID"),
    client_secret = Sys.getenv("GITHUB_OAUTH_CLIENT_SECRET"),
    redirect_uri = "http://127.0.0.1:8100",
    scopes = c("read:user", "user:email")
  )

  # Choose which app you want to run
  app_to_run <- NULL
  while (!isTRUE(app_to_run %in% c(1:4))) {
    app_to_run <- readline(
      prompt = paste0(
        "Which example app do you want to run?\n",
        "  1: Auto-redirect login\n",
        "  2: Manual login button\n",
        "  3: Fetch additional resource with access token\n",
        "  4: No app (all will be defined but none run)\n",
        "Enter 1, 2, 3, or 4... "
      )
    )
  }

  if (app_to_run %in% c(1:3)) {
    cli::cli_alert_info(paste0(
      "Will run example app {app_to_run} on {.url http://127.0.0.1:8100}\n",
      "Open this URL in a regular browser (viewers in RStudio/Positron/etc. ",
      "cannot perform necessary redirects)"
    ))
  }

  # Example app with auto-redirect (1) ------------------------------------------

  ui_1 <- oauth_ui(
    fluidPage(
      uiOutput("login")
    ),
    id = "auth",
    client = client
```

```
)

server_1 <- function(input, output, session) {
  # Auto-redirect (default):
  auth <- oauth_module_server(
    "auth",
    client,
    auto_redirect = TRUE
  )

  output[["login"]] <- renderUI({
    if (auth[["authenticated"]]) {
      user_info <- auth[["token"]]@userinfo
      tagList(
        tags[["p"]]("You are logged in!"),
        tags[["pre"]](paste(capture.output(str(user_info)), collapse = "\n"))
      )
    } else {
      tags[["p"]]("You are not logged in.")
    }
  })
}

app_1 <- shinyApp(ui_1, server_1)
if (app_to_run == "1") {
  runApp(
    app_1,
    port = 8100,
    launch.browser = FALSE
  )
}

# Example app with manual login button (2) ----------------------------------

ui_2 <- oauth_ui(
  fluidPage(
    actionButton("login_btn", "Login"),
    actionButton("logout_btn", "Logout"),
    uiOutput("login")
  ),
  id = "auth",
  client = client
)

server_2 <- function(input, output, session) {
  auth <- oauth_module_server(
    "auth",
    client,
    auto_redirect = FALSE
  )

  observeEvent(input[["login_btn"]], {
    auth[["request_login"]]()
```

```
  })
  observeEvent(input[["logout_btn"]], {
    auth[["logout"]]()
  })

  output[["login"]] <- renderUI({
    if (auth[["authenticated"]]) {
      user_info <- auth[["token"]]@userinfo
      tagList(
        tags[["p"]]("You are logged in!"),
        tags[["pre"]](paste(capture.output(str(user_info)), collapse = "\n"))
      )
    } else {
      tags[["p"]]("You are not logged in.")
    }
  })
}

app_2 <- shinyApp(ui_2, server_2)
if (app_to_run == "2") {
  runApp(
    app_2,
    port = 8100,
    launch.browser = FALSE
  )
}

# Example app requesting additional resource with access token (3) -----------

# Below app shows the authenticated username + their GitHub repositories,
# fetched via GitHub API using the access token obtained during login

ui_3 <- oauth_ui(
  fluidPage(
    uiOutput("ui")
  ),
  id = "auth",
  client = client
)

server_3 <- function(input, output, session) {
  auth <- oauth_module_server(
    "auth",
    client,
    auto_redirect = TRUE
  )

  repositories <- reactiveVal(NULL)
  repository_error <- reactiveVal(FALSE)

  observe({
    req(auth[["authenticated"]])
```

```
  # Example additional API request using the access token
  # (e.g., fetch user repositories from GitHub)
  # This loads one page; use the API's pagination for further results.
  repos_data <- tryCatch(
    {
      resp <- perform_resource_req(
        auth[["token"]],
        "https://api.github.com/user/repos",
        query = list(per_page = 30)
      )
      httr2::resp_check_status(resp)
      httr2::resp_body_json(resp, simplifyVector = TRUE)
    },
    error = function(e) NULL
  )

  repository_error(is.null(repos_data))
  repositories(repos_data)
})

# Render username + their repositories
output[["ui"]] <- renderUI({
  if (isTRUE(auth[["authenticated"]])) {
    user_info <- auth[["token"]]@userinfo
    repos <- repositories()

    return(tagList(
      tags[["p"]](paste("You are logged in as:", user_info[["login"]])),
      tags[["h4"]]("Your repositories:"),
      if (repository_error()) {
        tags[["p"]]("Could not load repositories.")
      } else if (!is.null(repos) && length(repos) == 0) {
        tags[["p"]]("No repositories returned.")
      } else if (!is.null(repos)) {
        tags[["ul"]](
          Map(
            function(url, name) {
              # Render names as text; accept only GitHub HTTPS links.
              if (isTRUE(grepl("^https://github\\.com/", url))) {
                tags[["li"]](tags[["a"]](
                  href = url,
                  target = "_blank",
                  rel = "noopener noreferrer",
                  name
                ))
              } else {
                tags[["li"]](name)
              }
            },
            repos[["html_url"]],
            repos[["full_name"]]
          )
        )
```

```
          } else {
            tags[["p"]]("Loading repositories...")
          }
        ))
      }

      return(tags[["p"]]("You are not logged in."))
    })
  }

  app_3 <- shinyApp(ui_3, server_3)
  if (app_to_run == "3") {
    runApp(
      app_3,
      port = 8100,
      launch.browser = FALSE
    )
  }
}
```

---

oauth_provider *Configure OAuth/OIDC provider endpoints and validation settings*

Description

Configure a service from its documented endpoint URLs and protocol settings when no named provider helper fits or OIDC discovery is unavailable. Pass the resulting provider to oauth_client(). For a supported service, its named helper is an easier starting point; for OIDC, oauth_provider_oidc_discover() can look up the settings.

Usage

```
oauth_provider(
  name,
  auth_url,
  token_url,
  userinfo_url = NA_character_,
  introspection_url = NA_character_,
  revocation_url = NA_character_,
  par_url = NA_character_,
  par_required = FALSE,
  authorization_request_front_channel_mode = "compat",
  request_object_signing_alg_values_supported = character(),
  request_object_encryption_alg_values_supported = character(),
  request_object_encryption_enc_values_supported = character(),
  request_object_encryption_jwk = NULL,
  signed_request_object_required = FALSE,
  request_parameter_supported = NA,
```

```
  request_uri_parameter_supported = NA,
  request_uri_registration_required = NA,
  token_endpoint_auth_signing_alg_values_supported = character(),
  dpop_signing_alg_values_supported = character(),
  authorization_response_iss_parameter_supported = FALSE,
  response_modes_supported = character(),
  mtls_endpoint_aliases = list(),
  mtls_client_certificate_bound_access_tokens = FALSE,
  issuer = NA_character_,
  issuer_match = "url",
  use_nonce = NULL,
  use_pkce = TRUE,
  pkce_method = "S256",
  userinfo_required = NULL,
  userinfo_id_token_match = NULL,
  userinfo_signed_jwt_required = FALSE,
  userinfo_id_selector = function(userinfo) {
     userinfo[["sub"]]
 },
  id_token_required = NULL,
  id_token_validation = NULL,
  extra_auth_params = list(),
  extra_token_params = list(),
  extra_token_headers = character(),
  token_auth_style = "header",
  jwks_cache = NULL,
  jwks_pins = character(),
  jwks_pin_mode = "any",
  jwks_host_issuer_match = NULL,
  jwks_host_allow_only = NULL,
 id_token_allowed_algs = c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512",
    "Ed25519", "EdDSA"),
  allowed_token_types = c("Bearer"),
  leeway = getOption("shinyOAuth.leeway", 30),
  id_token_at_hash_required = FALSE,
  infer_oidc_from_issuer = TRUE,
  jwks_uri = NA_character_,
  userinfo_allowed_algs = NULL,
  jarm_signing_alg_values_supported = character(),
  jarm_encryption_alg_values_supported = character(),
  jarm_encryption_enc_values_supported = character(),
  jarm_tolerate_duplicate_top_level_iss = FALSE,
  endpoint_auth_metadata = list(),
  ...,
  allowed_algs = NULL,
  allow_missing_token_type = FALSE,
  token_target_mode = "none",
  token_response_scope_format = "space",
```

```
  userinfo_id_selector_policy_id = NA_character_
)
```

Arguments

`name`	Provider name (e.g., "github", "google"). Cosmetic only; used in logging and audit events
`auth_url`	URL of the provider's login and permission page.
`token_url`	URL where R exchanges the returned code for tokens.
`userinfo_url`	User info endpoint URL (optional)
`introspection_url`	Optional URL where the provider can confirm whether a token is still active (RFC 7662).
`revocation_url`	Optional URL where the app can ask the provider to invalidate a token, for example during logout (RFC 7009).
`par_url`	Optional Pushed Authorization Request (PAR) URL (RFC 9126). When set, shinyOAuth first sends the authorization request from server to provider and then redirects the browser with the returned `request_uri` handle instead of the full request payload. Use PAR to keep most request details out of the browser URL, submit large requests, or meet a provider's PAR requirement. The provider must support this endpoint.
`par_required`	Logical. Whether the provider requires authorization requests to be sent via PAR. When `TRUE`, `par_url` must also be configured.
`authorization_request_front_channel_mode`	Character scalar controlling which browser-visible outer parameters shinyOAuth keeps when the actual authorization request is carried by JAR or PAR. Use `"compat"` (default) to keep OIDC-compatible parameters with outer `client_id`, `response_type`, and `scope` when an issuer is configured. Use `"minimal"` for plain OAuth browser redirects and for PAR deployments whose authorization endpoint accepts only `client_id` plus the provider-issued `request_uri` handle. OpenID Connect by-value `request` and caller-managed `request_uri` transports reject `"minimal"` because OIDC still requires outer `response_type` and an outer `scope` containing `openid`.
`request_object_signing_alg_values_supported`	Optional vector of JWS algorithms that the provider advertises for signed Request Objects (RFC 9101). This is mainly used for early validation when an OAuthClient sends `request_object_mode = "request"` or `request_object_mode = "request_uri"`.
`request_object_encryption_alg_values_supported`	Optional vector of JWE key-management algorithms that the provider advertises for encrypted Request Objects. This metadata is used for early validation when an OAuthClient enables Request Object encryption.
`request_object_encryption_enc_values_supported`	Optional vector of JWE content-encryption algorithms that the provider advertises for encrypted Request Objects. This metadata is used for early validation when an OAuthClient enables Request Object encryption.

request_object_encryption_jwk
: Optional explicit recipient public key used to encrypt Request Objects when discovery-backed JWKS selection is not available or when you need to pin one specific encryption key. Accepts an OpenSSL public key, a PEM public-key string, a parsed JWK object, or a JWK JSON string.

signed_request_object_required
: Logical. Whether the provider requires signed Request Objects for authorization requests. When TRUE, client construction requires request_object_mode = "request" or "request_uri". This represents the provider's registration policy and enforces local construction. Provider-side rejection of unsigned requests depends on that registration, not on this local flag.

request_parameter_supported
: Logical or NA. Whether discovery metadata explicitly advertises support for the authorization-request request parameter. NA means the provider did not say. Discovery-derived providers apply the OpenID Connect default (FALSE) when this metadata is omitted.

request_uri_parameter_supported
: Logical or NA. Whether discovery metadata explicitly advertises support for the authorization-request request_uri parameter for caller-managed request URIs. NA means the provider did not say. Discovery-derived providers apply the OpenID Connect default (TRUE) when this metadata is omitted. PAR-issued request_uri handles remain valid even when this metadata is FALSE.

request_uri_registration_required
: Logical or NA. Whether discovery metadata says caller-managed request_uri values must be pre-registered. NA means the provider did not say. Discovery-derived providers apply the OpenID Connect default (FALSE) when this metadata is omitted. shinyOAuth can publish caller-managed request_uri values through oauth_module_server(). TRUE means the public URI or a matching prefix belongs in the provider's client registration. The package stores this requirement as metadata; provider-side registration is outside local validation.

token_endpoint_auth_signing_alg_values_supported
: Optional vector of JWS algorithms that the provider advertises for JWT-based client authentication (client_secret_jwt / private_key_jwt) at the token endpoint. This metadata is used for early validation of OAuthClient@client_assertion_alg and inferred JWT client-assertion defaults.

dpop_signing_alg_values_supported
: Optional vector of JWS algorithms that the provider advertises for DPoP proof JWTs (RFC 9449). This metadata is used for early validation of OAuthClient@dpop_signing_alg and inferred outbound DPoP signing defaults.

authorization_response_iss_parameter_supported
: Logical. Whether the provider advertises RFC 9207 support for returning an iss parameter on the authorization response. When TRUE, the oauth_client() helper can auto-enable callback issuer enforcement when the caller leaves enforce_callback_issuer unset and the provider also has a configured issuer.

response_modes_supported
: Optional character vector of OAuth/OIDC response_mode values advertised by the provider. Discovery-backed providers use the discovery metadata value, defaulting to c("query", "fragment") when omitted per OIDC Discovery/RFC

8414. Generic providers may leave this empty when capabilities are not known. Provider metadata may include response modes that shinyOAuth does not implement. Requesting an unsupported mode raises an error.

`mtls_endpoint_aliases`
: Optional named list of RFC 8705 mTLS endpoint aliases. Names should follow the metadata keys such as `token_endpoint`, `userinfo_endpoint`, `introspection_endpoint`, `revocation_endpoint`, `par_endpoint`, or `pushed_authorization_request_endpoint`, and values must be absolute URLs. This is an advanced setting used when a provider publishes separate mTLS-specific endpoints.

`mtls_client_certificate_bound_access_tokens`
: Logical. Whether the authorization server advertises RFC 8705 capability to issue certificate-bound access tokens. This describes server capability; the client still has to opt into mTLS separately. When `TRUE`, token responses may include a `cnf` claim with an `x5t#S256` thumbprint that downstream requests must match with the same certificate.

`issuer`
: Optional authorization-server issuer URL. You need this for issuer validation and features such as ID-token validation. shinyOAuth uses it to verify issuer claims and locate signing keys (JWKS), typically through an OIDC discovery document.

`issuer_match`
: Character scalar controlling how strictly the discovery document's `issuer` is validated against `issuer` when it later performs runtime discovery to locate the JWKS URI.

  - `"url"` (default): require the issuer used for discovery to match the discovery metadata exactly, including any trailing slash.
  - `"host"`: compare only scheme + host.
  - `"none"`: do not validate discovery issuer consistency.

  The Microsoft helper configures tenant-independent issuer handling for supported alias tenants.

`use_nonce`
: Whether to tie the ID token to this login using a random nonce. The package sends it in the request and checks it in the returned ID token.

`use_pkce`
: Whether to protect the code exchange using Proof Key for Code Exchange (PKCE), which public clients require. It sends a `code_challenge` with the login request and a matching secret `code_verifier` during token exchange.

`pkce_method`
: PKCE code challenge method: `"S256"` (default), or `"plain"` for compatibility with providers that require it.

`userinfo_required`
: Whether to fetch a user profile after token exchange. The result is stored in `token@userinfo`; a failed required fetch stops login. In `oauth_provider()`, this defaults to `TRUE` when `userinfo_url` is supplied and `FALSE` otherwise.

`userinfo_id_token_match`
: Require the fetched profile to belong to the validated login identity. Defaults to `TRUE` when a UserInfo URL and ID-token or nonce validation are enabled, including for manual `get_userinfo()` calls. Pass the complete OAuthToken for this check. An explicit `FALSE` permits profile retrieval without a validated identity; when both are available, their actual `sub` values must still match.

`userinfo_signed_jwt_required`
: Require a profile signed by the provider. Default `FALSE` accepts ordinary JSON profiles. When `TRUE`, configure a UserInfo URL and issuer, and arrange signed responses with the provider. Signatures use `userinfo_allowed_algs`; unsigned, HMAC-signed and encrypted profiles are not accepted by normal configuration. This requirement also applies to manual retrieval when `userinfo_required = FALSE`.

`userinfo_id_selector`
: A function that extracts the user ID from the userinfo response. Should take a single argument (the userinfo list) and return the user ID as a string.

  This is used for helpers that need a provider-specific application user identifier, such as audit fields. It does not replace OIDC subject binding: when a validated ID token and UserInfo are both available, their actual `sub` claims are always compared. Helper constructors like `oauth_provider()` and `oauth_provider_oidc()` provide a default selector that extracts `sub`.

`id_token_required`
: Whether to require an ID token to be returned during token exchange. If no ID token is returned, the token exchange will fail. This only makes sense for OpenID Connect providers and may require the client's scope to include `openid`.

  Both the S7 constructor and `oauth_provider()` enable this when an issuer is supplied and `infer_oidc_from_issuer = TRUE`. Pure OAuth 2.0 providers keep this disabled by default.

`id_token_validation`
: Whether to perform ID token validation after token exchange. This requires the provider to be a valid OpenID Connect provider with a configured `issuer` and the token response to include an ID token (may require setting the client's scope to include `openid`).

  Both the S7 constructor and `oauth_provider()` enable this when an issuer is provided and `infer_oidc_from_issuer = TRUE`. An explicit `FALSE` disables automatic ID token validation.

`extra_auth_params`
: Extra parameters for authorization URL

`extra_token_params`
: Extra parameters for token exchange. `scope` is reserved and cannot be unblocked. For explicit refresh scope narrowing use a managed connection's `[["refresh"]](scopes = ...` Configure login scopes on `oauth_client()` instead.

`extra_token_headers`
: Extra headers for back-channel token-style requests (named character vector), applied only to token exchange and refresh. Configure `oauth_client(endpoint_auth = ...)` for headers needed by PAR, introspection, or revocation.

`token_auth_style`
: How the client authenticates at the token endpoint. One of:

  - "header": HTTP Basic (client_secret_basic)
  - "body": Form body (client_secret_post)
  - "public": Public-client form body (none in discovery metadata); sends `client_id` but never `client_secret`, even if one is configured. The alias `"none"` is also accepted.

- "tls_client_auth": RFC 8705 mutual TLS client authentication using a client certificate chained to a trusted CA
- "self_signed_tls_client_auth": RFC 8705 mutual TLS client authentication using a self-signed client certificate registered out of band with the provider
- "client_secret_jwt": JWT client assertion signed with HMAC using client_secret (RFC 7523)
- "private_key_jwt": JWT client assertion signed with an asymmetric key (RFC 7523)

`jwks_cache` Storage for the provider's public signing keys. Defaults to `cachem::cache_mem(max_age = 3600)`, an in-memory cache lasting one hour. A `custom_cache()` can share keys across processes. Shorter lifetimes pick up changed keys sooner; longer lifetimes reduce network requests. HTTP cache directives can shorten this lifetime. Responses marked `no-store` are not retained, and `no-cache` responses are fetched again before reuse. Advertised freshness also accounts for `Age` and `Expires`. The package also attempts a rate-limited refresh when a key is missing or no longer verifies a signature. For async login and token operations, custom caches use synchronous fallback unless they explicitly declare worker safety; see `custom_cache()` and its `worker_safe` argument.

`jwks_pins` Optional character vector of RFC 7638 JWK thumbprints (base64url) to pin against. If non-empty, fetched JWKS must contain keys whose thumbprints match these values depending on `jwks_pin_mode`. This is an advanced hardening option that lets you pre-authorize expected keys. Only keys matching a configured pin are eligible for signature verification or Request Object encryption; `jwks_pin_mode` controls whether the surrounding JWK Set may also contain unpinned keys.

`jwks_pin_mode` Pinning policy when `jwks_pins` is provided. Either "any" (default; at least one key in JWKS must match) or "all" (every RSA/EC/OKP public key in JWKS must match one of the configured pins)

`jwks_host_issuer_match`
When TRUE, enforce that the discovery `jwks_uri` host matches the issuer host exactly. Defaults to FALSE at the class level, but helper constructors for OIDC (e.g., `oauth_provider_oidc()` and `oauth_provider_oidc_discover()`) enable this by default. The generic helper `oauth_provider()` will also automatically set this to TRUE when an `issuer` is provided and either `id_token_validation` or `id_token_required` is TRUE (OIDC-like configuration). An explicit FALSE disables issuer-host matching. `jwks_host_allow_only` takes precedence and can specify a different exact hostname, as used by the Google helper.

`jwks_host_allow_only`
Optional explicit hostname that the jwks_uri must match. When provided, jwks_uri host must equal this value (exact match). You can pass either just the host (e.g., "www.googleapis.com") or a full URL; only the host component will be used. If you need to include a port or an IPv6 literal, pass a full URL (e.g., `https://[::1]:8443`) - the port is ignored and only the hostname part is used for matching. Takes precedence over `jwks_host_issuer_match`.

`id_token_allowed_algs`
Signing algorithms accepted for ID tokens. Defaults to `c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512", "Ed25519", "EdDSA")`; Ed448 and un-

supported algorithms are rejected. Use a narrower set to match your provider registration. HMAC (HS256, HS384, HS512) requires a server-only client secret, options(shinyOAuth.allow_hs = TRUE), and exactly one HS algorithm. Reused keys or secrets must not have conflicting algorithm settings. For RSA key metadata, defaults and rotation, see signing-key validation.

allowed_token_types
: Character vector of acceptable OAuth token types returned by the token endpoint (case-insensitive). Successful token responses must include token_type by default; when allowed_token_types is non-empty, its value must also be one of the allowed values or the flow fails fast with a shinyOAuth_token_error. The oauth_provider() helper defaults to c("Bearer"). When the OAuthClient is configured with dpop_private_key, shinyOAuth also accepts token_type = "DPoP" and uses DPoP proofs on supported token and downstream requests. Other non-Bearer token types (for example MAC) are rejected. Set allowed_token_types = character() explicitly only to disable the value allowlist while still requiring token_type itself.

leeway
: Clock skew leeway (seconds) applied to ID token exp/iat/nbf checks and state payload issued_at future check. Default 30. Can be globally overridden via option shinyOAuth.leeway.

id_token_at_hash_required
: Whether to require the at_hash (Access Token hash) claim in the ID token. When TRUE, login fails if the ID token does not contain an at_hash claim or if the claim does not match the access token. When FALSE (default), at_hash is validated only when present. Requires id_token_validation = TRUE.

infer_oidc_from_issuer
: Whether setting issuer enables OpenID Connect behavior. Default TRUE: helpers enable OIDC nonce/ID token defaults and the client adds the openid scope. Set FALSE for an OAuth-only server that has an issuer identifier but does not implement OIDC.

jwks_uri
: Optional URL of the provider's public signing keys (JWKS). Normally these are located through OIDC discovery. Set this for manual key configuration, including OAuth-only JARM providers.

userinfo_allowed_algs
: Optional signing algorithm allowlist for UserInfo JWTs. NULL inherits id_token_allowed_algs for manually configured providers. Discovery negotiates this independently against UserInfo metadata. Use a single algorithm to enforce the client's registered UserInfo signing choice. An empty vector rejects all signed UserInfo algorithms. Unlabelled RSA keys follow the same binding policy as id_token_allowed_algs.

jarm_signing_alg_values_supported
: Optional vector of JWS algorithms that the provider advertises for signed JWT Secured Authorization Responses (JARM).

jarm_encryption_alg_values_supported
: Optional vector of JWE key-management algorithms that the provider advertises for encrypted JARM responses.

jarm_encryption_enc_values_supported
: Optional vector of JWE content-encryption algorithms that the provider advertises for encrypted JARM responses.

`jarm_tolerate_duplicate_top_level_iss`
: Logical. Whether shinyOAuth should tolerate repeated identical top-level `iss` members in signed JARM payloads for this provider. This is a compatibility setting for providers that emit duplicate identical top-level `iss` claims. When `TRUE`, shinyOAuth collapses repeated identical top-level `iss` members before duplicate-member rejection. Conflicting duplicates and nested duplicate `iss` members are rejected. Defaults to `FALSE`.

`endpoint_auth_metadata`
: Named list of independent `introspection` and `revocation` authentication metadata. Each entry has `methods` and `signing_algs` character vectors (or `NULL` for omitted metadata). Discovery retains these fields and applies the RFC 8414 Basic-auth default for omitted revocation methods. Omitted introspection methods have no default.

`...`
: Deprecated renamed arguments accepted temporarily for backward compatibility.

`allowed_algs`
: Compatibility alias for `id_token_allowed_algs`. Supply only one spelling.

`allow_missing_token_type`
: Logical, default `FALSE`. Compatibility for providers that issue Bearer tokens without the OAuth-required `token_type`. When `TRUE`, login and refresh assume `"Bearer"` only when the field is absent. Explicit null, empty, invalid, or unsupported values still fail validation. The fallback never applies to clients configured with DPoP.

`token_target_mode`
: How this provider obtains API-specific tokens: `"none"` (default), `"rfc8707"`, or `"microsoft"`. This applies to client `token_targets` and depends on provider support. The Microsoft helper selects `"microsoft"` automatically; discovery does not enable targets. See `vignette("token-targets", package = "shinyOAuth")` (online guide) for setup and permission rules.

`token_response_scope_format`
: Separator used for permissions in token responses: `"space"` (default) or `"comma"`. The GitHub helper sets `"comma"` automatically. Set it explicitly for a custom provider using that format.

`userinfo_id_selector_policy_id`
: Optional version name for a custom `userinfo_id_selector`, or `NA_character_`. Identifies settings or helper behavior outside the function body that affect identity selection. These form part of the client's policy: a new version ID lets pending logins and saved connections detect a change.

Details

Supply `name`, `auth_url`, and `token_url` to start. Add `userinfo_url` to fetch profiles. Supplying `issuer` enables OIDC defaults, including ID token validation, unless `infer_oidc_from_issuer = FALSE`. Advanced arguments must match your provider's capabilities; see the advanced security vignette.

Value

OAuthProvider object

Examples

```
# Configure generic OAuth 2.0 provider (no OIDC)
generic_provider <- oauth_provider(
  name = "example",
  auth_url = "https://example.com/oauth/authorize",
  token_url = "https://example.com/oauth/token",
  # Optional URL for fetching user info:
  userinfo_url = "https://example.com/oauth/userinfo"
)

# Configure generic OIDC provider manually
# (This defaults to using nonce & ID token validation)
generic_oidc_provider <- oauth_provider_oidc(
  name = "My OIDC",
  base_url = "https://my-issuer.example.com"
)

# Look up an OIDC provider's settings with discovery
# (requires network access)
if (interactive()) {
  # Using Auth0 sample issuer as an example
  oidc_discovery_provider <- oauth_provider_oidc_discover(
    issuer = "https://samples.auth0.com/"
  )
}

# GitHub preconfigured provider
github_provider <- oauth_provider_github()

# Google preconfigured provider
google_provider <- oauth_provider_google()

# Microsoft preconfigured provider
# For a complete app using a custom tenant ID, see:
# https://lukakoning.github.io/shinyOAuth/reference/oauth_provider_microsoft.html

# Spotify preconfigured provider
spotify_provider <- oauth_provider_spotify()

# Slack via OIDC discovery
# (requires network access)
if (interactive()) {
  slack_provider <- oauth_provider_slack()
}

# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
if (interactive()) {
  options(shinyOAuth.allow_insecure_oidc_loopback = TRUE)
  oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")
}
```

```
# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_auth0(domain = "your-tenant.auth0.com")
}

# Okta
# (requires configured Okta domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_okta(domain = "dev-123456.okta.com")
}
```

`oauth_provider_apple` *Create an Apple OAuthProvider*

Description

Look up Apple's login settings and return an OAuthProvider for use with `oauth_client()`. This helper makes an OIDC discovery request during setup.

Usage

```
oauth_provider_apple(name = "apple")
```

Arguments

`name`	Optional provider name (default "apple")

Details

Configure your client with:

- Your Apple Services ID or App ID as `client_id`.
- A client secret created with `oauth_client_secret_apple()` using your Apple developer key.
- An HTTPS return address with a domain name; Apple does not accept localhost or IP addresses.
- `response_mode = "form_post"` and `oauth_form_post_ui()` when requesting `email` or `name`.

Identity information comes from the validated ID token's claims. Apple has no UserInfo endpoint, so `userinfo_required` is `FALSE`. The one-time `user` payload that Apple may send with a form POST callback is not mapped into `token@userinfo`. Apple's documented `email_verified` strings (`"true"` and `"false"`) are normalized to logical values in validated claims after signature and issuer verification. Other providers retain the standard JSON Boolean requirement.

Value

OAuthProvider object configured for Sign in with Apple

Examples

```
# Sign in with Apple requires an Apple Services ID, Team ID, key ID, and the
# corresponding P-256 private key. Network access is required for discovery.
if (interactive()) {
  apple_provider <- oauth_provider_apple()

  apple_secret <- oauth_client_secret_apple(
    client_id = "com.example.web",
    team_id = "ABCDEFGHIJ",
    key_id = "ABC123DEFG",
    private_key = openssl::read_key("AuthKey_ABC123DEFG.p8")
  )

  apple_client <- oauth_client(
    provider = apple_provider,
    client_id = "com.example.web",
    client_secret = apple_secret,
    redirect_uri = "https://example.com/oauth/callback",
    scopes = c("openid", "email", "name"),
    response_mode = "form_post"
  )
}
```

---

oauth_provider_auth0 *Create an Auth0 OAuthProvider (via OIDC discovery)*

Description

Look up login settings for your Auth0 domain. Pass the result to `oauth_client()` with your registered app credentials. This helper makes a discovery request during setup.

Usage

```
oauth_provider_auth0(domain, name = "auth0", audience = NULL)
```

Arguments

`domain`	Your Auth0 domain, e.g., "your-domain.auth0.com"
`name`	Optional provider name (default "auth0")
`audience`	Optional audience value to send in authorization requests.

Value

OAuthProvider object configured for the specified Auth0 domain

Examples

```
## Not run:
oauth_provider_auth0("your-domain.auth0.com")

## End(Not run)
```

`oauth_provider_authentik` *Configure sign-in with authentik*

Description

Configure login for an authentik OAuth2/OIDC application. Supply the instance URL and application slug, then pass the registered credentials to `oauth_client()` with the scopes configured in authentik.

Usage

```
oauth_provider_authentik(
  base_url,
  application_slug,
  name = "authentik",
  token_auth_style = NULL
)
```

Arguments

`base_url`	authentik instance URL, for example `"https://auth.example.com"`.
`application_slug`	Application slug configured in authentik (not the display name or client ID).
`name`	Optional provider name (default `"authentik"`).
`token_auth_style`	Authentication style for token requests: "header" (client_secret_basic), "body" (client_secret_post), or "public" (public client; send `client_id` only). The alias `"none"` is also accepted for `"public"`. If NULL (default), it is inferred conservatively from discovery: `"header"` (client_secret_basic) is preferred, followed by `"body"` (client_secret_post), then `"public"` if none is advertised and PKCE is enabled. Set `token_auth_style = "public"` explicitly for a public client registration. JWT methods (`"client_secret_jwt"`, `"private_key_jwt"`) and mTLS methods (`"tls_client_auth"`, `"self_signed_tls_client_auth"`) must be selected explicitly. See `oauth_provider()` for the supported methods and their credentials.

Details

Uses authentik's default per-application issuer mode and preserves its trailing slash. Global issuer mode is not supported by this preset; configure the endpoints explicitly with `oauth_provider()` for that mode. For refresh tokens, both request `"offline_access"` and enable the corresponding scope mapping in authentik. Configure a signing key that supports RS256 for OIDC discovery. See authentik's provider guide.

Value

OAuthProvider object configured for the authentik application.

Examples

```
## Not run:
oauth_provider_authentik("https://auth.example.com", "shiny-app")

## End(Not run)
```

---

oauth_provider_cognito

Configure sign-in with an Amazon Cognito user pool

Description

Let users sign in through your Amazon Cognito user pool. Register an app client, enable the authorization code grant and a user pool login domain, then pass this provider and the app credentials to `oauth_client()`. Request the login and profile scopes enabled for that registration.

Usage

```
oauth_provider_cognito(issuer, name = "cognito", token_auth_style = NULL)
```

Arguments

`issuer`	Exact user pool issuer URL from AWS, for example `"https://cognito-idp.eu-west-1.amazonaws.co`
`name`	Optional provider name (default `"cognito"`).
`token_auth_style`	Authentication style for token requests: "header" (client_secret_basic), "body" (client_secret_post), or "public" (public client; send `client_id` only). The alias `"none"` is also accepted for `"public"`. If NULL (default), it is inferred conservatively from discovery: `"header"` (client_secret_basic) is preferred, followed by `"body"` (client_secret_post), then `"public"` if none is advertised and PKCE is enabled. Set `token_auth_style = "public"` explicitly for a public client registration. JWT methods (`"client_secret_jwt"`, `"private_key_jwt"`) and mTLS methods (`"tls_client_auth"`, `"self_signed_tls_client_auth"`) must be selected explicitly. See `oauth_provider()` for the supported methods and their credentials.

Details

Use the exact user pool issuer, not the managed-login or custom domain. Discovery obtains authorization, token, and UserInfo URLs on that login domain while retaining the pool issuer for ID token validation. Both the original `cognito-idp` and updated `issuer-cognito-idp` issuer forms are supported, as are AWS partition-specific hostnames. No AWS credentials are needed for discovery.

See Cognito endpoints.

Value

OAuthProvider object configured for a Cognito user pool.

Examples

```
## Not run:
oauth_provider_cognito(
  "https://cognito-idp.eu-west-1.amazonaws.com/eu-west-1_Example"
)

## End(Not run)
```

`oauth_provider_github` *Create a GitHub OAuthProvider*

Description

Create the provider configuration for a GitHub OAuth App, then pass it with your app credentials to `oauth_client()`. This configures profile retrieval from GitHub's API; GitHub does not return an OIDC ID token. The helper handles GitHub's permission-response format automatically.

Usage

```
oauth_provider_github(name = "github")
```

Arguments

`name`	Optional provider name (default "github")

Details

You can register a new GitHub OAuth 2.0 app in your OAuth App settings.

Value

OAuthProvider object for use with a GitHub OAuth 2.0 app

Examples

```
oauth_provider_github()
```

`oauth_provider_gitlab` *Configure sign-in with GitLab*

Description

Configure OIDC sign-in with GitLab.com or a self-managed GitLab instance. Register an OAuth application on that instance, then pass its credentials to `oauth_client()` with `scopes = c("openid", "profile", "email")`.

Usage

```
oauth_provider_gitlab(
  base_url = "https://gitlab.com",
  name = "gitlab",
  token_auth_style = NULL
)
```

Arguments

`base_url`	GitLab instance URL, including HTTPS and any deployment subpath. Defaults to `"https://gitlab.com"`.
`name`	Optional provider name (default `"gitlab"`).
`token_auth_style`	Authentication style for token requests: "header" (client_secret_basic), "body" (client_secret_post), or "public" (public client; send `client_id` only). The alias `"none"` is also accepted for `"public"`. If NULL (default), it is inferred conservatively from discovery: `"header"` (client_secret_basic) is preferred, followed by `"body"` (client_secret_post), then `"public"` if none is advertised and PKCE is enabled. Set `token_auth_style = "public"` explicitly for a public client registration. JWT methods (`"client_secret_jwt"`, `"private_key_jwt"`) and mTLS methods (`"tls_client_auth"`, `"self_signed_tls_client_auth"`) must be selected explicitly. See `oauth_provider()` for the supported methods and their credentials.

Details

The `"openid"` scope requests an ID token. API and repository access require separate GitLab scopes. See GitLab's OIDC guide.

Value

OAuthProvider object configured for GitLab.

Examples

```
## Not run:
oauth_provider_gitlab()
oauth_provider_gitlab("https://gitlab.example.edu")

## End(Not run)
```

---

oauth_provider_globus *Configure sign-in with Globus Auth*

Description

Let users sign in with Globus Auth. Register a web application with a client secret at https://app.globus.org/settings/developers, then pass this provider and the credentials to oauth_client() with scopes = c("openid", "profile", "email").

Usage

```
oauth_provider_globus(name = "globus")
```

Arguments

name	Optional provider name (default "globus").

Details

This helper supports OIDC login. Globus API grants can return separate credentials for several resource servers in other_tokens. Those fields are preserved in token@extra_fields, but shinyOAuth does not automatically select, refresh, or revoke the nested tokens. The supported login scopes are openid, profile, email and optionally offline_access. Multi-resource API authorization needs a Globus-aware integration that manages each resource server's separate token.

Value

OAuthProvider object configured for Globus Auth login.

Advanced provider settings

The preset uses Globus's published endpoints and RS512 identity-token signatures. It does not use generic discovery, which requires advertised RS256 support.

See the Globus Auth guide.

Examples

```
oauth_provider_globus()
```

oauth_provider_google *Create a Google OAuthProvider*

Description

Use your Google app registration with `oauth_client()` to add Google sign-in. The helper configures OIDC validation and profile retrieval.

Usage

```
oauth_provider_google(name = "google")
```

Arguments

`name`	Optional provider name (default "google")

Details

You can register a new Google OAuth 2.0 app in the Google Cloud Console.

This preset uses a restricted Google OIDC profile: ID tokens must have `iss = "https://accounts.google.com"`, matching Google's discovery issuer. Google's ID token validation guidance also permits `"accounts.google.com"`; this alternate issuer is not accepted by this preset.

Value

OAuthProvider object for use with a Google OAuth 2.0 app

Examples

```
oauth_provider_google()
```

oauth_provider_huggingface *Configure sign-in with Hugging Face*

Description

Configure Sign in with Hugging Face. Register an OAuth application in your Hugging Face settings, then pass its credentials to `oauth_client()`.

Usage

```
oauth_provider_huggingface(name = "huggingface")
```

Arguments

name	Optional provider name (default "huggingface").

Details

Login uses openid with optional profile and email scopes. Repository access and inference require their own scopes, such as "read-repos" or "inference-api", and user consent. This preset targets confidential applications with a client secret. See Sign in with Hugging Face.

Value

OAuthProvider object configured for Hugging Face.

Examples

```
## Not run:
oauth_provider_huggingface()

## End(Not run)
```

---

oauth_provider_keycloak

Create a Keycloak OAuthProvider (via OIDC discovery)

Description

Look up login settings for a Keycloak realm. Supply your server URL and realm name, then pass the result to oauth_client(). This helper contacts the Keycloak server during setup.

Usage

```
oauth_provider_keycloak(
  base_url,
  realm,
  name = paste0("keycloak-", realm),
  token_auth_style = "body",
  jarm_tolerate_duplicate_top_level_iss = TRUE
)
```

Arguments

base_url	Base URL of the Keycloak server, e.g., "http://localhost:8080". Local HTTP development also requires options(shinyOAuth.allow_insecure_oidc_loopback = TRUE).
realm	Keycloak realm name, e.g., "myrealm"
name	Optional provider name. Defaults to paste0('keycloak-', realm)

token_auth_style
: Optional override for token endpoint authentication method. One of "header" (client_secret_basic), "body" (client_secret_post), "public" (send client_id only; "none" alias also accepted), "private_key_jwt", or "client_secret_jwt". Defaults to "body" for Keycloak. "public" omits client_secret, including one supplied through the environment. NULL infers the method from the provider's token_endpoint_auth_methods_supported metadata.

jarm_tolerate_duplicate_top_level_iss
: Logical. Defaults to TRUE for Keycloak because current Keycloak JARM responses may repeat an identical top-level iss claim. Set FALSE to reject duplicate top-level iss members, including identical ones.

Value

OAuthProvider object configured for the specified Keycloak realm

Examples

```
## Not run:
oauth_provider_keycloak("https://login.example.com", realm = "myrealm")

## End(Not run)
```

---

oauth_provider_microsoft

Create a Microsoft (Entra ID) OAuthProvider

Description

Create provider settings for Microsoft Entra ID. Choose which accounts may sign in with tenant, then pass the provider and your own registered app's credentials to oauth_client().

Usage

```
oauth_provider_microsoft(
  name = "microsoft",
  tenant = c("common", "organizations", "consumers"),
  id_token_validation = NULL,
  userinfo_required = TRUE
)
```

Arguments

name	Optional friendly name for the provider. Defaults to "microsoft"
tenant	Tenant identifier ("common", "organizations", "consumers", or directory GUID). Defaults to "common"

`id_token_validation`
: Optional override (logical). If `NULL` (default), it's enabled automatically when `tenant` looks like a GUID or one of the Microsoft alias tenants (`common`, `organizations`, `consumers`). `common` and `organizations` use Microsoft's tenant-independent issuer and signing-key validation rules; `consumers` uses the stable consumer tenant issuer

`userinfo_required`
: Retrieve profile information from Microsoft Graph. Set FALSE for an API token with a different audience. Manual `get_userinfo()` calls require the complete validated OAuthToken when OIDC validation is enabled.

Details

Use a directory (tenant) ID to target one organization. `"organizations"` allows work or school accounts, `"consumers"` allows personal Microsoft accounts, and `"common"` allows both. Your app registration and app access rules must also permit the intended accounts.

Profiles are fetched from Microsoft Graph by default. Set `userinfo_required = FALSE` for an API token that cannot call Graph.

Value

OAuthProvider object configured for Microsoft identity platform

Advanced identity settings

ID token validation is enabled for these tenant choices. For a directory ID, the issuer must match that directory. `"common"` and `"organizations"` use Microsoft's tenant-independent issuer template and signing-key issuer rules. `"consumers"` uses the consumer tenant issuer. The helper restricts ID token algorithms to RS256.

Setting `id_token_validation = FALSE` disables ID token and nonce checks and leaves OAuth plus any configured profile retrieval. Tenant domains and other unrecognized tenant identifiers require that setting; directory GUIDs and the aliases above support OIDC validation automatically.

Examples

```
if (
  # Example requires configured Microsoft Entra ID (Azure AD) tenant:
  nzchar(Sys.getenv("MS_TENANT")) &&
    interactive() &&
    requireNamespace("later", quietly = TRUE)
) {
  library(shiny)
  library(shinyOAuth)

  # Configure provider and client (Microsoft Entra ID with your tenant)
  client <- oauth_client(
    provider = oauth_provider_microsoft(
      # Provide your own tenant ID here (set as environment variable MS_TENANT)
      tenant = Sys.getenv("MS_TENANT")
    ),
```

```
  # Azure CLI public-client app ID; the tenant must permit this app.
  # For your deployed app, use your own registration and redirect URI:
  client_id = "04b07795-8ddb-461a-bbee-02f9e1bf7b46",
  client_secret = "",
  redirect_uri = "http://localhost:8100",
  scopes = c("openid", "profile", "email")
)

# UI
ui <- oauth_ui(
  fluidPage(
    h3("OAuth demo (Microsoft Entra ID)"),
    uiOutput("oauth_error"),
    tags[["hr"]](),
    h4("Auth object (summary)"),
    verbatimTextOutput("auth_print"),
    tags[["hr"]](),
    h4("User info"),
    verbatimTextOutput("user_info")
  ),
  id = "auth",
  client = client
)

# Server
server <- function(input, output, session) {
  auth <- oauth_module_server("auth", client)

  output[["auth_print"]] <- renderText({
    authenticated <- auth[["authenticated"]]
    tok <- auth[["token"]]
    err <- auth[["error"]]

    paste0(
      "Authenticated?",
      if (isTRUE(authenticated)) " YES" else " NO",
      "\n",
      "Has token? ",
      if (!is.null(tok)) "YES" else "NO",
      "\n",
      "Has error? ",
      if (!is.null(err)) "YES" else "NO",
      "\n\n",
      "Token present: ",
      !is.null(tok),
      "\n",
      "Has refresh token: ",
      !is.null(tok) && isTRUE(nzchar(tok@refresh_token)),
      "\n",
      "Has ID token: ",
      !is.null(tok) && !is.na(tok@id_token),
      "\n",
      "Expires at: ",
```

```
      if (!is.null(tok)) tok@expires_at else "N/A"
    )
  })

  output[["user_info"]] <- renderPrint({
    req(auth[["authenticated"]])
    auth[["token"]]@userinfo
  })

  observeEvent(
    list(auth[["error"]], auth[["error_description"]]),
    {
      if (interactive() && !is.null(auth[["error_description"]])) {
        rlang::inform(c(
          "OAuth error details",
          "i" = paste0("error: ", auth[["error"]]),
          "i" = paste0("error_description: ", auth[["error_description"]])
        ))
      }
    },
    ignoreInit = TRUE
  )

  output[["oauth_error"]] <- renderUI({
    if (is.null(auth[["error"]])) {
      return(NULL)
    }

    msg <- if (identical(auth[["error"]], "access_denied")) {
      "Sign-in was canceled or denied. Please try again."
    } else {
      "Authentication failed. Please try again."
    }

    div(class = "alert alert-danger", role = "alert", msg)
  })
}

# Open localhost:8100 to match the Azure CLI app's registered redirect URI
# and keep browser cookies on the same host. Schedule the browser launch
# before runApp() starts blocking R.
later::later(
  function() {
    utils::browseURL("http://localhost:8100")
  },
  delay = 0.25
)

# Run app
runApp(shinyApp(ui, server), port = 8100, launch.browser = FALSE)
}
```

`oauth_provider_oidc` *Create a generic OpenID Connect (OIDC) OAuthProvider*

Description

Build OIDC provider URLs from a base address and known endpoint paths. Use this when configuring an OIDC service without discovery, with its endpoint paths available from the service configuration or documentation. Use `oauth_provider_oidc_discover()` if your provider offers discovery, which looks up its actual URLs. This helper is for manual configuration; its default paths must match the service you are using.

Usage

```
oauth_provider_oidc(
  name,
  base_url,
  auth_path = "/authorize",
  token_path = "/token",
  userinfo_path = "/userinfo",
  introspection_path = "/introspect",
  use_nonce = TRUE,
  id_token_validation = TRUE,
  jwks_host_issuer_match = TRUE,
  allowed_token_types = c("Bearer"),
  ...,
  token_auth_style = "header",
  issuer = base_url
)
```

Arguments

`name`	Friendly name for the provider
`base_url`	Base URL for OIDC endpoints
`auth_path`	Authorization endpoint path (default: "/authorize")
`token_path`	Token endpoint path (default: "/token")
`userinfo_path`	User info endpoint path (default: "/userinfo")
`introspection_path`	Token introspection endpoint path (default: "/introspect")
`use_nonce`	Logical, whether to use OIDC nonce. Defaults to TRUE
`id_token_validation`	Logical, whether to validate ID tokens automatically for this provider. Defaults to TRUE
`jwks_host_issuer_match`	When TRUE (default), require the discovered JWKS host to match the issuer host exactly. `jwks_host_allow_only` takes precedence and can specify a different exact hostname. `FALSE` disables issuer-host matching.

allowed_token_types
: Character vector of allowed token types for access tokens issued by this provider. Defaults to 'Bearer'

...
: Additional arguments passed to oauth_provider()

token_auth_style
: Token endpoint client authentication style passed to oauth_provider(). Defaults to "header".

issuer
: Exact OIDC issuer identifier. Defaults to base_url, retaining any trailing slash. Supply this explicitly when your provider identifies itself with a different URL from the one used to build endpoint addresses.

Value

OAuthProvider object

Examples

```
# Configure generic OAuth 2.0 provider (no OIDC)
generic_provider <- oauth_provider(
  name = "example",
  auth_url = "https://example.com/oauth/authorize",
  token_url = "https://example.com/oauth/token",
  # Optional URL for fetching user info:
  userinfo_url = "https://example.com/oauth/userinfo"
)

# Configure generic OIDC provider manually
# (This defaults to using nonce & ID token validation)
generic_oidc_provider <- oauth_provider_oidc(
  name = "My OIDC",
  base_url = "https://my-issuer.example.com"
)

# Look up an OIDC provider's settings with discovery
# (requires network access)
if (interactive()) {
  # Using Auth0 sample issuer as an example
  oidc_discovery_provider <- oauth_provider_oidc_discover(
    issuer = "https://samples.auth0.com/"
  )
}

# GitHub preconfigured provider
github_provider <- oauth_provider_github()

# Google preconfigured provider
google_provider <- oauth_provider_google()

# Microsoft preconfigured provider
# For a complete app using a custom tenant ID, see:
# https://lukakoning.github.io/shinyOAuth/reference/oauth_provider_microsoft.html
```

```
# Spotify preconfigured provider
spotify_provider <- oauth_provider_spotify()

# Slack via OIDC discovery
# (requires network access)
if (interactive()) {
  slack_provider <- oauth_provider_slack()
}

# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
if (interactive()) {
  options(shinyOAuth.allow_insecure_oidc_loopback = TRUE)
  oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")
}

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_auth0(domain = "your-tenant.auth0.com")
}

# Okta
# (requires configured Okta domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_okta(domain = "dev-123456.okta.com")
}
```

`oauth_provider_oidc_discover` *Discover and create an OpenID Connect (OIDC) OAuthProvider*

Description

Supply your provider's issuer URL to create an OAuthProvider without entering each service URL yourself. The helper downloads the provider's discovery document (OpenID Connect Discovery) and enables OIDC login checks. Pass the result to `oauth_client()`.

Usage

```
oauth_provider_oidc_discover(
  issuer,
  name = NULL,
  use_pkce = TRUE,
  use_nonce = TRUE,
  id_token_validation = TRUE,
  token_auth_style = NULL,
  id_token_allowed_algs = c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512",
```

```
    "Ed25519", "EdDSA"),
  allowed_token_types = c("Bearer"),
  jwks_host_issuer_match = TRUE,
  issuer_match = c("url", "host", "none"),
  ...,
  allowed_algs = NULL
)
```

Arguments

issuer	The OIDC issuer base URL (including scheme), e.g., "https://login.example.com". The standard discovery-document URL ending in `/.well-known/openid-configuration` is also accepted. Its discovered issuer must map back to that metadata location; the exact returned issuer, including any trailing slash, is retained for subsequent validation.
name	Optional friendly provider name. Defaults to the issuer hostname
use_pkce	Logical, whether to use PKCE for this provider. Defaults to TRUE. Public clients require PKCE. Setting FALSE also prevents automatic selection of public-client authentication; a confidential-client method must be available or explicitly configured instead.
use_nonce	Logical, whether to use OIDC nonce. Defaults to TRUE
id_token_validation	Logical, whether to validate ID tokens automatically for this provider. Defaults to TRUE
token_auth_style	Authentication style for token requests: "header" (client_secret_basic), "body" (client_secret_post), or "public" (public client; send `client_id` only). The alias `"none"` is also accepted for `"public"`. If NULL (default), it is inferred conservatively from discovery: `"header"` (client_secret_basic) is preferred, followed by `"body"` (client_secret_post), then `"public"` if none is advertised and PKCE is enabled. Set `token_auth_style = "public"` explicitly for a public client registration. JWT methods (`"client_secret_jwt"`, `"private_key_jwt"`) and mTLS methods (`"tls_client_auth"`, `"self_signed_tls_client_auth"`) must be selected explicitly. See `oauth_provider()` for the supported methods and their credentials.
id_token_allowed_algs	Character vector of allowed ID token signing algorithms. Defaults to a broad set of common algorithms, including RSA (RS*), ECDSA (ES*), Ed25519, and legacy EdDSA. If the discovery document advertises supported algorithms, the intersection of advertised and caller-provided algorithms is used to avoid runtime mismatches. If there's no overlap, discovery fails with a configuration error (no fallback).
allowed_token_types	Character vector of allowed token types for access tokens issued by this provider. Defaults to 'Bearer'
jwks_host_issuer_match	When TRUE (default), require the discovered JWKS host to match the issuer

host exactly. jwks_host_allow_only takes precedence and can specify a different exact hostname. FALSE disables issuer-host matching.

issuer_match — Character scalar controlling how strictly to validate the discovery document's issuer against the input issuer.

- "url" (default): require the issuer used for discovery to match exactly, including any trailing slash. For a full discovery URL input, require the discovered issuer's standard metadata location to match that URL.
- "host": compare only scheme + host.
- "none": skip issuer-consistency validation.

... — Additional fields passed to oauth_provider(), including pkce_method = "plain" for compatibility with plain-only PKCE metadata.

allowed_algs — Compatibility alias for id_token_allowed_algs. Supply only one spelling.

Details

This function makes a network request. Call it once during app setup, outside server(). Copy the issuer URL exactly from your provider's configuration, including any trailing slash. A full discovery-document URL is also accepted.

Discovered token, UserInfo, introspection, and revocation endpoints are copied into the provider when present. Discovering an introspection endpoint does not itself require token introspection; set introspect = TRUE on oauth_client() when login and refresh must perform that check.

Discovery describes the service, not your app registration. Use the client ID and secret or key issued for your app. If the registration is public (no secret), explicitly select token_auth_style = "public". Other authentication choices and metadata restrictions are described below.

Value

OAuthProvider object configured from discovery

Advanced authentication selection

Automatic selection prefers "header", then "body", then "public" when none is advertised and PKCE is enabled. JWT and mTLS methods require an explicit choice. Selected methods must match the advertised metadata. If token_endpoint_auth_methods_supported is absent, only the OIDC default client_secret_basic ("header") is assumed.

Discovery validation

The discovered issuer must match the requested identifier by default. Endpoints must use HTTPS. Host allowlisting does not permit HTTP: local OIDC development requires options(shinyOAuth.allow_insecure_oidc_l = TRUE) and a loopback host. options(shinyOAuth.allowed_hosts) can further restrict endpoint hosts. Signing-key hosts have their own policy: by default they must match the issuer host; use jwks_host_allow_only for a known different host.

The document must advertise the code flow (response_types_supported includes "code"), non-empty subject_types_supported, RS256 in id_token_signing_alg_values_supported, and a jwks_uri even when automatic ID token validation is disabled. The permitted ID token algorithms are the intersection of id_token_allowed_algs and the advertised algorithms; an empty

intersection is an error. Discovery keeps PKCE `S256` and errors when the provider explicitly excludes it, unless you select `pkce_method = "plain"`.

Advanced metadata

Discovery also records capabilities for PAR, signed/encrypted requests, JARM, DPoP, mTLS, and callback issuer identification. Client construction checks the selected features against this metadata. See `oauth_provider()` for individual fields and the advanced security vignette for setup.

When omitted by the provider, OIDC defaults apply: the JAR `request` parameter is unsupported, request URIs are supported, request URI registration is not required, and response modes are `c("query", "fragment")`. A mode still has to be implemented by shinyOAuth to be usable. Caller-published request URI registration must be arranged with the provider; discovery cannot check your registration. PAR-issued handles do not need that client-hosted URI registration.

Examples

```
# Configure generic OAuth 2.0 provider (no OIDC)
generic_provider <- oauth_provider(
  name = "example",
  auth_url = "https://example.com/oauth/authorize",
  token_url = "https://example.com/oauth/token",
  # Optional URL for fetching user info:
  userinfo_url = "https://example.com/oauth/userinfo"
)

# Configure generic OIDC provider manually
# (This defaults to using nonce & ID token validation)
generic_oidc_provider <- oauth_provider_oidc(
  name = "My OIDC",
  base_url = "https://my-issuer.example.com"
)

# Look up an OIDC provider's settings with discovery
# (requires network access)
if (interactive()) {
  # Using Auth0 sample issuer as an example
  oidc_discovery_provider <- oauth_provider_oidc_discover(
    issuer = "https://samples.auth0.com/"
  )
}

# GitHub preconfigured provider
github_provider <- oauth_provider_github()

# Google preconfigured provider
google_provider <- oauth_provider_google()

# Microsoft preconfigured provider
# For a complete app using a custom tenant ID, see:
# https://lukakoning.github.io/shinyOAuth/reference/oauth_provider_microsoft.html

# Spotify preconfigured provider
```

```
spotify_provider <- oauth_provider_spotify()

# Slack via OIDC discovery
# (requires network access)
if (interactive()) {
  slack_provider <- oauth_provider_slack()
}

# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
if (interactive()) {
  options(shinyOAuth.allow_insecure_oidc_loopback = TRUE)
  oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")
}

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_auth0(domain = "your-tenant.auth0.com")
}

# Okta
# (requires configured Okta domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_okta(domain = "dev-123456.okta.com")
}
```

`oauth_provider_okta` *Create an Okta OAuthProvider (via OIDC discovery)*

Description

Look up login settings for your Okta domain and authorization server. Pass the result to `oauth_client()` with your registered app credentials. This helper makes a discovery request during setup.

Usage

```
oauth_provider_okta(domain, auth_server = "default", name = "okta")
```

Arguments

`domain`	Your Okta domain, e.g., "dev-123456.okta.com"
`auth_server`	Authorization server ID for a custom authorization server (default "default"). Use `NULL` to target the org authorization server at `https://{yourOktaDomain}`.
`name`	Optional provider name (default "okta")

Value

OAuthProvider object configured for the specified Okta domain

Examples

```
## Not run:
oauth_provider_okta("dev-123456.okta.com")

## End(Not run)
```

`oauth_provider_orcid` *Configure researcher sign-in with ORCID*

Description

Let users sign in with their ORCID account. Register your app with ORCID, then pass this provider, the client ID and client secret to `oauth_client()` with `scopes = "openid"`.

Usage

```
oauth_provider_orcid(sandbox = FALSE, name = "orcid")
```

Arguments

`sandbox`	Logical; use ORCID's sandbox instead of production.
`name`	Optional provider name (default `"orcid"`).

Details

ORCID's OIDC login uses the `openid` scope; `/authenticate` is its separate OAuth login scope. Its OIDC metadata does not advertise `profile` or `email` scopes. Additional record permissions depend on whether you registered a Public or Member API client.

Sandbox and production require separate credentials and user accounts. Production redirect URIs must use HTTPS.

Value

OAuthProvider object configured for ORCID.

Advanced registration settings

This preset uses `client_secret_post` authentication. ORCID does not advertise PKCE, so the helper uses client authentication and OIDC nonce validation without PKCE. See the ORCID authentication guide.

Examples

```
## Not run:
oauth_provider_orcid()
oauth_provider_orcid(sandbox = TRUE)

## End(Not run)
```

oauth_provider_slack *Create a Slack OAuthProvider (via OIDC discovery)*

Description

Look up Slack's OpenID Connect settings for Sign in with Slack. This helper contacts the discovery service during setup; pass its result to `oauth_client()` with your Slack app credentials.

Usage

```
oauth_provider_slack(
  name = "slack",
  profile = c("confidential", "public_pkce")
)
```

Arguments

`name`	Optional provider name (default "slack")
`profile`	Slack app registration profile: `"confidential"` (default) uses HTTP Basic and OIDC nonce validation without PKCE; `"public_pkce"` uses S256 PKCE and sends no client secret, matching a Slack registration with PKCE enabled. Enabling PKCE marks the Slack app public; reversing that registration setting requires contacting Slack support. See https://docs.slack.dev/authentication/using-pkce/.

Value

OAuthProvider object configured for Slack

Examples

```
## Not run:
oauth_provider_slack()

## End(Not run)
```

oauth_provider_spotify *Create a Spotify OAuthProvider*

Description

Connect your app to a user's Spotify account. Pass this provider to `oauth_client()` and request the scopes needed by the Spotify API calls you plan to make. The helper configures profile retrieval through Spotify's API and does not expect an ID token.

Usage

```
oauth_provider_spotify(name = "spotify", allow_legacy_id = FALSE)
```

Arguments

name	Optional provider name (default "spotify")
allow_legacy_id	Whether to fall back to Spotify's mutable id when account_id is absent. Default FALSE.

Details

The client's scopes specify permissions in the authorization request. Identity uses Spotify's immutable account_id. Earlier versions used id, so stored account mappings and audit digests based on that field will differ. A custom provider@userinfo_id_selector, such as function(userinfo) userinfo[["id"]], retains the earlier identity mapping; allow_legacy_id only supplies a fallback when account_id is absent.

Value

OAuthProvider object for use with a Spotify OAuth 2.0 app

See Also

For a Shiny app that connects to Spotify to display the user's listening data, see the Spotify example.

Examples

```
oauth_provider_spotify()
```

---

oauth_provider_surfconext

Configure institutional sign-in with SURFconext

Description

Configure institutional OIDC login through SURFconext. Register your service with SURFconext before using its client credentials with oauth_client().

Usage

```
oauth_provider_surfconext(
  environment = c("production", "test"),
  name = "surfconext",
  token_auth_style = NULL
)
```

Arguments

environment	SURFconext environment: "production" (default) or "test".
name	Optional provider name (default "surfconext").
token_auth_style	Authentication style for token requests: "header" (client_secret_basic), "body" (client_secret_post), or "public" (public client; send client_id only). The alias "none" is also accepted for "public". If NULL (default), it is inferred conservatively from discovery: "header" (client_secret_basic) is preferred, followed by "body" (client_secret_post), then "public" if none is advertised and PKCE is enabled. Set token_auth_style = "public" explicitly for a public client registration. JWT methods ("client_secret_jwt", "private_key_jwt") and mTLS methods ("tls_client_auth", "self_signed_tls_client_auth") must be selected explicitly. See oauth_provider() for the supported methods and their credentials.

Details

Login uses the "openid" scope. Additional claims depend on the service's agreed attribute release policy; requesting "email" or "profile" does not guarantee those attributes will be released. The validated issuer and sub establish the login identity. Test and production environments require their own service configuration. See the SURFconext overview.

Value

OAuthProvider object configured for SURFconext.

Examples

```
## Not run:
oauth_provider_surfconext()
oauth_provider_surfconext(environment = "test")

## End(Not run)
```

---

oauth_token_target *Declare an API token target*

Description

Build one entry for oauth_client()'s token_targets. Give the entry a name in that list, such as calendar, then select it with a connection method's target argument. Plain named lists with the same fields are also accepted.

Usage

```
oauth_token_target(
  resource,
  scopes,
  required_scopes = character(),
  resource_ids = character(),
  scope_aliases = character()
)
```

Arguments

resource	Exact API identifier sent to the authorization provider: an absolute URI, such as "urn:example:calendar", or a Microsoft application GUID. This identifies the token's audience; HTTP destinations are configured separately through client resource_bases.
scopes	Non-empty character vector of requested API permissions. A space-separated string is also accepted. Identity and refresh-consent scopes belong in client scopes.
required_scopes	Minimum permissions for this API, default character(). The default permits reduced grants; specify the permissions that every usable token for this target must cover.
resource_ids	Names of approved client resource_bases associated with this target. Optional when only exporting tokens to another R package.
scope_aliases	Optional Microsoft short permission names used in token responses. See oauth_client() for provider-specific restrictions.

Details

This helper checks field syntax and size limits. Client construction checks provider support, scope coverage, API associations and relationships between targets. It preserves the resource identifier exactly and normalizes scopes into distinct tokens.

Value

A plain named list suitable for an entry in token_targets.

Examples

```
oauth_token_target(
  "urn:example:calendar",
  c("calendar.read", "calendar.write"),
  required_scopes = "calendar.read",
  resource_ids = "calendar_api"
)
```

oauth_ui *Set up a Shiny UI for shinyOAuth*

Description

Adds browser setup for `oauth_module_server()` and handles the provider's return before showing the app. The UI wrapper and server module share the same `id` and `client`.

Usage

```
oauth_ui(
  base_ui,
  id = NULL,
  client = NULL,
  request_uri_resolver = NULL,
  clients = NULL
)
```

Arguments

`base_ui`	Your app's complete UI, such as a `fluidPage()` or `tagList()`. Can also be a UI function, optionally accepting the Shiny request.
`id`	Shiny module ID, required with `client` for GET callbacks.
`client`	OAuthClient used by the server module, required with `id`.
`request_uri_resolver`	Optional trusted public request URI resolver; see `oauth_form_post_ui()` for proxy requirements.
`clients`	Optional named list of OAuthClient objects keyed by module ID, mutually exclusive with `id` and `client`.

Details

The wrapper accepts a complete UI object or a UI function, including one accepting the Shiny request. It includes `use_shinyOAuth()` and returns a UI function for `shiny::shinyApp()`. `oauth_form_post_ui()` provides the equivalent handling for POST returns (`response_mode = "form_post"` or `"form_post.jwt"`). Callback paths below the app root require `uiPattern = ".*"` in `shiny::shinyApp()` so Shiny routes those requests to the wrapper.

Value

A UI function to use as the `ui` argument to `shiny::shinyApp()`.

Several login modules

A registry such as `clients = list(auth_a = client_a, auth_b = client_b)` replaces the single `id` and `client` pair. Names match the server module IDs; one wrapper handles all configured query and POST routes.

Advanced routing and storage

Each client in a registry must select a multi-server defense. Shared routes require `authorization_server_mode = "multi_issuer"` and distinct trusted issuers. An RFC 9207 `iss` or signed JARM issuer selects the configured client; the callback is then fully verified. Encrypted JARM on a shared route needs an outer `iss` identifying one configured issuer, which the inner signed response must match.

Callbacks use short-lived, single-use state-store records and a clean continuation URL before application HTML or scripts load. Shared workers need a shared state store with atomic consumption; see `oauth_form_post_ui()` and `custom_cache()`. Continuation URLs retain only application query parameters registered in `redirect_uri`. Client-hosted Request Objects also use this store and are served at the app root.

Without `id` and `client`, ordinary pages render but raw login callbacks are rejected. Callback handling in a custom `use_shinyOAuth()` integration belongs to its HTTP layer; the dependency alone does not process callbacks.

The wrapper sets no-store cache headers and `Referrer-Policy: no-referrer` before page resources load. A meta tag alone may be read too late to protect early requests. A web server may set the same headers.

See Also

`oauth_module_server()`, `oauth_form_post_ui()`, `use_shinyOAuth()`

Examples

```
base_ui <- shiny::fluidPage(
  shiny::h2("My app"),
  shiny::uiOutput("login")
)

# After creating your OAuth client, wrap the UI once with callback handling:
# ui <- oauth_ui(base_ui, id = "auth", client = client)
# Use this UI with your app's server function:
# shiny::shinyApp(ui = ui, server = server)
```

`OAuthClient` *OAuthClient S7 class*

Description

An `OAuthClient` holds your app's registration with a provider: its client ID, credentials, return address, and requested permissions. It also holds the pending login state and client-specific token validation settings used by the Shiny module and token helpers. Create it with `oauth_client()`, which resolves defaults from the provider and the supplied client settings.

Usage

```
OAuthClient(
  provider = NULL,
  client_id = character(0),
  client_secret = character(0),
  client_assertion_private_key = NULL,
  client_assertion_private_key_kid = NA_character_,
  client_assertion_alg = NA_character_,
  client_assertion_audience = NA_character_,
  mtls_client_cert_file = NA_character_,
  mtls_client_key_file = NA_character_,
  mtls_client_key_password = NA_character_,
  mtls_client_ca_file = NA_character_,
  mtls_certificate_bound_access_tokens = FALSE,
  request_object_mode = "parameters",
  response_mode = NA_character_,
  request_object_signing_alg = NA_character_,
  request_object_audience = NA_character_,
  request_object_encryption_alg = NA_character_,
  request_object_encryption_enc = NA_character_,
  request_object_encryption_kid = NA_character_,
  request_object_ttl = 45,
  request_object_nbf_skew = NA_real_,
  dpop_private_key = NULL,
  dpop_private_key_kid = NA_character_,
  dpop_signing_alg = NA_character_,
  dpop_require_access_token = !is.null(dpop_private_key),
  redirect_uri = character(0),
  enforce_callback_issuer = FALSE,
  scopes = token_target_default_scopes(token_targets, provider),
  resource = character(0),
  claims = NULL,
  state_store = cachem::cache_mem(max_age = 300),
  state_payload_max_age = 300,
  state_entropy = 64,
  state_key = random_urlsafe(n = 128),
  scope_validation = "warn",
  claims_validation = "none",
  userinfo_jwt_required_time_claims = character(0),
  required_acr_values = character(0),
  introspect = FALSE,
  introspection_checks = character(0),
  endpoint_auth = list(),
  authorization_server_mode = "single",
  authorization_server_redirect_uris = character(0),
  dpop_require_observed_cnf = FALSE,
  jarm_signed_response_alg = NA_character_,
  jarm_encrypted_response_alg = NA_character_,
```

```
  jarm_encrypted_response_enc = NA_character_,
  jarm_decryption_private_key = NULL,
  jarm_decryption_private_key_kid = NA_character_,
  jarm_max_lifetime = 600,
  mtls_require_observed_cnf = TRUE,
  trusted_id_token_audiences = character(0),
  compare_callback_issuer = is_valid_string(provider@issuer) &&
    (missing(enforce_callback_issuer) || isTRUE(enforce_callback_issuer)),
  client_assertion_typ = "JWT",
  resource_bases = character(0),
  required_scopes = character(0),
  label = default_client_label(provider),
  authorization_method = "GET",
  scope_policy = list(),
  smart = list(),
  introspect_elements = NULL,
  client_private_key = NULL,
  client_private_key_kid = NULL,
  userinfo_jwt_required_temporal_claims = NULL,
  mtls_request_certificate_bound_access_tokens = NULL,
  tls_client_cert_file = NULL,
  tls_client_key_file = NULL,
  tls_client_key_password = NULL,
  tls_client_ca_file = NULL,
  authorization_request_mode = NULL,
  authorization_request_signing_alg = NULL,
  authorization_request_audience = NULL,
  authorization_request_encryption_alg = NULL,
  authorization_request_encryption_enc = NULL,
  authorization_request_encryption_kid = NULL,
  authorization_request_ttl = NULL,
  authorization_request_nbf_skew = NULL,
  token_targets = list(),
 token_target_default = if (length(token_targets) == 1L) names(token_targets) else
    character()
)
```

Arguments

provider	The service configuration, created with a provider helper such as oauth_provider_google() or oauth_provider_oidc_discover().
client_id	The identifier assigned when you register your app with the provider.
client_secret	The secret issued for your app, preferably read with Sys.getenv(). Omit it for registrations that do not use a secret. It is required for token_auth_style = "header". With "body" and PKCE, an empty secret is omitted. With "public" (alias "none"), it is never sent for client authentication. HMAC-signed ID token validation still requires a non-empty secret, regardless of the client authentication method.

client_assertion_private_key
: Optional private key for private_key_jwt client authentication at the token endpoint. Can be an openssl::key or a PEM string containing a private key. Required when the provider's token_auth_style = 'private_key_jwt'. Also used to sign JAR Request Objects, regardless of the token auth style. Current outbound private-key JWT signing supports RSA, EC, and Ed25519 private keys. RSA keys support RS256 and explicitly selected RS384; RS512 and RSA-PSS (PS256, PS384, PS512) are not supported. Ed25519 keys support Ed25519 (RFC 9864) and legacy EdDSA (the default for compatibility); Ed448 is not supported.

client_assertion_private_key_kid
: Optional key identifier (kid) to include in the JWT header for private_key_jwt assertions and JAR Request Objects. Useful when the authorization server uses kid to select the correct verification key.

client_assertion_alg
: Optional JWT signing algorithm to use for client assertions. When omitted, defaults to HS256 for client_secret_jwt. For private_key_jwt, a compatible default is selected based on the private key type/curve (e.g., RS256 for RSA or ES256/ES384/ES512 for EC P-256/384/521, or EdDSA for Ed25519). If an explicit value is provided but incompatible with the key, validation fails early with a configuration error. When the provider advertises token_endpoint_auth_signing_alg_values_supp both explicit values and inferred defaults must be included in that set. Supported values are HS256, HS384, HS512 for client_secret_jwt and asymmetric algorithms supported for outbound signing (RS256, RS384, ES256, ES384, ES512, and Ed25519 or legacy EdDSA with Ed25519 keys) for private keys. RS512, PS256, PS384, and PS512 are not currently supported for outbound client assertions. Reusing an RSA key or HMAC secret requires the same signing algorithm across enabled profiles and endpoint overrides. Use separate keys or secrets when algorithms differ; see the advanced security vignette.

client_assertion_audience
: Optional override for the aud claim used when building JWT client assertions (client_secret_jwt / private_key_jwt). By default, shinyOAuth uses the active token, introspection, or revocation request URL. PAR uses the issuer when configured, otherwise the canonical PAR URL, including when the request uses an mTLS alias. Set an explicit value when required by the provider's registration agreement.

mtls_client_cert_file
: Optional path to the PEM-encoded client certificate (or certificate chain) used for RFC 8705 mutual TLS (mTLS) client authentication and certificate-bound protected-resource requests. Required when provider@token_auth_style is "tls_client_auth" or "self_signed_tls_client_auth". The certificate matching the private key must appear first, followed by its issuers in chain order. CA-first bundles are rejected.

mtls_client_key_file
: Optional path to the PEM-encoded private key used with mtls_client_cert_file. Must be supplied together with mtls_client_cert_file, and is required for RFC 8705 mTLS client authentication.

`mtls_client_key_password`
: Optional password used to decrypt an encrypted PEM private key referenced by `mtls_client_key_file`.

`mtls_client_ca_file`
: Optional path to a PEM CA bundle used to validate the remote HTTPS server certificate when making mTLS requests. This is mainly useful for local or test environments that use self-signed server certificates.

`mtls_certificate_bound_access_tokens`
: Logical. Whether this client intends to request RFC 8705 certificate-bound access tokens when the provider advertises that capability. Default is `FALSE`.

  When `TRUE`, authorization-server requests prefer discovered `mtls_endpoint_aliases`, even when `token_auth_style` is not an mTLS auth style. Token and protected-resource requests present the certificate.

  Requires `mtls_client_cert_file` and `mtls_client_key_file`, and the provider must be configured with `mtls_client_certificate_bound_access_tokens = TRUE`. By default, `mtls_require_observed_cnf = TRUE` also requires locally observable confirmation of the certificate binding. For opaque tokens whose binding is enforced only by the servers, keep `mtls_certificate_bound_access_tokens = TRUE` and set `mtls_require_observed_cnf = FALSE`.

`request_object_mode`
: Controls how the authorization request is transported to the provider.

  - `"parameters"` (default): send OAuth parameters directly on the browser redirect URL.
  - `"request"`: send a signed JWT-secured authorization request (JAR; RFC 9101) via the `request` parameter.
  - `"request_uri"`: publish a signed Request Object by reference and send its URL via the `request_uri` parameter.

  If the provider has a `par_url`, `"parameters"` and `"request"` are sent to that endpoint first using Pushed Authorization Requests (PAR). The browser then receives the provider-issued `request_uri` handle. Caller-published `"request_uri"` mode is separate from PAR and cannot be used when the provider requires PAR.

  Use a signed Request Object when the provider requires JAR or when it must verify the integrity of the authorization parameters. `"request_uri"` lets the provider fetch the object from a published URL instead of carrying the JWT in the browser redirect. Both modes require signing material on the client. shinyOAuth prefers `client_assertion_private_key` when present; otherwise it falls back to HMAC signing with `client_secret`. When Request Object encryption is configured, shinyOAuth signs first and then wraps the signed Request Object in a JWE. Caller-managed `request_uri` publication requires HTTPS; HTTP URLs are rejected even when another configured host policy would otherwise allow them, as required by RFC 9101 Section 5.2. If the provider advertises `request_uri_registration_required = TRUE`, caller-managed `request_uri` publication still depends on the provider having that URI or a matching wildcard prefix registered for the client; shinyOAuth cannot verify that server-side registration automatically.

`response_mode`
: How the provider returns the login result. `NULL` (default) omits the parameter, using the provider's normal query callback. `"query"` explicitly requests

URL parameters; "form_post" requests an HTTP POST. POST callbacks require oauth_form_post_ui().
Signed responses (JWT Secured Authorization Response Mode, JARM) use "jwt", "query.jwt", or "form_post.jwt" and require oauth_module_server(). "jwt" uses the query transport for this authorization-code flow. "form_post.jwt" also needs oauth_form_post_ui(). handle_callback() does not handle JARM. Requested modes must be in response_modes_supported when advertised; fragment modes are not supported.

request_object_signing_alg
: Optional JWS algorithm override for signed authorization requests when request_object_mode uses a Request Object ("request" or "request_uri"). When omitted, shinyOAuth chooses HS256 for HMAC-based signing or a compatible asymmetric default based on client_assertion_private_key (for example RS256, RS384, ES256, ES384, ES512, or EdDSA for Ed25519). RS512, PS256, PS384, and PS512 are not currently supported for outbound signed authorization requests.

request_object_audience
: Optional override for the aud claim used in signed authorization requests. By default, shinyOAuth uses the provider issuer when available. When request_object_mode = "request" or "request_uri", the provider needs either a configured issuer or this override to identify the authorization server in the signed Request Object.

request_object_encryption_alg
: Optional JWE key-management algorithm override for encrypted Request Objects. Current outbound support is limited to RSA-OAEP. Encryption requires both this value and request_object_encryption_enc.

request_object_encryption_enc
: Optional JWE content-encryption algorithm override for encrypted Request Objects. Current outbound support is limited to the AES-CBC-HMAC family (A128CBC-HS256, A192CBC-HS384, A256CBC-HS512). Encryption also requires request_object_encryption_alg.

request_object_encryption_kid
: Optional key identifier (kid) used to select one provider encryption key and emit the outer JWE kid header. This is mainly useful when the provider publishes more than one Request Object encryption key.

request_object_ttl
: Positive number of seconds to keep signed authorization request objects (request JWTs) valid. When request_object_mode = "request_uri", shinyOAuth also uses this value as the default publication window for the referenced Request Object URI. Default is 45.

request_object_nbf_skew
: Optional non-negative number of seconds. When provided, shinyOAuth adds an nbf claim set to iat - request_object_nbf_skew so deployments can tolerate small clock skew while still emitting bounded request-object validity windows. NULL (the default) omits nbf. Request-object nbf is reserved by shinyOAuth and cannot be supplied through extra authorization parameters.

dpop_private_key
: Private key for tying tokens to this app's requests using Demonstrating Proof of Possession (DPoP). Only needed when your provider/API supports DPoP. Accepts an openssl::key or PEM private-key string, using RSA, EC, or Ed25519.

oauth_client() then defaults dpop_require_access_token to TRUE. Supported signing algorithms are RS256, RS384, ES256, ES384, ES512, and Ed25519 or legacy EdDSA with Ed25519 keys; RSA-PSS and other RSA signing algorithms are not supported for outgoing proofs. See dpop_signing_alg and the advanced security vignette.

dpop_private_key_kid
: Optional key identifier (kid) to include in the JOSE header of DPoP proofs. Useful when the authorization or resource server expects a stable key identifier alongside the embedded public JWK.

dpop_signing_alg
: Optional JWT signing algorithm to use for DPoP proofs. When omitted, a compatible asymmetric default is selected based on the private key type/curve (for example RS256, ES256, ES384, or ES512, or EdDSA for Ed25519). RS512, PS256, PS384, and PS512 are not currently supported for outbound DPoP proofs. If an explicit value is provided but incompatible with the key, validation fails early with a configuration error. When the provider advertises dpop_signing_alg_values_supported, both explicit values and inferred defaults must be included in that set.

dpop_require_access_token
: Logical or NULL. When TRUE and dpop_private_key is configured, shinyOAuth requires the authorization server to return token_type = "DPoP" for access tokens and rejects other token types, independently of the access token's representation. Observed binding data must match the configured key; requiring its presence is a separate policy (dpop_require_observed_cnf). In oauth_client(), the default NULL resolves to TRUE when dpop_private_key is configured and to FALSE otherwise. Explicit FALSE also permits Bearer access tokens, including deployments where DPoP binds only the refresh token.

redirect_uri
: The URL where users return after login. It must match the callback URL registered with your provider, including scheme, host, port, and path. Use HTTPS in production.

enforce_callback_issuer
: Logical or NULL. When TRUE, enforce that authorization responses handled through this client include an RFC 9207 iss parameter and reject callbacks unless it exactly matches provider@issuer. This is recommended when one callback URL can receive responses from more than one authorization server. Requires the provider to have a configured issuer.

  When NULL (the oauth_client() helper default), shinyOAuth auto-enables this check for providers that advertise authorization_response_iss_parameter_supported = TRUE and have a configured issuer, such as OIDC discovery providers that expose RFC 9207 support. Set FALSE to opt out explicitly.

scopes
: Character vector of permissions to request. The provider defines the available names. For OIDC (issuer set and infer_oidc_from_issuer = TRUE), shinyOAuth adds "openid" automatically if absent. The resulting set is used in the request and subsequent scope checks. When omitted with token_targets, API scopes are inferred from those declarations. Explicit scopes must include every target's API scopes; they are never expanded from declarations. With no targets, the default requests no API scopes. OIDC adds "openid"; request further identity and refresh-consent scopes explicitly.

`resource` Optional RFC 8707 resource indicator(s). Supply a character vector of absolute URIs to request audience-restricted tokens for one or more protected resources. Each value is sent as a repeated `resource` parameter on the authorization request, initial token exchange, and token refresh requests. Default is `character(0)`.

`claims` Optional request for specific OIDC user information, beyond scopes. Default `NULL` sends no request. Supply a list with `userinfo` and/or `id_token` members, for example `list(userinfo = list(email = list(essential = TRUE)))`. Use `claims_validation = "strict"` if an unmet request must stop login.

Use `NULL` to request a claim without a constraint, `value` for one acceptable value, or `values` for several. Lists are converted to JSON for the provider; a pre-encoded JSON object is also accepted. The provider must support claims requests. See Claims requests below for validation and numeric limits.

`state_store` Storage for pending logins. The default `cachem::cache_mem(max_age = 300)` is suitable for one R process. For multiple app processes, supply a shared `custom_cache()` with atomic `[["take"]]()` and use the same `state_key` on every process. Plain `cachem::cache_disk()` is unsafe for shared login state because its separate read and delete operations do not prevent simultaneous reuse. See `custom_cache()` for method and stored-value requirements.

`state_payload_max_age`

Maximum age of a pending login's encrypted state, in seconds. Default 300. This is checked separately from the state store's entry lifetime; both must allow the returning login.

`state_entropy` Length in characters of the random state identifier, from 22 to 128. Default 64.

`state_key` Secret used to encrypt and protect pending login details. A random key is generated when omitted. This is separate from `client_secret` and is also used for public clients.

Deployments with multiple R processes use the same key and `state_store` across processes. The key is a character string or raw vector containing at least 32 cryptographically random bytes. State uses AES-GCM authenticated encryption.

`scope_validation`

Controls how scope discrepancies are handled when the authorization server grants fewer scopes than requested. RFC 6749 Section 3.3 permits servers to issue tokens with reduced scope, and Section 5.1 allows token responses to omit `scope` when it is unchanged from the requested scope.

- `"warn"` (default): Emits a warning but continues authentication if scopes are missing.
- `"strict"`: Throws an error if any requested scope is missing from the granted scopes. Omitted `scope` is treated as unchanged, not as an error.
- `"none"`: Skips scope validation entirely.

`claims_validation`

What to do if requested claims are missing or have unexpected values: `"warn"` continues with a warning, `"strict"` stops login, and `"none"` skips the check. When omitted, `oauth_client()` uses `"warn"` if `claims` includes `essential = TRUE`, `value`, or `values` requirements, and `"none"` otherwise. Checks on

claims[["id_token"]] require ID token validation (id_token_validation = TRUE or use_nonce = TRUE). Checks on claims[["userinfo"]] require userinfo_required = TRUE and a configured UserInfo endpoint on the provider.

userinfo_jwt_required_time_claims

Optional character vector of temporal JWT claims that must be present when the UserInfo response is a signed JWT (application/jwt). Allowed values are "exp", "iat", and "nbf".

The default, character(0), checks these claims only when present. Set, for example, userinfo_jwt_required_time_claims = "exp" to require an expiry on signed UserInfo JWTs; multiple values require the corresponding time claims.

required_acr_values

Optional character vector of acceptable login requirements, such as a provider's multi-factor authentication (MFA) policy. Use the provider's Authentication Context Class Reference (ACR) identifiers. The validated ID token must contain a matching acr or login fails. The request also sends acr_values as a hint to the provider. Requires id_token_validation = TRUE and an issuer. Default character(0) imposes no requirement.

introspect

If TRUE, ask the provider to confirm the access token is active before completing login and module refreshes. Requires introspection_url; an unsuccessful check or a response other than active = TRUE stops the operation. Default FALSE.

introspection_checks

Optional character vector of additional requirements to enforce on the introspection response when introspect = TRUE. Supported values:

- "sub": require the introspected sub to match the session subject (from a validated ID token sub when available, else from userinfo sub).
- "client_id": require the introspected client_id to match your OAuth client id.
- "scope": validate introspected scope against requested scopes (respects the client's scope_validation mode).
- "token_type": require introspection to return token_type. This is useful for sender-constrained deployments such as DPoP, where introspection can authoritatively report token_type = "DPoP". Default is character(0). Providers may omit these fields from introspection responses.

endpoint_auth

Named list of authentication overrides for par, introspection, and revocation. Token exchange and refresh use the top-level client/provider authentication settings. Each entry may supply token_auth_style, client_secret, client_assertion_private_key, client_assertion_private_key_kid, client_assertion_alg, client_assertion_audience, client_assertion_typ, extra_headers (named character vector), and the mtls_client_* certificate/key/CA fields. Introspection and revocation may also use a separate client_id. Unspecified credentials inherit the client's settings. Discovered endpoint methods and signing algorithms are checked independently. PAR inherits token authentication. Extra token headers apply only to token exchange and refresh; set extra_headers explicitly for every other endpoint that needs them.

`authorization_server_mode`
: Declares whether this client is part of an application that can interact with more than one authorization server, and which RFC 9700 mix-up defense it uses. One of:
  - `"single"` (default): the application uses only one authorization server, so RFC 9700 does not require a mix-up defense.
  - `"multi_issuer"`: authorization responses identify their issuer. JARM response modes satisfy this requirement through their validated `iss` claim. Direct response modes require the provider to advertise `authorization_response_iss_parameter = TRUE`; shinyOAuth then requires and validates the RFC 9207 `iss` response parameter. Missing support metadata is treated as absence of this defense.
  - `"multi_redirect_uri"`: each authorization server uses a distinct redirect URI. Supply the complete set through `authorization_server_redirect_uris`. This mode is supported by `oauth_module_server()`, which compares the browser-visible canonical scheme, authority, and path before parsing callback values.

`authorization_server_redirect_uris`
: Complete character vector of redirect URIs used by the application for its authorization servers when `authorization_server_mode = "multi_redirect_uri"`. It must contain at least two canonically distinct scheme/authority/path routes and include this client's `redirect_uri`. Query and fragment components do not make routes distinct.

`dpop_require_observed_cnf`
: Logical. When `TRUE`, shinyOAuth rejects `token_type = "DPoP"` access tokens unless it can observe `cnf[["jkt"]]` locally, from the token response, introspection, or optional JWT access-token inspection. Set `options(shinyOAuth.access_token_cnf = "opaque")` to disable access-token decoding for both DPoP and mTLS; the compatibility default `"jwt"` inspects JWT `cnf` without treating it as signature validation. Opaque tokens then need binding metadata in the token response or introspection result. Default is `FALSE`.

`jarm_signed_response_alg`
: Optional expected JWS algorithm for signed JWT Secured Authorization Responses (JARM). When omitted and the effective response mode is JARM, shinyOAuth defaults to `RS256`. This value is not sent dynamically on the authorization request; it must match the client metadata and provider behavior configured out-of-band for that client. Current inbound support accepts `HS256`, `HS384`, `HS512`, `RS256`, `RS384`, `RS512`, `ES256`, `ES384`, `ES512`, `Ed25519`, and `EdDSA`. RSA-PSS (`PS256`, `PS384`, `PS512`) and unsecured `none` are not accepted for inbound JARM.

`jarm_encrypted_response_alg`
: Optional expected JWE key-management algorithm for encrypted JARM responses. Current inbound support is limited to `RSA-OAEP`. Like `jarm_signed_response_alg`, this reflects out-of-band client metadata and expected provider behavior rather than an authorization request parameter emitted by shinyOAuth.

`jarm_encrypted_response_enc`
: Optional expected JWE content-encryption algorithm for encrypted JARM responses. Current inbound support is limited to the AES-CBC-HMAC family (`A128CBC-HS256`, `A192CBC-HS384`, `A256CBC-HS512`). When omitted while

jarm_encrypted_response_alg is set, shinyOAuth defaults to A128CBC-HS256. This must also match the provider-side JARM client metadata when encrypted responses are enabled.

jarm_decryption_private_key
: Optional private key used to decrypt encrypted JARM responses. Can be an openssl::key or a PEM string containing a private key. Required when encrypted JARM is enabled.

jarm_decryption_private_key_kid
: Optional key identifier (kid) associated with jarm_decryption_private_key.

jarm_max_lifetime
: Positive number of seconds. Maximum accepted lifetime for a JARM response JWT. Default is 600 seconds, matching JARM's recommended 10-minute upper bound for authorization response JWTs. When a JARM payload includes iat, shinyOAuth enforces exp - iat <= jarm_max_lifetime; otherwise it falls back to the remaining exp window at validation time. Applies only when response_mode uses JARM.

mtls_require_observed_cnf
: Logical, default TRUE. When mtls_certificate_bound_access_tokens = TRUE, require cnf[["x5t#S256"]] in the token response, JWT access token, or introspection and verify that it matches the configured certificate. FALSE allows missing confirmation for server-enforced opaque bindings. Observed confirmation is validated; mismatches and conflicting claims are rejected.

trusted_id_token_audiences
: Character vector of additional ID-token audiences explicitly trusted by this client. Defaults to character(0), which permits only client_id. The token must always include client_id in aud; when azp is present it must equal client_id. Values are matched exactly and case-sensitively and apply to identity-token recipients, independently of API resource addresses.

compare_callback_issuer
: Logical or NULL. Compare any supplied callback iss exactly with provider@issuer, while allowing absence when enforce_callback_issuer = FALSE. NULL enables comparison when an issuer is configured, except when enforce_callback_issuer = FALSE was explicitly supplied. This preserves the existing complete opt-out. Set compare_callback_issuer = TRUE with enforce_callback_issuer = FALSE to check present values without requiring older providers to send iss. Required issuer presence always enables comparison, even when this separate flag is FALSE. Validated JARM supplies its own issuer protection without requiring a redundant outer iss.

client_assertion_typ
: JWT header typ for client authentication. Defaults to "JWT" for existing providers. Use "client-authentication+jwt" with client_assertion_audience set to the provider's trusted issuer identifier for RFC7523bis-11 / OAuth 2.1 draft 16.

resource_bases
: Optional named character vector of approved API base URLs for oauth_connection() and oauth_connections(), default character(). Each resource ID starts with a letter and contains letters, digits, _ or - (at most 64 bytes). Up to 64 bases are supported. HTTPS is required except for loopback development URLs. Requests through a connection stay within the exact scheme, host, effective port

and base path; redirects are disabled. Bases exclude user information, query strings, fragments, dot segments, repeated slashes, semicolon parameters and ambiguous encoded characters. This is local request policy, not evidence of token audience, and does not add the OAuth `resource` authorization parameter.

`required_scopes`
: Optional requested scopes that every usable connection needs, default `character()`. Other requested scopes may be absent from a limited grant. Ordinary OAuth clients compare literal scopes; `smart_client()` selects SMART semantic comparison and also enforces these permissions when validating token responses. Explicit refresh narrowing retains these scopes.

`label`
: Optional display label used in connection summaries; defaults to the provider name, with control characters replaced by spaces and shortened to 128 UTF-8 bytes if needed. If the provider name is empty, missing or not a single string, the default is `"OAuth provider"`. Explicit labels must be non-empty strings of at most 128 bytes without control characters. Labels contain no credentials or patient context.

`authorization_method`
: Browser method for sending the authorization request: `"GET"` (default) or provider-supported `"POST"`. POST submits form fields instead of a URL query. Use the module's `request_login()` or `prepare_authorization_request()`; URL-only helpers reject POST.

`scope_policy`
: Internal versioned scope policy, installed by SMART adapters. Ordinary OAuth clients use the default policy. SMART permission requirements apply independently of `scope_validation`. This parameter is not an argument to `oauth_client()`.

`smart`
: Internal SMART configuration installed by `smart_client()`, empty for ordinary clients. This is not an argument to `oauth_client()`.

`introspect_elements`
: Compatibility alias for `introspection_checks`.

`client_private_key`
: Compatibility alias for `client_assertion_private_key`.

`client_private_key_kid`
: Compatibility alias for `client_assertion_private_key_kid`.

`userinfo_jwt_required_temporal_claims`
: Compatibility alias for `userinfo_jwt_required_time_claims`.

`mtls_request_certificate_bound_access_tokens`
: Compatibility alias for `mtls_certificate_bound_access_tokens`.

`tls_client_cert_file`
: Compatibility alias for `mtls_client_cert_file`.

`tls_client_key_file`
: Compatibility alias for `mtls_client_key_file`.

`tls_client_key_password`
: Compatibility alias for `mtls_client_key_password`.

`tls_client_ca_file`
: Compatibility alias for `mtls_client_ca_file`.

`authorization_request_mode`
: Compatibility alias for `request_object_mode`.

`authorization_request_signing_alg`
: Compatibility alias for `request_object_signing_alg`.

`authorization_request_audience`
: Compatibility alias for `request_object_audience`.

`authorization_request_encryption_alg`
: Compatibility alias for `request_object_encryption_alg`.

`authorization_request_encryption_enc`
: Compatibility alias for `request_object_encryption_enc`.

`authorization_request_encryption_kid`
: Compatibility alias for `request_object_encryption_kid`.

`authorization_request_ttl`
: Compatibility alias for `request_object_ttl`.

`authorization_request_nbf_skew`
: Compatibility alias for `request_object_nbf_skew`.

`token_targets`
: Optional named list for obtaining tokens for several APIs through one login. Build entries with `oauth_token_target()` or plain named lists. Each entry needs a `resource` identifier and `scopes` (also included in explicit client `scopes`). Optional `required_scopes` set that API's minimum permissions; `resource_ids` associate it with named `resource_bases`. Optional Microsoft `scope_aliases` identify unusual short permission names in responses. Requires provider support and `userinfo_required = FALSE`; cannot be combined with `resource` or SMART configuration. See `vignette("token-targets", package = "shinyOAuth")` (online guide) for examples, provider-specific permission rules and declaration limits.

`token_target_default`
: Name of the API to use when a connection method omits `target`. Required with several targets; the sole target is selected automatically.

Details

Configure the app registration with `provider`, `client_id`, `client_secret` (if issued), `redirect_uri`, and `scopes`. Create the client outside your Shiny `server()` function, then pass it to `oauth_module_server()`.

Use the state-store settings for deployments where callbacks can reach different R processes, and the validation settings to enforce required scopes, claims, or authentication context. Certificate (mTLS), key-binding (DPoP), and signed-request/response (JAR/JARM) settings enable those protocol features when supported by your provider and required by your deployment. See the advanced security vignette for examples. The defaults described below refer to `oauth_client()` unless stated otherwise.

Value

Calling the constructor creates an `OAuthClient` object.

Claims requests

Use an R list for most applications. `values` vectors become JSON arrays, including single-element vectors. Each claim entry is `NULL` or a named list; `essential` must be logical, and `value` and

values cannot be combined. Pre-encoded JSON must use unique member names, object-valued target sections and nonempty arrays for values. Invalid requests are rejected even when claims_validation = "none" disables checking returned values.

Numbers use R's numeric precision and must be finite and within +/- (2^53 - 1). Use strings for larger exact identifiers or decimal values that R cannot represent accurately. UserInfo claim checks require automatic profile retrieval and a configured UserInfo endpoint.

Examples

```
# Register an app with GitHub and store its credentials in your environment.
# This creates the configuration; it does not start login or contact GitHub.
client <- oauth_client(
  provider = oauth_provider_github(),
  client_id = "your-client-id",
  client_secret = "your-client-secret",
  redirect_uri = "http://127.0.0.1:8100",
  scopes = c("read:user", "user:email")
)

# In a real app, read credentials with Sys.getenv() and create client
# outside server(). Inside server(), start login with:
# auth <- oauth_module_server("auth", client)
```

OAuthConnection *OAuthConnection R6 class*

Description

A connection lets your Shiny server use the current login to call an API or supply a token to another R package. It checks that the login and permissions are still available, so your app does not need to keep its own token cache.

Details

Inside reactive server code, obtain a connection with connection <- shiny::req(auth$connection()), where auth comes from oauth_module_server(). With oauth_connections_server(), select a saved login using auth$connection(connection_id).

Use $access_token() for another R package or database driver, $request() for an API in the client's resource_bases, and $has_scopes() to check permissions for optional app features. $request(refresh = TRUE) can refresh before sending. See vignette("external-integrations", package = "shinyOAuth") (online guide) for a complete example.

Connection lifetime

A connection belongs to one login and one Shiny session. Token refresh keeps the same connection and ID. Logout, replacement or session closure invalidates the old connection; obtain a new one after the next login. A connection manager can retain the underlying login across navigation, but a new session gets its own connection reference.

Methods check the owning Shiny session. `$is_usable()` returns `FALSE` when access is unavailable; request and token methods raise errors.

Advanced lifecycle behavior

Reactive expressions that read `connection()`, `$access_token()` or `$has_scopes()` do not rerun on token refresh alone if the login and permissions stay the same. Status reads such as `$summary()` may update more often. Reads do not extend a managed owner's inactivity limit. The manager's `touch()` method records explicit user activity.

Use the module's `connection()` for coordinated token acquisition, refresh and token targets. The `oauth_connection()` wrapper follows a supplied token reactive across logins and does not support those features.

Active bindings

`id` Read-only opaque character string identifying this reference. A single module preserves it across refresh and replaces it on login. A manager uses the stored grant's ID across sessions and refreshes; `oauth_connection()` generates an ID lasting only for that reference.

Methods

Public methods:

- `OAuthConnection$new()`
- `OAuthConnection$access_token()`
- `OAuthConnection$has_scopes()`
- `OAuthConnection$is_usable()`
- `OAuthConnection$refresh()`
- `OAuthConnection$summary()`
- `OAuthConnection$identity()`
- `OAuthConnection$request()`
- `OAuthConnection$smart_context()`
- `OAuthConnection$targets()`
- `OAuthConnection$smart_resource()`
- `OAuthConnection$print()`

`OAuthConnection$new()`: Initialize a reference. This constructor is for internal use; applications should use a module's `connection()` method or the legacy `oauth_connection()` wrapper to establish session ownership.

Usage:

```
OAuthConnection$new(
  id,
  client,
  resolve,
  refresh = NULL,
  acquire = NULL,
  is_current = NULL
)
```

Arguments:

`id` Opaque character string identifying the reference.

`client` The OAuthClient to bind to this reference.

`resolve` Internal function with no arguments that enforces session ownership and returns a list with `client` identical to this reference's client and `token` containing the current OAuthToken or `NULL`. It must raise an error when the owning session is unavailable.

`refresh` Optional internal function implementing the owner's coordinated refresh. Legacy wrappers leave this `NULL`.

`acquire` Internal coordinated credential refresh for external use, accepting `async` and returning TRUE or a promise resolving to TRUE.

`is_current` Optional internal reactive predicate. FALSE permanently ends permission notifications for this authorization reference.

Returns: A new `OAuthConnection` instance.

`OAuthConnection$access_token()`: Get a current bearer token for an SDK or database driver, refreshing when needed. Obtain this connection from a module's `connection()` method. The method refreshes credentials when needed, without starting login, requesting new permissions or retrying the SDK operation.

Usage:

```
OAuthConnection$access_token(
  required_scopes = character(),
  target = NULL,
  min_valid_for_seconds = 60,
  force_refresh = FALSE,
  async = FALSE
)
```

Arguments:

`required_scopes` Permissions needed for this operation, in addition to client requirements.

`target` Optional API name from `token_targets`. `NULL` selects the client's default. Its token is acquired when needed.

`min_valid_for_seconds` Minimum remaining lifetime in seconds (default 60). Allow time for the operation and clock differences. Unknown or insufficient expiry triggers refresh; an unusable replacement raises an error.

`force_refresh` Require a new token even if a cached one is usable.

`async` If `TRUE`, return a promise, including for a cached token. If `FALSE`, return a string or raise an error; a synchronous call cannot wait for an already running async refresh.

Details: Errors can indicate an unavailable login, insufficient permission, pending or failed refresh, or an unsuitable token lifetime. Stable reason names include `authorization_unavailable`, `insufficient_scope`, `refresh_pending`, `refresh_unavailable`, `interaction_required`, `lifetime_unavailable`, `unsupported_token_binding`, `unknown_target` and `unsupported_target`. See `vignette("external-integrations", package = "shinyOAuth")` (online guide) for error recovery and async examples.
DPoP and mTLS-bound tokens require their signing key or certificate; use `$request()` instead of exporting a bearer string. Exported tokens retain all provider-issued permissions, which may exceed the app's local operation limits. Recheck the current login and permissions before using an async token snapshot.

Returns: A bearer-token string, or a promise resolving to one. Token failures inherit `shinyOAuth_access_error` with a reason in `condition$context$reason`. Argument and configuration errors can occur before a promise is returned.

`OAuthConnection$has_scopes()`: Check recorded operation permissions for optional-feature UI, without refreshing or requiring an unexpired access token.

Usage:

```
OAuthConnection$has_scopes(scopes, target = NULL)
```

Arguments:

`scopes` Character vector of operation scopes to check.

`target` Optional declared token-target name; defaults to the client's primary target. An unacquired target returns FALSE, without acquisition.

Returns: TRUE when the current authorization covers the configured scopes, otherwise FALSE. Malformed arguments raise input errors.

`OAuthConnection$is_usable()`: Check whether the current token is locally usable. This checks token presence, known unexpired lifetime and the client's required scopes. Request-specific scopes are checked by `[["request"]]()`.

Usage:

```
OAuthConnection$is_usable(target = NULL)
```

Arguments:

`target` Optional API name from `token_targets`. `NULL` selects the client's default. This check never acquires or refreshes a token.

Returns: A single logical value: `TRUE` for an `active` or `limited` connection, otherwise `FALSE`, including when resolution fails.

`OAuthConnection$refresh()`: Refresh a connection obtained from either module's `connection()` method. Its owner coordinates refresh and verifies ownership before updating it. References created with `oauth_connection()` use their existing module's refresh lifecycle and cannot invoke this method.

Usage:

```
OAuthConnection$refresh(scopes = NULL, target = NULL)
```

Arguments:

`scopes` Optional non-empty character vector requesting fewer permissions for this connection, or `NULL` (default). Scopes must be covered by the current grant and client configuration, and retain the client's required scopes. SMART clients use semantic coverage.

`target` Optional declared token-target name. With targets, narrowing applies only to the selected entry and persists across target switching and reauthorization. Other targets keep their own retained scope limits.

Details: Successful narrowing persists across later automatic refreshes and retained sessions. A new authorization is needed to request broader permissions. It changes the app's accepted permissions, not the provider's underlying consent. Providers may reject narrowing; the request is not retried without the requested scopes.
Microsoft API targets do not support explicit narrowing because their tokens can include earlier consent. The call fails before sending with reason `unsupported_scope_narrowing`, leaving

the current login usable. Use a separate registration or change provider consent for a token with fewer permissions.
OIDC clients that require a profile must retain `openid`. Include any further profile permissions in client `required_scopes`. See `vignette("token-targets", package = "shinyOAuth")` (online guide) for OIDC consent and provider-specific refresh rules.

Returns: `TRUE` after a successful commit, or a promise resolving to `TRUE` when the module uses async transport. Failure raises a redacted error.

`OAuthConnection$summary()`: Resolve the current connection and return status information without credentials, identity claims or token extension fields. Raises an error when called outside the owning session or after that session closes.

Usage:

```
OAuthConnection$summary(target = NULL)
```

Arguments:

`target` Optional API name from `token_targets`. `NULL` selects the client's default. Status and expiry describe that API's current token; this read never acquires or refreshes it.

Details: Managed lifecycle states take precedence: `refreshing` means a refresh claim is in progress, `uncertain` requires a new authorization after an ambiguous refresh outcome, `disconnected` means local access was removed, and `unavailable` means the stored credentials could not be restored. Otherwise token status is evaluated in this order:

- `not_acquired`: the selected target has no token yet.
- `disconnected`: there is no current token.
- `expiry_unknown`: the token's expiry is unknown.
- `expired`: the token has reached its expiry time.
- `insufficient_scope`: the grant lacks a client-required scope.
- `limited`: required scopes are covered, but some other requested scopes are absent from the grant.
- `active`: all requested scopes are covered.

Scope checks enforce retained target limits and use the token's current `granted_scopes`, which may be assumed or carried forward when an ordinary OAuth provider omits scope information. SMART clients require explicit evidence and use semantic coverage for both connection and operation permissions. See OAuthToken for the distinction from verified scope evidence.

Returns: A named list with the following entries:

- `connection_id`: the reference's character ID.
- `client_label`: the client's application-defined character label.
- `status`: one of the character values listed in this method's details.
- `expires_at`: numeric seconds since the Unix epoch, `NA_real_` when there is no token or its expiry is unknown, or `Inf` for a non-expiring token.
- `resource_ids`: character vector of the client's approved resource IDs.
- `replaces_connection_id`: present on managed replacement authorizations; identifies the locally ended connection passed to `reauthorize()`.

`OAuthConnection$identity()`: Read explicitly selected fields from the authorization's cryptographically validated OIDC identity.

Usage:

```
OAuthConnection$identity(claims = c("iss", "sub"), userinfo = character())
```

Arguments:

`claims` Character vector of ID-token claim names, defaulting to `c("iss", "sub")`. Use `character()` to select none.

`userinfo` Character vector of previously fetched UserInfo field names, defaulting to none. UserInfo must have a `sub` exactly matching the validated ID token before any requested profile fields are returned.

Details: Available inside the owning session's reactive context for connections with validated OIDC identity.
Ordinary connections require a usable access token with `openid` currently granted. Target connections retain their validated identity while the authorization remains available in the owning session, even when the primary access token expires or its current scopes no longer include `openid`.
These are the last validated identity/profile snapshots; an OAuth refresh can retain earlier ID-token claims.

Returns: A list with `id_token_claims` and `userinfo`, each containing only selected fields that exist. Missing fields are omitted.

`OAuthConnection$request()`: Resolve the current token and perform an authenticated request within a named resource base. The connection must be usable, and its current grant must cover any scopes required for this operation.

Usage:

```
OAuthConnection$request(
  resource_id,
  path = "",
  query = NULL,
  method = "GET",
  required_scopes = character(),
  configure = NULL,
  target = NULL,
  refresh = FALSE,
  min_valid_for_seconds = 60
)
```

Arguments:

`resource_id` Single character string naming an entry in the client's `resource_bases`.

`path` Single character string resolved relative to the selected base directory; `""` selects the base itself. Absolute and root-relative URLs, including pagination links, must remain within the same approved origin and base path. Dot segments and ambiguous encodings are rejected.

`query` Optional named list of query parameters, or `NULL`.

`method` Single HTTP method string, defaulting to `"GET"`. `TRACE` and `TRACK` are rejected by the resource transport.

`required_scopes` Character vector of scopes required for this operation, in addition to the client's required scopes. They must have been requested by the client and be covered by the current grant. `character()` adds no operation-specific scope check.

`configure` Optional function taking an unauthenticated `httr2::request()` and returning it with only body and application headers changed. Use `httr2::req_body_json()`, `httr2::req_body_form()`, `httr2::req_body_raw()` and `httr2::req_headers()`. Set the HTTP method with `method` above. URL, transport policies, authentication and Host headers cannot be changed. Configuration runs once. Both refresh modes recheck the current authorization afterward, before sending the request.

`target` Optional declared token-target name. Target clients require an explicit association through that declaration's `resource_ids`.

`refresh` If `TRUE`, obtain a usable token before sending the request. Default `FALSE` uses the current token. This is synchronous and cannot wait for an async refresh. Failed API calls are not refreshed or replayed; ordinary HTTP retries are disabled for this call. DPoP nonce handling remains available. The destination and `configure` result are checked before refresh, and permission is checked again before sending.

`min_valid_for_seconds` Minimum remaining lifetime when `refresh = TRUE`, in seconds. Has the same meaning as on `$access_token()`.

Details: Uses `perform_resource_req()` with the configured client for Bearer, DPoP and mTLS authentication. Redirects are never followed. Transport error messages are redacted to exclude resource paths, queries and response bodies. Scope requirements are supplied by the application; they cannot be inferred from an arbitrary API's HTTP method and path.

Returns: An httr2 response object. Invalid resources, unusable connections, insufficient scopes and transport failures raise errors.

`OAuthConnection$smart_context()`: **[Experimental]**

Read interpreted context for a usable SMART connection in this session.

Usage:

```
OAuthConnection$smart_context()
```

Returns: The sensitive context list documented in `smart_context()`.

`OAuthConnection$targets()`: Inspect declared targets without exposing credentials.

Usage:

```
OAuthConnection$targets()
```

Returns: A named list of target status and granted scopes. A target with no response yet has status `not_acquired`; this is not proof of denied consent.

`OAuthConnection$smart_resource()`: **[Experimental]**

Fetch the contextual Patient or validated fhirUser through the approved FHIR base, using current read permissions. `smart_patient()` and `smart_fhir_user()` are the corresponding public helpers.

Usage:

```
OAuthConnection$smart_resource(kind)
```

Arguments:

`kind` Either `"patient"` or `"fhirUser"`.

Returns: An httr2 response. Missing context, scope or resource binding raises an error before an authenticated request is sent.

`OAuthConnection$print()`: Print the class name and session-binding description, with credentials redacted.

Usage:

```
OAuthConnection$print(...)
```

Arguments:

`...` Unused; accepted for compatibility with `base::print()`.

Returns: This reference, invisibly.

See Also

`oauth_connection()`, OAuthClient, `perform_resource_req()`

Examples

```
## Not run:
# Configure outside server(), using an existing provider:
client <- oauth_client(
  provider, client_id = "registered-app",
  redirect_uri = "https://app.example/callback", scopes = c("read", "write"),
  resource_bases = c(api = "https://api.example/v1"),
  required_scopes = "read"
)
server <- function(input, output, session) {
  auth <- oauth_module_server("auth", client, refresh_proactively = TRUE)
  output[["status"]] <- shiny::renderText({
    connection <- shiny::req(auth[["connection"]]())
    connection[["summary"]]()[["status"]]
  })
  records <- shiny::reactive({
    connection <- shiny::req(auth[["connection"]]())
    response <- httr2::request("https://api.example/v1/records") |>
      httr2::req_auth_bearer_token(connection[["access_token"]]("read")) |>
      httr2::req_perform()
    httr2::resp_body_json(response)
  })
}

## End(Not run)
```

`OAuthProvider` *OAuthProvider S7 class*

Description

An `OAuthProvider` describes a service such as Google or GitHub: where users sign in, where your app requests tokens, and which checks to perform. Start with a provider helper such as `oauth_provider_google()` or `oauth_provider_oidc_discover()`. Use `oauth_provider()` for manual setup; these functions return an instance of this class with the corresponding endpoint and validation settings.

Usage

```
OAuthProvider(
  name = character(0),
  auth_url = character(0),
  token_url = character(0),
  userinfo_url = NA_character_,
  introspection_url = NA_character_,
  revocation_url = NA_character_,
  par_url = NA_character_,
  par_required = FALSE,
  authorization_request_front_channel_mode = "compat",
  request_object_signing_alg_values_supported = character(0),
  request_object_encryption_alg_values_supported = character(0),
  request_object_encryption_enc_values_supported = character(0),
  request_object_encryption_jwk = NULL,
  signed_request_object_required = FALSE,
  request_parameter_supported = NA,
  request_uri_parameter_supported = NA,
  request_uri_registration_required = NA,
  token_endpoint_auth_signing_alg_values_supported = character(0),
  dpop_signing_alg_values_supported = character(0),
  authorization_response_iss_parameter_supported = FALSE,
  response_modes_supported = character(0),
  issuer = NA_character_,
  issuer_match = "url",
  use_nonce = is_valid_string(issuer) && isTRUE(infer_oidc_from_issuer),
  use_pkce = TRUE,
  pkce_method = "S256",
  userinfo_required = FALSE,
  userinfo_id_selector = function(userinfo) userinfo[["sub"]],
 userinfo_id_token_match = is_valid_string(userinfo_url) && is_ok_host(userinfo_url) &&
    (isTRUE(id_token_validation) || isTRUE(use_nonce)),
  userinfo_signed_jwt_required = FALSE,
 id_token_required = is_valid_string(issuer) && isTRUE(infer_oidc_from_issuer),
 id_token_validation = is_valid_string(issuer) && isTRUE(infer_oidc_from_issuer),
  id_token_at_hash_required = FALSE,
  extra_auth_params = list(),
  extra_token_params = list(),
  extra_token_headers = character(0),
  mtls_endpoint_aliases = list(),
  mtls_client_certificate_bound_access_tokens = FALSE,
  token_auth_style = "header",
  jwks_cache = cachem::cache_mem(max_age = 3600),
  jwks_pins = character(0),
  jwks_pin_mode = "any",
 jwks_host_issuer_match = is_valid_string(issuer) && (isTRUE(id_token_validation) ||
    isTRUE(id_token_required)),
  jwks_host_allow_only = NA_character_,
```

```
  id_token_allowed_algs = c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512",
    "Ed25519", "EdDSA"),
  allowed_token_types = "Bearer",
  leeway = getOption("shinyOAuth.leeway", 30),
  infer_oidc_from_issuer = TRUE,
  jwks_uri = NA_character_,
  userinfo_allowed_algs = NULL,
  allow_missing_token_type = FALSE,
  jarm_signing_alg_values_supported = character(0),
  jarm_encryption_alg_values_supported = character(0),
  jarm_encryption_enc_values_supported = character(0),
  jarm_tolerate_duplicate_top_level_iss = FALSE,
  endpoint_auth_metadata = list(),
  allowed_algs = NULL,
  require_pushed_authorization_requests = NULL,
  require_signed_request_object = NULL,
  require_request_uri_registration = NULL,
  tls_client_certificate_bound_access_tokens = NULL,
  token_target_mode = "none",
  token_response_scope_format = "space",
  userinfo_id_selector_policy_id = NA_character_
)
```

Arguments

`name`	Provider name (e.g., "github", "google"). Cosmetic only; used in logging and audit events
`auth_url`	URL of the provider's login and permission page.
`token_url`	URL where R exchanges the returned code for tokens.
`userinfo_url`	User info endpoint URL (optional)
`introspection_url`	Optional URL where the provider can confirm whether a token is still active (RFC 7662).
`revocation_url`	Optional URL where the app can ask the provider to invalidate a token, for example during logout (RFC 7009).
`par_url`	Optional Pushed Authorization Request (PAR) URL (RFC 9126). When set, shinyOAuth first sends the authorization request from server to provider and then redirects the browser with the returned `request_uri` handle instead of the full request payload. Use PAR to keep most request details out of the browser URL, submit large requests, or meet a provider's PAR requirement. The provider must support this endpoint.
`par_required`	Logical. Whether the provider requires authorization requests to be sent via PAR. When `TRUE`, `par_url` must also be configured.
`authorization_request_front_channel_mode`	Character scalar controlling which browser-visible outer parameters shinyOAuth keeps when the actual authorization request is carried by JAR or PAR. Use `"compat"` (default) to keep OIDC-compatible parameters with outer `client_id`,

response_type, and scope when an issuer is configured. Use "minimal" for plain OAuth browser redirects and for PAR deployments whose authorization endpoint accepts only client_id plus the provider-issued request_uri handle. OpenID Connect by-value request and caller-managed request_uri transports reject "minimal" because OIDC still requires outer response_type and an outer scope containing openid.

request_object_signing_alg_values_supported
: Optional vector of JWS algorithms that the provider advertises for signed Request Objects (RFC 9101). This is mainly used for early validation when an OAuthClient sends request_object_mode = "request" or request_object_mode = "request_uri".

request_object_encryption_alg_values_supported
: Optional vector of JWE key-management algorithms that the provider advertises for encrypted Request Objects. This metadata is used for early validation when an OAuthClient enables Request Object encryption.

request_object_encryption_enc_values_supported
: Optional vector of JWE content-encryption algorithms that the provider advertises for encrypted Request Objects. This metadata is used for early validation when an OAuthClient enables Request Object encryption.

request_object_encryption_jwk
: Optional explicit recipient public key used to encrypt Request Objects when discovery-backed JWKS selection is not available or when you need to pin one specific encryption key. Accepts an OpenSSL public key, a PEM public-key string, a parsed JWK object, or a JWK JSON string.

signed_request_object_required
: Logical. Whether the provider requires signed Request Objects for authorization requests. When TRUE, client construction requires request_object_mode = "request" or "request_uri". This represents the provider's registration policy and enforces local construction. Provider-side rejection of unsigned requests depends on that registration, not on this local flag.

request_parameter_supported
: Logical or NA. Whether discovery metadata explicitly advertises support for the authorization-request request parameter. NA means the provider did not say. Discovery-derived providers apply the OpenID Connect default (FALSE) when this metadata is omitted.

request_uri_parameter_supported
: Logical or NA. Whether discovery metadata explicitly advertises support for the authorization-request request_uri parameter for caller-managed request URIs. NA means the provider did not say. Discovery-derived providers apply the OpenID Connect default (TRUE) when this metadata is omitted. PAR-issued request_uri handles remain valid even when this metadata is FALSE.

request_uri_registration_required
: Logical or NA. Whether discovery metadata says caller-managed request_uri values must be pre-registered. NA means the provider did not say. Discovery-derived providers apply the OpenID Connect default (FALSE) when this metadata is omitted. shinyOAuth can publish caller-managed request_uri values through oauth_module_server(). TRUE means the public URI or a matching

prefix belongs in the provider's client registration. The package stores this requirement as metadata; provider-side registration is outside local validation.

`token_endpoint_auth_signing_alg_values_supported`
: Optional vector of JWS algorithms that the provider advertises for JWT-based client authentication (`client_secret_jwt` / `private_key_jwt`) at the token endpoint. This metadata is used for early validation of `OAuthClient@client_assertion_alg` and inferred JWT client-assertion defaults.

`dpop_signing_alg_values_supported`
: Optional vector of JWS algorithms that the provider advertises for DPoP proof JWTs (RFC 9449). This metadata is used for early validation of `OAuthClient@dpop_signing_alg` and inferred outbound DPoP signing defaults.

`authorization_response_iss_parameter_supported`
: Logical. Whether the provider advertises RFC 9207 support for returning an `iss` parameter on the authorization response. When `TRUE`, the `oauth_client()` helper can auto-enable callback issuer enforcement when the caller leaves `enforce_callback_issuer` unset and the provider also has a configured `issuer`.

`response_modes_supported`
: Optional character vector of OAuth/OIDC `response_mode` values advertised by the provider. Discovery-backed providers use the discovery metadata value, defaulting to `c("query", "fragment")` when omitted per OIDC Discovery/RFC 8414. Generic providers may leave this empty when capabilities are not known. Provider metadata may include response modes that shinyOAuth does not implement. Requesting an unsupported mode raises an error.

`issuer`
: Optional authorization-server issuer URL. You need this for issuer validation and features such as ID-token validation. shinyOAuth uses it to verify issuer claims and locate signing keys (JWKS), typically through an OIDC discovery document.

`issuer_match`
: Character scalar controlling how strictly the discovery document's `issuer` is validated against `issuer` when it later performs runtime discovery to locate the JWKS URI.

  - `"url"` (default): require the issuer used for discovery to match the discovery metadata exactly, including any trailing slash.
  - `"host"`: compare only scheme + host.
  - `"none"`: do not validate discovery issuer consistency.

  The Microsoft helper configures tenant-independent issuer handling for supported alias tenants.

`use_nonce`
: Whether to tie the ID token to this login using a random nonce. The package sends it in the request and checks it in the returned ID token.

`use_pkce`
: Whether to protect the code exchange using Proof Key for Code Exchange (PKCE), which public clients require. It sends a `code_challenge` with the login request and a matching secret `code_verifier` during token exchange.

`pkce_method`
: PKCE code challenge method: `"S256"` (default), or `"plain"` for compatibility with providers that require it.

`userinfo_required`
: Whether to fetch a user profile after token exchange. The result is stored in `token@userinfo`; a failed required fetch stops login. In `oauth_provider()`, this defaults to `TRUE` when `userinfo_url` is supplied and `FALSE` otherwise.

userinfo_id_selector
: A function that extracts the user ID from the userinfo response. Should take a single argument (the userinfo list) and return the user ID as a string.

  This is used for helpers that need a provider-specific application user identifier, such as audit fields. It does not replace OIDC subject binding: when a validated ID token and UserInfo are both available, their actual sub claims are always compared. Helper constructors like oauth_provider() and oauth_provider_oidc() provide a default selector that extracts sub.

userinfo_id_token_match
: Require the fetched profile to belong to the validated login identity. Defaults to TRUE when a UserInfo URL and ID-token or nonce validation are enabled, including for manual get_userinfo() calls. Pass the complete OAuthToken for this check. An explicit FALSE permits profile retrieval without a validated identity; when both are available, their actual sub values must still match.

userinfo_signed_jwt_required
: Require a profile signed by the provider. Default FALSE accepts ordinary JSON profiles. When TRUE, configure a UserInfo URL and issuer, and arrange signed responses with the provider. Signatures use userinfo_allowed_algs; unsigned, HMAC-signed and encrypted profiles are not accepted by normal configuration. This requirement also applies to manual retrieval when userinfo_required = FALSE.

id_token_required
: Whether to require an ID token to be returned during token exchange. If no ID token is returned, the token exchange will fail. This only makes sense for OpenID Connect providers and may require the client's scope to include openid.

  Both the S7 constructor and oauth_provider() enable this when an issuer is supplied and infer_oidc_from_issuer = TRUE. Pure OAuth 2.0 providers keep this disabled by default.

id_token_validation
: Whether to perform ID token validation after token exchange. This requires the provider to be a valid OpenID Connect provider with a configured issuer and the token response to include an ID token (may require setting the client's scope to include openid).

  Both the S7 constructor and oauth_provider() enable this when an issuer is provided and infer_oidc_from_issuer = TRUE. An explicit FALSE disables automatic ID token validation.

id_token_at_hash_required
: Whether to require the at_hash (Access Token hash) claim in the ID token. When TRUE, login fails if the ID token does not contain an at_hash claim or if the claim does not match the access token. When FALSE (default), at_hash is validated only when present. Requires id_token_validation = TRUE.

extra_auth_params
: Extra parameters for authorization URL

extra_token_params
: Extra parameters for token exchange. scope is reserved and cannot be unblocked. For explicit refresh scope narrowing use a managed connection's [["refresh"]](scopes = ... Configure login scopes on oauth_client() instead.

`extra_token_headers`
: Extra headers for back-channel token-style requests (named character vector), applied only to token exchange and refresh. Configure `oauth_client(endpoint_auth = ...)` for headers needed by PAR, introspection, or revocation.

`mtls_endpoint_aliases`
: Optional named list of RFC 8705 mTLS endpoint aliases. Names should follow the metadata keys such as `token_endpoint`, `userinfo_endpoint`, `introspection_endpoint`, `revocation_endpoint`, `par_endpoint`, or `pushed_authorization_request_endpoint`, and values must be absolute URLs. This is an advanced setting used when a provider publishes separate mTLS-specific endpoints.

`mtls_client_certificate_bound_access_tokens`
: Logical. Whether the authorization server advertises RFC 8705 capability to issue certificate-bound access tokens. This describes server capability; the client still has to opt into mTLS separately. When `TRUE`, token responses may include a `cnf` claim with an `x5t#S256` thumbprint that downstream requests must match with the same certificate.

`token_auth_style`
: How the client authenticates at the token endpoint. One of:

  - "header": HTTP Basic (client_secret_basic)
  - "body": Form body (client_secret_post)
  - "public": Public-client form body (`none` in discovery metadata); sends `client_id` but never `client_secret`, even if one is configured. The alias `"none"` is also accepted.
  - "tls_client_auth": RFC 8705 mutual TLS client authentication using a client certificate chained to a trusted CA
  - "self_signed_tls_client_auth": RFC 8705 mutual TLS client authentication using a self-signed client certificate registered out of band with the provider
  - "client_secret_jwt": JWT client assertion signed with HMAC using client_secret (RFC 7523)
  - "private_key_jwt": JWT client assertion signed with an asymmetric key (RFC 7523)

`jwks_cache`
: Storage for the provider's public signing keys. Defaults to `cachem::cache_mem(max_age = 3600)`, an in-memory cache lasting one hour. A `custom_cache()` can share keys across processes. Shorter lifetimes pick up changed keys sooner; longer lifetimes reduce network requests. HTTP cache directives can shorten this lifetime. Responses marked `no-store` are not retained, and `no-cache` responses are fetched again before reuse. Advertised freshness also accounts for `Age` and `Expires`. The package also attempts a rate-limited refresh when a key is missing or no longer verifies a signature. For async login and token operations, custom caches use synchronous fallback unless they explicitly declare worker safety; see `custom_cache()` and its `worker_safe` argument.

`jwks_pins`
: Optional character vector of RFC 7638 JWK thumbprints (base64url) to pin against. If non-empty, fetched JWKS must contain keys whose thumbprints match these values depending on `jwks_pin_mode`. This is an advanced hardening option that lets you pre-authorize expected keys. Only keys matching a configured pin are eligible for signature verification or Request Object encryption;

jwks_pin_mode controls whether the surrounding JWK Set may also contain unpinned keys.

jwks_pin_mode
: Pinning policy when jwks_pins is provided. Either "any" (default; at least one key in JWKS must match) or "all" (every RSA/EC/OKP public key in JWKS must match one of the configured pins)

jwks_host_issuer_match
: When TRUE, enforce that the discovery jwks_uri host matches the issuer host exactly. Defaults to FALSE at the class level, but helper constructors for OIDC (e.g., oauth_provider_oidc() and oauth_provider_oidc_discover()) enable this by default. The generic helper oauth_provider() will also automatically set this to TRUE when an issuer is provided and either id_token_validation or id_token_required is TRUE (OIDC-like configuration). An explicit FALSE disables issuer-host matching. jwks_host_allow_only takes precedence and can specify a different exact hostname, as used by the Google helper.

jwks_host_allow_only
: Optional explicit hostname that the jwks_uri must match. When provided, jwks_uri host must equal this value (exact match). You can pass either just the host (e.g., "www.googleapis.com") or a full URL; only the host component will be used. If you need to include a port or an IPv6 literal, pass a full URL (e.g., https://[::1]:8443) - the port is ignored and only the hostname part is used for matching. Takes precedence over jwks_host_issuer_match.

id_token_allowed_algs
: Signing algorithms accepted for ID tokens. Defaults to c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512", "Ed25519", "EdDSA"); Ed448 and unsupported algorithms are rejected. Use a narrower set to match your provider registration. HMAC (HS256, HS384, HS512) requires a server-only client secret, options(shinyOAuth.allow_hs = TRUE), and exactly one HS algorithm. Reused keys or secrets must not have conflicting algorithm settings. For RSA key metadata, defaults and rotation, see signing-key validation.

allowed_token_types
: Character vector of acceptable OAuth token types returned by the token endpoint (case-insensitive). Successful token responses must include token_type by default; when allowed_token_types is non-empty, its value must also be one of the allowed values or the flow fails fast with a shinyOAuth_token_error. The oauth_provider() helper defaults to c("Bearer"). When the OAuthClient is configured with dpop_private_key, shinyOAuth also accepts token_type = "DPoP" and uses DPoP proofs on supported token and downstream requests. Other non-Bearer token types (for example MAC) are rejected. Set allowed_token_types = character() explicitly only to disable the value allowlist while still requiring token_type itself.

leeway
: Clock skew leeway (seconds) applied to ID token exp/iat/nbf checks and state payload issued_at future check. Default 30. Can be globally overridden via option shinyOAuth.leeway.

infer_oidc_from_issuer
: Whether setting issuer enables OpenID Connect behavior. Default TRUE: helpers enable OIDC nonce/ID token defaults and the client adds the openid scope. Set FALSE for an OAuth-only server that has an issuer identifier but does not implement OIDC.

`jwks_uri` Optional URL of the provider's public signing keys (JWKS). Normally these are located through OIDC discovery. Set this for manual key configuration, including OAuth-only JARM providers.

`userinfo_allowed_algs`
Optional signing algorithm allowlist for UserInfo JWTs. `NULL` inherits `id_token_allowed_algs` for manually configured providers. Discovery negotiates this independently against UserInfo metadata. Use a single algorithm to enforce the client's registered UserInfo signing choice. An empty vector rejects all signed UserInfo algorithms. Unlabelled RSA keys follow the same binding policy as `id_token_allowed_algs`.

`allow_missing_token_type`
Logical, default `FALSE`. Compatibility for providers that issue Bearer tokens without the OAuth-required `token_type`. When `TRUE`, login and refresh assume `"Bearer"` only when the field is absent. Explicit null, empty, invalid, or unsupported values still fail validation. The fallback never applies to clients configured with DPoP.

`jarm_signing_alg_values_supported`
Optional vector of JWS algorithms that the provider advertises for signed JWT Secured Authorization Responses (JARM).

`jarm_encryption_alg_values_supported`
Optional vector of JWE key-management algorithms that the provider advertises for encrypted JARM responses.

`jarm_encryption_enc_values_supported`
Optional vector of JWE content-encryption algorithms that the provider advertises for encrypted JARM responses.

`jarm_tolerate_duplicate_top_level_iss`
Logical. Whether shinyOAuth should tolerate repeated identical top-level `iss` members in signed JARM payloads for this provider. This is a compatibility setting for providers that emit duplicate identical top-level `iss` claims. When `TRUE`, shinyOAuth collapses repeated identical top-level `iss` members before duplicate-member rejection. Conflicting duplicates and nested duplicate `iss` members are rejected. Defaults to `FALSE`.

`endpoint_auth_metadata`
Named list of independent `introspection` and `revocation` authentication metadata. Each entry has `methods` and `signing_algs` character vectors (or `NULL` for omitted metadata). Discovery retains these fields and applies the RFC 8414 Basic-auth default for omitted revocation methods. Omitted introspection methods have no default.

`allowed_algs` Compatibility alias for `id_token_allowed_algs`.

`require_pushed_authorization_requests`
Compatibility alias for `par_required`.

`require_signed_request_object`
Compatibility alias for `signed_request_object_required`.

`require_request_uri_registration`
Compatibility alias for `request_uri_registration_required`.

`tls_client_certificate_bound_access_tokens`
Compatibility alias for `mtls_client_certificate_bound_access_tokens`.

`token_target_mode`
: How this provider obtains API-specific tokens: `"none"` (default), `"rfc8707"`, or `"microsoft"`. This applies to client `token_targets` and depends on provider support. The Microsoft helper selects `"microsoft"` automatically; discovery does not enable targets. See `vignette("token-targets", package = "shinyOAuth")` (online guide) for setup and permission rules.

`token_response_scope_format`
: Separator used for permissions in token responses: `"space"` (default) or `"comma"`. The GitHub helper sets `"comma"` automatically. Set it explicitly for a custom provider using that format.

`userinfo_id_selector_policy_id`
: Optional version name for a custom `userinfo_id_selector`, or `NA_character_`. Identifies settings or helper behavior outside the function body that affect identity selection. These form part of the client's policy: a new version ID lets pending logins and saved connections detect a change.

Details

Endpoint URLs identify the provider's authorization, token, and profile services. Provider helpers fill in these URLs and suitable defaults. A separate `oauth_client()` holds your app's credentials and requested permissions. See the usage vignette for the complete setup.

Value

Calling the constructor creates an `OAuthProvider` object.

Examples

```
# Configure generic OAuth 2.0 provider (no OIDC)
generic_provider <- oauth_provider(
  name = "example",
  auth_url = "https://example.com/oauth/authorize",
  token_url = "https://example.com/oauth/token",
  # Optional URL for fetching user info:
  userinfo_url = "https://example.com/oauth/userinfo"
)

# Configure generic OIDC provider manually
# (This defaults to using nonce & ID token validation)
generic_oidc_provider <- oauth_provider_oidc(
  name = "My OIDC",
  base_url = "https://my-issuer.example.com"
)

# Look up an OIDC provider's settings with discovery
# (requires network access)
if (interactive()) {
  # Using Auth0 sample issuer as an example
  oidc_discovery_provider <- oauth_provider_oidc_discover(
    issuer = "https://samples.auth0.com/"
  )
```

```
}

# GitHub preconfigured provider
github_provider <- oauth_provider_github()

# Google preconfigured provider
google_provider <- oauth_provider_google()

# Microsoft preconfigured provider
# For a complete app using a custom tenant ID, see:
# https://lukakoning.github.io/shinyOAuth/reference/oauth_provider_microsoft.html

# Spotify preconfigured provider
spotify_provider <- oauth_provider_spotify()

# Slack via OIDC discovery
# (requires network access)
if (interactive()) {
  slack_provider <- oauth_provider_slack()
}

# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
if (interactive()) {
  options(shinyOAuth.allow_insecure_oidc_loopback = TRUE)
  oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")
}

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_auth0(domain = "your-tenant.auth0.com")
}

# Okta
# (requires configured Okta domain; example below is therefore not run)
if (interactive()) {
  oauth_provider_okta(domain = "dev-123456.okta.com")
}
```

`OAuthToken` *OAuthToken S7 class*

Description

An `OAuthToken` holds credentials and user information returned after login. The Shiny module supplies it as `auth[["token"]]`, and `handle_callback()` returns it for custom integrations. Pass it to `perform_resource_req()` to call an API, or to the token helpers for refresh, introspection, and revocation.

Properties use `@`, for example `auth[["token"]]@userinfo`. Profile fields depend on the provider.

Usage

```
OAuthToken(
  access_token = character(0),
  token_type = NA_character_,
  refresh_token = NA_character_,
  id_token = NA_character_,
  expires_at = Inf,
  userinfo = list(),
  cnf = list(),
  granted_scopes = character(0),
  granted_scopes_verified = FALSE,
  id_token_validated = FALSE,
  original_id_token = NA_character_,
  extra_fields = list(),
  initial_extra_fields = list(),
  smart_context = list(),
  original_granted_scopes = character(0)
)
```

Arguments

access_token	Access token
token_type	OAuth access token type (for example Bearer or DPoP)
refresh_token	Refresh token (if provided by the provider)
id_token	ID token (if provided by the provider; OpenID Connect)
expires_at	Numeric timestamp (seconds since epoch) when the access token expires, NA_real_ when the expiry is unknown, or Inf for a non-expiring token
userinfo	List containing user information fetched from the provider's userinfo endpoint (if fetched)
cnf	Optional binding metadata for an access token tied to a key or certificate. It can come from a token response, the access token's JWT payload, or an introspection response. For RFC 8705 certificate-bound tokens, this may contain x5t#S256 with the SHA-256 thumbprint of the client certificate that must accompany later requests. For DPoP-bound tokens, this may contain jkt with the RFC 7638 thumbprint of the public JWK bound to the token. When cnf is learned by locally parsing a raw JWT access token, shinyOAuth is observing the token payload and is not independently verifying the access-token signature. A token or introspection response received directly from the provider gives stronger confirmation.
granted_scopes	Normalized scope tokens currently associated with the access token. When a provider omits scope in a token response, shinyOAuth carries forward the best-known scope set instead of dropping it.
granted_scopes_verified	Logical flag indicating whether the current token response explicitly proved granted_scopes. FALSE means the scope set was assumed or carried forward because the provider omitted scope. With introspection_checks = "scope", provider introspection can verify the granted scope set.

`id_token_validated`
: Logical flag indicating whether the ID token was cryptographically validated (signature verified and standard claims checked) during the OAuth flow. Defaults to `FALSE`.

`original_id_token`
: Initial login ID token retained as the refresh continuity baseline. Refresh never replaces it with a newer ID token. For manually constructed tokens, the first refresh initializes this from `id_token` if omitted.

`extra_fields`
: List of additional parameters from the latest successful token endpoint response. Excludes `access_token`, `token_type`, `refresh_token`, `id_token`, `expires_in`, `scope`, and `cnf`, which have dedicated token properties. Defaults to an empty list. Successful refresh replaces this list, including when the response contains no extra fields.

`initial_extra_fields`
: List of additional parameters from the initial successful authorization-code exchange. Preserved across refreshes and replaced on a new login. Defaults to an empty list for manually constructed tokens; refresh does not infer an initial response from `extra_fields`.

`smart_context`
: Internal interpreted SMART context. Empty for ordinary tokens; populated only by SMART token processing. Use `smart_context()` on a connection to read it. Includes sensitive patient and identity references.

`original_granted_scopes`
: Initial accepted SMART grant, preserved across refreshes to distinguish unchanged grants from strict scope reductions. Empty for ordinary OAuth tokens. Set by SMART token processing.

Details

Read `token@id_token_claims` for the decoded ID-token payload as a named list, including standard and optional OIDC claims such as `sub`, `iss`, `aud`, `acr`, `amr`, `auth_time`, `nonce` and `at_hash`. This property is read-only and returns an empty list when the token is absent or cannot be decoded.

Decoding does not verify the signature. Check `id_token_validated` before using these claims as a validated identity. For validated Apple ID tokens, exact `"true"`/`"false"` strings in `email_verified` are returned as logical values, as during validation. The original signed `id_token` is retained unchanged.

Additional response parameters retain their parsed names and values, including nested lists and explicit JSON `null` values (R `NULL`). Use `"custom_field" %in% names(token@extra_fields)` to distinguish an absent field from a field explicitly returned as `null`. These parameters are not ID token claims and are not covered by `id_token_validated`. The initial snapshot records the initial response data, not current access permissions. No automatic merging, resource fetching, or interpretation is performed. `print()` and `format()` redact both lists.

Value

Calling the constructor creates an `OAuthToken` object.

Examples

```
# Inside reactive server code, after a successful login:
# auth[["token"]]@userinfo
# auth[["token"]]@expires_at
# auth[["token"]]@id_token_validated
# auth[["token"]]@id_token_claims[["sub"]]
# auth[["token"]]@extra_fields[["custom_field"]]
# auth[["token"]]@initial_extra_fields[["custom_field"]]
```

---

perform_client_bearer_req

Alias for `perform_resource_req()`

Description

[Deprecated]

Deprecated alias for `perform_resource_req()`. Use `perform_resource_req()` for Bearer, DPoP, and mTLS-protected resource requests instead.

Usage

```
perform_client_bearer_req(
  token,
  url,
  method = "GET",
  headers = NULL,
  query = NULL,
  follow_redirect = FALSE,
  check_url = TRUE,
  client = NULL,
  token_type = NULL,
  dpop_nonce = NULL,
  idempotent = NULL,
  resource_hosts = NULL,
  oauth_client = NULL
)
```

Arguments

`token`	Either an OAuthToken object or a raw access token string.
`url`	The absolute API URL, or a prepared `httr2::request()` object. The URL must be at most 8192 bytes before and after adding `query`. The helper adds current authentication and defaults to a prepared request, then applies explicit `method`, `headers`, `query` and redirect overrides. See Prepared requests for customization restrictions.

`method` Optional HTTP method (character). Defaults to "GET". When the effective token type is `DPoP`, this must be the final request method because the proof is signed against it. `TRACE` and the nonstandard `TRACK` method are rejected because authenticated requests could be reflected by the server and disclose credentials.

`headers` Optional named list or named character vector of extra headers to set on the request. Header names are case-insensitive. Names must be unique HTTP field-name tokens, and values must not contain prohibited control characters. DPoP requests reject `Host` overrides. This function manages the `Authorization` and `DPoP` headers and ignores user-supplied replacements.

`query` Optional named list of query parameters to append to the URL.

`follow_redirect` Logical or `NULL`. `FALSE` (the default) disables HTTP redirects even when `shinyOAuth.allow_redirect` is enabled. `NULL` inherits that global option (disabled by default). `TRUE` enables redirects. DPoP requests reject enabled redirects because each target needs a new proof.

`check_url` Logical. If `TRUE` (the default), validates `url` against `is_ok_host()` before attaching the access token. This rejects relative URLs, plain HTTP to non-loopback hosts, and when `options(shinyOAuth.allowed_hosts)` is set, hosts outside the allowlist. Without an allowlist this performs HTTPS and URL-syntax validation only (with the configured non-HTTPS exceptions); any HTTPS host is accepted. `FALSE` skips this policy check; URL length and resource-host restrictions still apply.

`client` Optional OAuthClient; required for DPoP or mTLS-bound tokens so the request can use the matching signing key or certificate and check any token-binding information.

`token_type` Optional override for the access token type when `token` is supplied as a raw string. Supported values are `Bearer` and `DPoP`. Invalid or multi-valued inputs are rejected. When omitted, shinyOAuth preserves `OAuthToken@token_type`, and may infer `DPoP` from explicit `OAuthToken@cnf[["jkt"]]` metadata. Raw access-token strings default to `Bearer` unless you pass `token_type = "DPoP"` explicitly.

`dpop_nonce` Optional DPoP nonce to embed in the proof for this request. This is primarily useful after a resource server challenges with `DPoP-Nonce`.

`idempotent` Whether ordinary network/HTTP failures may be retried safely. `NULL` (default) infers this from the final HTTP method: GET, HEAD, OPTIONS, PUT, and DELETE permit retries. Set it explicitly if your API has different guarantees. One DPoP nonce challenge retry is allowed independently of this setting.

`resource_hosts` Optional non-empty character vector of trusted resource host patterns, using `is_ok_host()` matching rules. This call-scoped allowlist adds to the global policy and is enforced even if `check_url` is `FALSE`. It constrains the initial URL, not redirect destinations or resolved IPs. `NULL` adds no resource-specific policy.

`oauth_client` Compatibility alias for `client`. Supply only one spelling.

Value

Same value as `perform_resource_req()`.

`perform_resource_req` *Call an API with an access token*

Description

Send an authenticated API request on the user's behalf and return an httr2 response. Pass the token from login and the API URL, then read the response with `httr2::resp_body_json()` or another httr2 response helper.

Usage

```
perform_resource_req(
  token,
  url,
  method = "GET",
  headers = NULL,
  query = NULL,
  follow_redirect = FALSE,
  check_url = TRUE,
  client = NULL,
  token_type = NULL,
  dpop_nonce = NULL,
  idempotent = NULL,
  resource_hosts = NULL,
  oauth_client = NULL
)
```

Arguments

`token`	Either an OAuthToken object or a raw access token string.
`url`	The absolute API URL, or a prepared `httr2::request()` object. The URL must be at most 8192 bytes before and after adding `query`. The helper adds current authentication and defaults to a prepared request, then applies explicit `method`, `headers`, `query` and redirect overrides. See Prepared requests for customization restrictions.
`method`	Optional HTTP method (character). Defaults to "GET". When the effective token type is `DPoP`, this must be the final request method because the proof is signed against it. `TRACE` and the nonstandard `TRACK` method are rejected because authenticated requests could be reflected by the server and disclose credentials.
`headers`	Optional named list or named character vector of extra headers to set on the request. Header names are case-insensitive. Names must be unique HTTP field-name tokens, and values must not contain prohibited control characters. DPoP requests reject `Host` overrides. This function manages the `Authorization` and `DPoP` headers and ignores user-supplied replacements.
`query`	Optional named list of query parameters to append to the URL.

follow_redirect
: Logical or NULL. FALSE (the default) disables HTTP redirects even when shinyOAuth.allow_redirect is enabled. NULL inherits that global option (disabled by default). TRUE enables redirects. DPoP requests reject enabled redirects because each target needs a new proof.

check_url
: Logical. If TRUE (the default), validates url against is_ok_host() before attaching the access token. This rejects relative URLs, plain HTTP to non-loopback hosts, and when options(shinyOAuth.allowed_hosts) is set, hosts outside the allowlist. Without an allowlist this performs HTTPS and URL-syntax validation only (with the configured non-HTTPS exceptions); any HTTPS host is accepted. FALSE skips this policy check; URL length and resource-host restrictions still apply.

client
: Optional OAuthClient; required for DPoP or mTLS-bound tokens so the request can use the matching signing key or certificate and check any token-binding information.

token_type
: Optional override for the access token type when token is supplied as a raw string. Supported values are Bearer and DPoP. Invalid or multi-valued inputs are rejected. When omitted, shinyOAuth preserves OAuthToken@token_type, and may infer DPoP from explicit OAuthToken@cnf[["jkt"]] metadata. Raw access-token strings default to Bearer unless you pass token_type = "DPoP" explicitly.

dpop_nonce
: Optional DPoP nonce to embed in the proof for this request. This is primarily useful after a resource server challenges with DPoP-Nonce.

idempotent
: Whether ordinary network/HTTP failures may be retried safely. NULL (default) infers this from the final HTTP method: GET, HEAD, OPTIONS, PUT, and DELETE permit retries. Set it explicitly if your API has different guarantees. One DPoP nonce challenge retry is allowed independently of this setting.

resource_hosts
: Optional non-empty character vector of trusted resource host patterns, using is_ok_host() matching rules. This call-scoped allowlist adds to the global policy and is enforced even if check_url is FALSE. It constrains the initial URL, not redirect destinations or resolved IPs. NULL adds no resource-specific policy.

oauth_client
: Compatibility alias for client. Supply only one spelling.

Details

The package applies its URL policy, timeouts and redirect defaults. It supports Bearer authentication and tokens tied to a key (DPoP) or certificate (mTLS). For DPoP or mTLS, client supplies the matching key or certificate.

The helper applies shinyOAuth.max_body_bytes to response bodies, including gzip decompression. Use it for both URL strings and prepared httr2 requests.

Value

An httr2 response object.

Prepared requests

Use `httr2::req_method()` and body helpers to prepare a request, then let shinyOAuth manage authentication and retries. Existing httr2 authentication, caching and retry policies, and curl authentication, method-changing or `request_target` options are rejected. HEAD requests cannot have bodies, because httr2 may send those as POST. Authenticated response caching is unsupported. Configure retries with `idempotent` and `shinyOAuth.retry_*`.

Examples

```
# Make request using OAuthToken object
# (code is not run because it requires a real token from user interaction)
if (interactive()) {
  # Inside reactive server code, after login has succeeded:
  token <- auth[["token"]]

  # Build and send in one step.
  response <- perform_resource_req(
    token,
    "https://api.example.com/resource",
    query = list(limit = 5)
  )

  # Prepare a request before sending it.
  request <- resource_req(
    token,
    "https://api.example.com/resource",
    query = list(limit = 5)
  )

  # Sending the prepared request applies the same response-size limits.
  response <- perform_resource_req(token, request)

  # Or start from your own httr2 request and still let shinyOAuth perform it
  # so response limits and DPoP nonce retries remain available.
  custom_request <- httr2::request("https://api.example.com/resource") |>
    httr2::req_headers(Accept = "application/json") |>
    httr2::req_url_query(limit = 5)

  response <- perform_resource_req(token, custom_request)

  # resource_hosts restricts this dynamic URL to api.example.com.
  response <- perform_resource_req(
    token,
    input[["resource_url"]],
    resource_hosts = "api.example.com",
    follow_redirect = FALSE
  )
}
```

`prepare_authorization_request` *Prepare a browser authorization request using GET or POST*

Description

Creates the same one-use state, PKCE and nonce as `prepare_call()`, returning a plain list describing how to send the request. Use this for applications that handle their own browser navigation and `handle_callback()`. In Shiny, the module's `request_login()` sends the request automatically.

Usage

```
prepare_authorization_request(
  client,
  browser_token,
  request_uri_publisher = NULL
)
```

Arguments

`client`	An OAuthClient object.
`browser_token`	Browser-bound token used to tie the login attempt to the current browser session.
`request_uri_publisher`	Optional function used when `request_object_mode = "request_uri"`. It must accept `request_object`, `request_handle_id`, `expires_at`, and `oauth_client` arguments and return an absolute HTTPS request-object URL that the provider can fetch.

Details

For GET, `url` is the browser navigation target. For POST, it is the action of an `application/x-www-form-urlencoded` browser form. Each `fields` entry represents one hidden input, with repeated names preserved as separate inputs. Names and values require DOM assignment or HTML escaping when constructing the form. Browser submission gives the provider access to the user's login cookies; an R HTTP request cannot perform that interaction.

The client selects `authorization_method = "POST"` explicitly; support depends on the provider. `smart_client()` additionally requires the `authorize-post` capability. The outgoing method is independent of the callback `response_mode`. Each result belongs to one login attempt with single-use state.

POST preserves the authorization endpoint's fixed query and sends newly composed fields in the body. It permits up to 256 fields and 128 KiB of encoded form data; CR/LF and the browser-reserved `_charset_` field are rejected to prevent the browser changing field values. Configure your app's Content Security Policy `form-action` to allow the authorization endpoint. Custom callers must preserve browser binding and callback handling.

Value

A list with `method` (`"GET"` or `"POST"`), `url` and `fields`. `fields` is empty for GET; for POST it is a list of lists, each with scalar character `name` and `value`. PAR expiry attributes are preserved as documented in `prepare_call()`. The result contains transient authorization data for that login attempt.

Examples

```
provider <- oauth_provider(
  name = "Example service",
  auth_url = "https://example.com/authorize",
  token_url = "https://example.com/token",
  token_auth_style = "public"
)
client <- oauth_client(
  provider = provider,
  client_id = "example-client",
  redirect_uri = "http://127.0.0.1:8100/callback"
)
# In a custom browser flow, bind this secret to the initiating browser and
# supply it again to handle_callback(). Shiny modules manage this for you.
browser_token <- paste(format(openssl::rand_bytes(64)), collapse = "")
request <- prepare_authorization_request(client, browser_token)
request[["method"]]
```

---

prepare_call *Prepare an OAuth 2.0 authorization request and build its URL*

Description

Prepare a login request and return the URL to open in the user's browser. Use this when your application controls the browser redirect and callback handling itself but needs shinyOAuth to construct the OAuth 2.0 authorization request. Pair it with `handle_callback()` to complete the code flow.

Usage

```
prepare_call(
  client,
  browser_token,
  request_uri_publisher = NULL,
  oauth_client = NULL
)
```

Arguments

`client`	An OAuthClient object.
`browser_token`	Browser-bound token used to tie the login attempt to the current browser session.

`request_uri_publisher`
: Optional function used when `request_object_mode = "request_uri"`. It must accept `request_object`, `request_handle_id`, `expires_at`, and `oauth_client` arguments and return an absolute HTTPS request-object URL that the provider can fetch.

`oauth_client`
: Compatibility alias for `client`. Supply only one spelling.

Details

In a Shiny app using `oauth_module_server()`, call `auth[["request_login"]]()` to start login through the module, which manages both operations and the reactive session state.

The helper records one-time state and creates any required PKCE and nonce values. Custom callers must preserve the browser binding and process the returning callback themselves. For an explicitly configured POST client, use `prepare_authorization_request()` instead. This URL-only helper rejects POST before storing a transaction.

Value

A length-1 string containing the authorization URL to send the user to. When PAR is used, the returned string also carries `shinyOAuth.par_request_uri`, `shinyOAuth.par_expires_in`, and `shinyOAuth.par_expires_at` attributes so callers can tell when the pushed authorization request should be regenerated.

Examples

```
# Advanced example: your code supplies browser redirects and callback handling.
# For a Shiny app, oauth_module_server() manages these steps for you.

if (interactive()) {
  # Define client
  client <- oauth_client(
    provider = oauth_provider_github(),
    client_id = Sys.getenv("GITHUB_OAUTH_CLIENT_ID"),
    client_secret = Sys.getenv("GITHUB_OAUTH_CLIENT_SECRET"),
    redirect_uri = "http://127.0.0.1:8100"
  )

  # Get the login URL and store state in client's state store
  # `<browser_token>` must be unpredictable and persisted for this transaction
  # in storage bound to the application's exact origin (scheme, host, port).
  # The module combines origin-scoped storage with an independent marker cookie
  # and checks both on return. A cookie alone does not provide this boundary:
  # cookies can be shared by applications on different ports of the same host.
  # Shiny applications should use oauth_module_server() for the complete flow.
  authorization_url <- prepare_call(client, "<browser_token>")

  # Redirect user to authorization URL; retrieve code & state from the query;
  # recover this transaction's `<browser_token>` through the origin-bound flow
  # and verify its independent marker before calling handle_callback().
  code <- "..."
  state <- "..."
```

```
  browser_token <- "..."

  # Handle callback, exchanging code for token and validating state
  token <- handle_callback(client, code, state, browser_token)
}
```

`refresh_token` *Refresh an OAuth 2.0 token*

Description

Obtains a new access token using a refresh token, without an interactive login. This low-level helper returns a new OAuthToken; it does not replace the input object. `oauth_module_server()` and `oauth_connections_server()` manage token replacement and automatic refresh for their connections.

Usage

```
refresh_token(
  client,
  token,
  async = FALSE,
  introspect = NULL,
  shiny_session = NULL,
  oauth_client = NULL
)
```

Arguments

`client`	OAuthClient object
`token`	OAuthToken object containing the refresh token
`async`	If `TRUE`, return a promise resolving to the result, using configured mirai daemons or a future plan; mirai takes priority. A non-sequential backend moves work outside the main R process. Default `FALSE` waits and returns the result directly.
`introspect`	`NULL` (default) or a logical. After a successful refresh, introspect the new access token when either this argument is `TRUE` or the client was configured with `introspect = TRUE`. A per-call `FALSE` cannot disable a configured client requirement. When enabled, refresh fails if introspection is unsupported, inactive, or missing required `introspection_checks`. The raw introspection result is not stored separately, but a successful introspection response may backfill `token@cnf`.
`shiny_session`	Optional captured Shiny session details for audit events, normally supplied by the module.
`oauth_client`	Compatibility alias for `client`. Supply only one spelling.

Details

The provider may replace the refresh token too; otherwise the old refresh token is kept. Required userinfo is fetched again, and configured client introspection must succeed before the refreshed token is returned.

For ordinary OAuth clients, refresh explicitly requests the token's retained `granted_scopes` when known. This preserves prior scope reductions and makes an omitted response `scope` refer to that requested set. Tokens without known scopes omit the request parameter. SMART uses its own original-grant rules.

OIDC refresh responses may omit the ID token, in which case the original is kept. If a new ID token is returned, an original must be available and the subject, issuer, and audience must remain consistent, as must `auth_time` and nonce when applicable. Full signature and claim validation runs when `id_token_validation = TRUE`. Userinfo is checked against a validated ID token when both are available; `userinfo_id_token_match = TRUE` requires that baseline.

Refresh preserves the original interactive login time. The module's `reauth_after_seconds` sets a separate limit on that login's age. A returned ID token must have an `iat` at or after the refresh request start, allowing the provider's configured clock leeway and same-second issuance.

Within one R process, overlapping asynchronous calls for the same client and refresh token share one promise and result. Token snapshots, client settings, and validation options must match; conflicting calls fail before dispatch. A synchronous or reentrant call while that refresh is pending raises an error. Coordination covers one R process; completed results are not cached. The module manages token replacement and checks that the result still belongs to the current login. Direct calls return a value object without that session-level assignment.

Refresh errors carry a non-secret `refresh_credential_outcome` field: `"not_consumed"`, `"consumed"`, `"possibly_consumed"`, or `"rejected"`. Only `"not_consumed"` permits retrying the input refresh credential. Other outcomes, including an unavailable worker result, require new authorization. The module applies this rule automatically, even with indefinite sessions; unvalidated access or identity data is never accepted.

Value

An updated OAuthToken object with refreshed credentials.

Updated fields:

- `access_token`: Always updated to the fresh token
- `expires_at`: Computed from `expires_in` when provided; otherwise a fallback lifetime set by `shinyOAuth.default_expires_in` (3600 seconds by default)
- `refresh_token`: Updated if the provider rotates it; otherwise preserved
- `id_token`: Updated only if the provider returns one (and it validates); otherwise the latest stored ID token is preserved
- `original_id_token`: Retained from login for continuity checks, even if intermediate refresh ID tokens omit `nonce` or `auth_time`
- `userinfo`: Refreshed if `userinfo_required = TRUE`; otherwise preserved
- `extra_fields`: Replaced by the additional parameters in the refresh response, or an empty list if none are returned. Not merged with earlier responses; explicit JSON `null` values remain named `NULL` entries.

- `initial_extra_fields`: Preserved from the initial code exchange. This historical snapshot does not establish current access permissions.
- `cnf`: Updated from the token response when present, and may be backfilled from refresh-time introspection when enabled. When the refresh response omits new observable `cnf`, shinyOAuth does not carry forward a prior `x5t#S256` thumbprint onto the refreshed token; mTLS sender-constrained state is kept only when the new token or its introspection response supplies fresh `cnf`

Refresh fails if a new ID token fails validation, or if the UserInfo subject differs from the new ID token's subject. Validation includes issuer, audience, expiry and agreement with the original login's subject. In `oauth_module_server()`, this clears the session and sets `authenticated = FALSE`, unless `indefinite_session = TRUE` keeps it with `token_stale = TRUE`.

Examples

```
# get_userinfo(), introspect_token(), and refresh_token() are typically
# called by oauth_module_server() according to your provider/client and
# module settings, rather than directly by application code. The module
# also calls revoke_token() during logout when the provider supports it.
# These helpers are exported for custom login flows, on-demand profile or
# token checks, and applications that manage token lifetime themselves.
#
# The examples below require a real token from a completed login.
# Inside a reactive expression in server(), after creating auth with
# oauth_module_server() and confirming auth[["authenticated"]]:
if (interactive()) {
  token <- auth[["token"]]
  user_info <- get_userinfo(client, token)

  # Requires an introspection endpoint. NA means activity is unknown.
  result <- introspect_token(client, token)
  isTRUE(result[["active"]])

  # Returns replacement credentials using the refresh token.
  token <- refresh_token(client, token)

  # Requires a revocation endpoint to invalidate the token at the provider.
  result <- revoke_token(client, token, token_kind = "refresh")
}
```

`resource_req` *Prepare an API request with an access token*

Description

Build an httr2 request that uses the user's access token. Use this when you want to inspect or customize a request before sending it. To build and send in one step, use `perform_resource_req()`.

Usage

```
resource_req(
  token,
  url,
  method = "GET",
  headers = NULL,
  query = NULL,
  follow_redirect = FALSE,
  check_url = TRUE,
  client = NULL,
  token_type = NULL,
  dpop_nonce = NULL,
  resource_hosts = NULL,
  oauth_client = NULL
)
```

Arguments

token	Either an OAuthToken object or a raw access token string.
url	The absolute URL to call. Resource URLs must not exceed 8192 bytes, including the query and fragment, both before and after applying query. This limit also applies when check_url = FALSE.
method	Optional HTTP method (character). Defaults to "GET". When the effective token type is DPoP, this must be the final request method because the proof is signed against it. TRACE and the nonstandard TRACK method are rejected because authenticated requests could be reflected by the server and disclose credentials.
headers	Optional named list or named character vector of extra headers to set on the request. Header names are case-insensitive. Names must be unique HTTP field-name tokens, and values must not contain prohibited control characters. DPoP requests reject Host overrides. This function manages the Authorization and DPoP headers and ignores user-supplied replacements.
query	Optional named list of query parameters to append to the URL.
follow_redirect	Logical or NULL. FALSE (the default) disables HTTP redirects even when shinyOAuth.allow_redirect is enabled. NULL inherits that global option (disabled by default). TRUE enables redirects. DPoP requests reject enabled redirects because each target needs a new proof.
check_url	Logical. If TRUE (the default), validates url against is_ok_host() before attaching the access token. This rejects relative URLs, plain HTTP to non-loopback hosts, and when options(shinyOAuth.allowed_hosts) is set, hosts outside the allowlist. Without an allowlist this performs HTTPS and URL-syntax validation only (with the configured non-HTTPS exceptions); any HTTPS host is accepted. FALSE skips this policy check; URL length and resource-host restrictions still apply.
client	Optional OAuthClient; required for DPoP or mTLS-bound tokens so the request can use the matching signing key or certificate and check any token-binding information.

`token_type` Optional override for the access token type when `token` is supplied as a raw string. Supported values are `Bearer` and `DPoP`. Invalid or multi-valued inputs are rejected. When omitted, shinyOAuth preserves `OAuthToken@token_type`, and may infer `DPoP` from explicit `OAuthToken@cnf[["jkt"]]` metadata. Raw access-token strings default to `Bearer` unless you pass `token_type = "DPoP"` explicitly.

`dpop_nonce` Optional DPoP nonce to embed in the proof for this request. This is primarily useful after a resource server challenges with `DPoP-Nonce`.

`resource_hosts` Optional non-empty character vector of trusted resource host patterns, using `is_ok_host()` matching rules. This call-scoped allowlist adds to the global policy and is enforced even if `check_url` is `FALSE`. It constrains the initial URL, not redirect destinations or resolved IPs. `NULL` adds no resource-specific policy.

`oauth_client` Compatibility alias for `client`. Supply only one spelling.

Details

The package applies its URL policy, timeouts and redirect defaults. It supports Bearer authentication and tokens tied to a key (DPoP) or certificate (mTLS). For DPoP or mTLS, `client` supplies the matching key or certificate.

`perform_resource_req()` sends the prepared request with the package's response-size limits. `httr2::req_perform()` uses httr2's transport settings.

Value

An httr2 request object, accepted by `perform_resource_req()` for bounded response handling. A DPoP proof is bound to the method and base URL at preparation time; later changes to either invalidate it.

Advanced request restrictions

The helper rejects additional `access_token` credentials in ordinary URL and form inputs. File, multipart and streaming bodies, and changes made after preparation, are outside that check.

`perform_resource_req()` limits both downloaded bytes and decoded gzip data. Direct httr2 execution does not provide shinyOAuth's decoded-response limit; its curl limit depends on the installed libcurl version. Prepared requests can be sent with `perform_resource_req(token, request, client = client)`.

DPoP note

DPoP proofs bind the current HTTP method and target URI (without query or fragment). Literal `.` and `..` path segments are removed before signing and sending; percent-encoded path segments remain escaped. Use the `query` argument to preserve encoded resource paths; external URL modifiers can decode reserved path characters. Changing the method, scheme, host, or path invalidates the proof. Automatic redirects are unsupported; each target URL requires its own proof.

Examples

```
# Make request using OAuthToken object
# (code is not run because it requires a real token from user interaction)
if (interactive()) {
  # Inside reactive server code, after login has succeeded:
  token <- auth[["token"]]

  # Build and send in one step.
  response <- perform_resource_req(
    token,
    "https://api.example.com/resource",
    query = list(limit = 5)
  )

  # Prepare a request before sending it.
  request <- resource_req(
    token,
    "https://api.example.com/resource",
    query = list(limit = 5)
  )

  # Sending the prepared request applies the same response-size limits.
  response <- perform_resource_req(token, request)

  # Or start from your own httr2 request and still let shinyOAuth perform it
  # so response limits and DPoP nonce retries remain available.
  custom_request <- httr2::request("https://api.example.com/resource") |>
    httr2::req_headers(Accept = "application/json") |>
    httr2::req_url_query(limit = 5)

  response <- perform_resource_req(token, custom_request)

  # resource_hosts restricts this dynamic URL to api.example.com.
  response <- perform_resource_req(
    token,
    input[["resource_url"]],
    resource_hosts = "api.example.com",
    follow_redirect = FALSE
  )
}
```

`revoke_token` *Revoke an OAuth 2.0 token*

Description

Ask the provider to invalidate an access or refresh token, for example when a user disconnects their account or your application disposes of stored credentials. The provider must support token revocation. The Shiny module calls this during logout; use `auth[["logout"]]()` to also clear its local session. Revocation does not end the user's login session at the provider.

Usage

```
revoke_token(
  client,
  token,
  token_kind = c("refresh", "access"),
  async = FALSE,
  shiny_session = NULL,
  oauth_client = NULL,
  oauth_token = NULL,
  which = NULL
)
```

Arguments

client	OAuthClient object
token	OAuthToken object containing tokens to revoke
token_kind	Which token to revoke: "refresh" (default) or "access"
async	If TRUE, return a promise resolving to the result, using configured mirai daemons or a future plan; mirai takes priority. A non-sequential backend moves work outside the main R process. Default FALSE waits and returns the result directly.
shiny_session	Optional captured Shiny session details for audit events, normally supplied by the module.
oauth_client	Compatibility alias for client. Supply only one spelling.
oauth_token	Compatibility alias for token. Supply only one spelling.
which	Compatibility alias for token_kind. Supply only one spelling.

Details

Uses the client's configured credentials and token_auth_style. The returned status distinguishes an absent endpoint or token from an unsuccessful HTTP response; these leave the revocation result unknown. A successful response means the provider accepted the request; local logout does not depend on it.

Value

A list with fields:

- supported: logical, TRUE when a revocation endpoint is configured.
- revoked: logical or NA, TRUE when the provider accepted the revocation request, NA when revocation could not be attempted or the result is unknown.
- status: machine-readable status such as "ok", "missing_token", "revocation_unsupported", or "http_<code>".

Examples

```
# get_userinfo(), introspect_token(), and refresh_token() are typically
# called by oauth_module_server() according to your provider/client and
# module settings, rather than directly by application code. The module
# also calls revoke_token() during logout when the provider supports it.
# These helpers are exported for custom login flows, on-demand profile or
# token checks, and applications that manage token lifetime themselves.
#
# The examples below require a real token from a completed login.
# Inside a reactive expression in server(), after creating auth with
# oauth_module_server() and confirming auth[["authenticated"]]:
if (interactive()) {
  token <- auth[["token"]]
  user_info <- get_userinfo(client, token)

  # Requires an introspection endpoint. NA means activity is unknown.
  result <- introspect_token(client, token)
  isTRUE(result[["active"]])

  # Returns replacement credentials using the refresh token.
  token <- refresh_token(client, token)

  # Requires a revocation endpoint to invalidate the token at the provider.
  result <- revoke_token(client, token, token_kind = "refresh")
}
```

---

smart_client *Configure a SMART on FHIR app registration*

Description

[Experimental]

Configure your Shiny app to access a hospital's FHIR API. Start with `smart_discover()`, then supply the app registration and requested permissions. The result is an OAuthClient for `oauth_module_server()` or `oauth_connections()`, with an approved API named `"fhir"`.

Usage

```
smart_client(
  discovery,
  client_id,
  redirect_uri,
  scopes,
  required_scopes = scopes,
  launch = c("standalone", "ehr"),
  identity = c("none", "openid", "fhirUser"),
  allow_v1_scopes = FALSE,
  online_access_policy = c("online_only", "allow_offline"),
```

```
  token_auth_style = c("public", "header", "private_key_jwt"),
  client_secret = character(),
  client_assertion_private_key = NULL,
  client_assertion_private_key_kid = NULL,
  client_assertion_alg = "RS384",
  authorization_method = "GET",
  response_mode = NULL,
  authorization_server_mode = "single",
  authorization_server_redirect_uris = character(),
  initial_expires_in_fallback = NULL,
  label = "FHIR server",
  state_store = cachem::cache_mem(max_age = 300),
  state_key = random_urlsafe(128),
  state_payload_max_age = 300
)
```

Arguments

`discovery`	A plain snapshot returned by `smart_discover()`. Its metadata and endpoint policy are revalidated locally; this performs no network calls.
`client_id`, `redirect_uri`	App registration values. The callback must use HTTPS, except for the snapshot's explicit HTTP loopback development policy.
`scopes`	Permissions to request, without automatic wildcard/offline access. Standalone patient scopes require `launch/patient`. EHR clients add `launch`. `online_access` requires EHR launch and `permission-online`; `offline_access` requires `permission-offline` in either launch mode. Each resource scope spelling must be advertised through `permission-v2` or, for v1 spellings, `permission-v1` with `allow_v1_scopes = TRUE`. A SMART scope comparison supports at most 256 distinct scopes on each side and 64 KiB (65,536 bytes) of combined scope text. Larger comparisons are rejected, including during token acceptance and refresh.
`required_scopes`	Minimum permissions, defaulting to `scopes`. Pass a subset to accept reduced grants as limited connections. Identity scopes are always required when identity is enabled. Unsupported comparisons are rejected.
`launch`	`"standalone"` or `"ehr"`, matching the registered app flow.
`identity`	`"none"` (default), `"openid"` for a validated OIDC subject, or `"fhirUser"` to also require the user's FHIR reference. `"openid"` does not interpret a `fhirUser` claim or populate `smart_context()[["fhirUser"]]`. A validated `fhirUser` claim may be an absolute URL or a supported resource instance reference relative to this client's FHIR base, such as `"Practitioner/example"` or `"Practitioner/example/_histor` Versioned references retain their version.
`allow_v1_scopes`	Explicit compatibility flag enabling `.read`, `.write` and `.*`. Requesting these spellings requires `permission-v1`; requesting v2 spellings requires `permission-v2`, including when this flag is enabled. Default `FALSE`.

`online_access_policy`
: `"online_only"` (default) or `"allow_offline"`. SMART permits an `online_access` request to negotiate `offline_access`. Opt in to `"allow_offline"` to accept that longer-lived permission in place of required `online_access`. The default rejects this substitution, even when `online_access` is optional. Explicitly requesting `offline_access` also authorizes offline persistence. Granted scopes retain their actual spelling; refresh responses cannot escalate an existing online grant.

`token_auth_style`
: Registration type: `"public"`, `"header"` for a symmetric secret using HTTP Basic, or `"private_key_jwt"`. Selection must agree with advertised capabilities. Confidential methods must also agree with authentication metadata; public clients do not authenticate and need no `"none"` entry in that metadata. Later property edits must retain a supported SMART token authentication method. Asymmetric assertions require `typ = "JWT"`, a key ID, an explicit RS384/ES384 algorithm, and the token endpoint as their audience.

`client_secret`
: Secret for a symmetric registration; otherwise omit.

`client_assertion_private_key`, `client_assertion_private_key_kid`
: Private signing key and registered key ID for asymmetric authentication; see `oauth_client()`. Required for `"private_key_jwt"`.

`client_assertion_alg`
: `"RS384"` (default) or `"ES384"`; the key and server metadata must support the selected algorithm. Other styles omit assertions.

`authorization_method`
: `"GET"` (default) or `"POST"` for the outgoing browser request. POST requires the discovered `authorize-post` capability and uses the module's `request_login()` or `prepare_authorization_request()`. POST supports longer browser requests but retains the scope comparison limits described under `scopes`.

`response_mode`
: `NULL`, `"query"`, or `"form_post"`. Advertised response modes, when present, must allow the selection.

`authorization_server_mode`, `authorization_server_redirect_uris`
: See `oauth_client()`. Multiple clients use distinct registered callback routes.

`initial_expires_in_fallback`
: Optional positive lifetime in seconds, supplied by the authorization server out of band for initial access tokens. Used only when an initial response omits `expires_in`; an explicit response value takes precedence. Default `NULL` requires the response to include a lifetime.

`label`
: Display label, default `"FHIR server"`.

`state_store`, `state_key`, `state_payload_max_age`
: See `oauth_client()`.

Details

`launch = "standalone"` describes an app opened directly; `"ehr"` describes one opened from an electronic health record. Each flow has its own app registration. Patient context identifies the chart being viewed; `identity` separately controls validation of the signed-in user. See `vignette("smart-on-fhir", package = "shinyOAuth")` (online guide) for a complete app.

Value

An OAuthClient for the module or connection manager. Its `resource_bases` contains the approved `fhir` API; its `smart` property records the selected profile.

Advanced SMART profile

The constructor applies SMART App Launch 2.2 permission rules, S256 PKCE and the exact FHIR base as the authorization request's `aud`. `identity = "openid"` requires signed identity-token and nonce checks; `"fhirUser"` also requires the user's FHIR reference.

Supported flows use direct authorization and query or form POST callbacks. Claims requests, remote authentication-freshness overrides, JAR, PAR, JARM, DPoP and mTLS are not supported by this constructor. The manager establishes a fresh registered transaction for each EHR launch; launch handles are single-use.

Tokens need a positive lifetime. Only an initial response may omit `expires_in`, and only with an explicit `initial_expires_in_fallback`. Refresh responses must supply a lifetime. Requests require TLS 1.2 or newer.

References

SMART 2.2 launch and client authentication.

Examples

```
## Not run:
site <- smart_discover("https://ehr.example/fhir/R4")
client <- smart_client(site, "registered-app", "https://app.example/callback",
  scopes = c("launch/patient", "patient/Patient.r"),
  required_scopes = "patient/Patient.r", token_auth_style = "public")
client@smart[["launch"]]

## End(Not run)
```

`smart_context` *Read a connection's current SMART context*

Description

[Experimental]

Returns the patient and encounter selected for a connection, plus validated user references when requested. This context identifies the current chart and user; raw token extensions remain available separately on tokens.

Usage

```
smart_context(connection)

smart_patient(connection)

smart_fhir_user(connection)
```

Arguments

connection	An OAuthConnection for a smart_client(), accessed in its owning Shiny session.

Details

smart_patient() fetches the contextual Patient only when the current grant covers patient/Patient.r or user/Patient.r. smart_fhir_user() fetches the identity reference only from a validated ID token. It accepts openid fhirUser, a matching user read scope, or patient read permission when the identity is the contextual Patient. Both stay inside the FHIR base and refuse redirects. Both request JSON with Accept: application/fhir+json. A foreign fhirUser reference is reported as context but is never fetched with this connection's token. FHIR search, write, batch and pagination helpers are outside these two convenience methods.

Patient IDs and user identity are different: the first identifies a chart, the second the authenticated user. General summaries omit this data. Experimental fhirContext and styling extensions are retained as raw data. Location-specific authorization_details apply scope, patient and encounter overrides only for the configured FHIR base, with omitted fields falling back to this response's top-level values. Other locations never add destinations. Malformed details or multiple entries matching this base are rejected.

Value

smart_context() returns a list with version, fhir_base, revision, changed, patient, encounter, fhirUser and need_patient_banner. Absent values are NULL. revision increases when interpreted context changes. Together, connection ID and revision identify the context for patient-dependent data. Refresh omission carries context forward, except that a changed or cleared patient clears an omitted encounter. Explicit null clears a field, unless patient access still requires patient context. An initial launch query never establishes this context.

smart_patient() and smart_fhir_user() return an httr2 response.

See Also

smart_client(), OAuthConnection

Examples

```
# Call these helpers with an authorized connection in its owning Shiny session.
# Reading the patient requires the corresponding patient or user read scope.
read_current_patient <- function(connection) {
  context <- smart_context(connection)
  if (is.null(context[["patient"]])) return(NULL)
  httr2::resp_body_json(smart_patient(connection))
}

# The signed-in user can be different from the patient whose chart is open.
read_signed_in_user <- function(connection) {
  httr2::resp_body_json(smart_fhir_user(connection))
}
```

`smart_discover` *Discover SMART on FHIR server metadata*

Description

[Experimental]

Read and validate the SMART App Launch STU 2.2 discovery document for a configured FHIR server. Returns server metadata for application setup.

Usage

```
smart_discover(fhir_base, endpoint_hosts = NULL, allow_http_loopback = FALSE)
```

Arguments

`fhir_base`	Trusted FHIR base URL, including its complete path, for example `"https://ehr.example/fhir/R4"`.
`endpoint_hosts`	Character vector of exact permitted hostnames for recognized metadata URLs. `NULL` defaults to the FHIR base hostname. An explicit vector replaces that default; include every permitted host. Hostnames are case-insensitive. Wildcards, URLs, and ports are not accepted.
`allow_http_loopback`	Logical, default `FALSE`. Explicit development exception permitting HTTP only at `localhost`, `127.0.0.1`, or `::1`. Applies to the base and metadata URLs.

Details

Call once during application setup, outside `server()`. The request appends `/.well-known/smart-configuration` to the full FHIR base, removing its terminal slash first. The supplied base and returned endpoint/issuer strings are retained exactly.

This is separate from `oauth_provider_oidc_discover()`. OAuth-only SMART metadata need not contain OIDC issuer or signing-key information. Advertised `sso-openid-connect` requires both `issuer` and `jwks_uri`; their presence does not validate an identity or fetch OIDC metadata or keys.

Required SMART fields, conditional launch/SSO fields, and advertised asymmetric-authentication metadata are checked. S256 must be advertised and plain PKCE is rejected. Client-specific capability, scope, key and algorithm selection belongs to the application's registration configuration. `scopes_supported` is informative, not an exhaustive permission allowlist. Its array may be empty. An empty `token_endpoint_auth_methods_supported` array permits public registrations; confidential registrations still require their selected method whenever that array is present.

Value

A plain named list with `fhir_base` (the supplied identifier), `discovery_url` (the requested URL), `smart_version` (`"2.2.0"`, the validation baseline, not a detected server version), `metadata` (the parsed document, with JSON arrays as lists), `endpoint_hosts` (the normalized policy), and `allow_http_loopback`. Invalid input or endpoint policy raises a `shinyOAuth_config_error`; malformed metadata raises a `shinyOAuth_parse_error`; failed HTTP requests raise a `shinyOAuth_http_error`.

Network and trust policy

The FHIR base is a deployment trust setting. EHR launch routes select an already configured client rather than discovering a browser-supplied base. The base must be an absolute HTTPS URL without userinfo, query, fragment, or ambiguous path syntax. Discovered URLs must be absolute; this reader does not repair relative URLs from legacy servers.

`endpoint_hosts` applies to the issuer and these top-level URL fields when present: `authorization_endpoint`, `token_endpoint`, `jwks_uri`, `registration_endpoint`, `management_endpoint`, `introspection_endpoint`, `revocation_endpoint`, `userinfo_endpoint`, `pushed_authorization_request_endpoint`, `smart_app_state_endpoint` and `user_access_brand_bundle`. Matching uses exact hostnames, independently of port. This is a discovery policy; it does not authorize resource requests. The generic `shinyOAuth.allowed_hosts` option can further restrict these URLs.

Redirects are refused even when the generic redirect option is enabled. JSON must be an object with no duplicate members; arrays and recognized URL fields have bounded validation. Errors do not include response bodies or returned metadata values.

Unknown extensions, including `associated_endpoints`, are retained as data only. Their URLs are not fetched or approved for credential use. Each call reads a fresh snapshot.

References

SMART STU 2.2 discovery and asymmetric client metadata.

See Also

`oauth_provider_oidc_discover()`, `smart_client()`

Examples

```
## Not run:
site <- smart_discover(
  "https://ehr.example/fhir/R4",
  endpoint_hosts = c("ehr.example", "login.example")
)
site[["metadata"]][["token_endpoint"]]
site[["metadata"]][["capabilities"]]


## End(Not run)
```

`smart_launch_route` *Register a SMART EHR launch entry route*

Description

[Experimental]

Declare which approved SMART clients an EHR launch URL may select. Pass the result in `launch_routes` to `oauth_connections_ui()`. The route is separate from OAuth callbacks: `iss` means the FHIR base only here. Both `iss` and `launch` are required, and callback parameters are rejected on this route.

Usage

```
smart_launch_route(path, client_names, max_age = 120)
```

Arguments

path	Absolute application path, such as "/smart/launch". Register paste0(manager[["app_origin"]], path) with the EHR. It must be inside the UI's app_base_path and distinct from every callback and other launch route. Accepted percent-encoded unreserved characters are stored decoded.
client_names	Character vector of names from the manager's clients list, not OAuth client_id values. Each must select an EHR-mode smart_client(). A route cannot contain two registrations for the same exact FHIR base; give those registrations separate launch routes.
max_age	Launch handoff lifetime in seconds, 30 to 300, default 120. The handoff must be consumed and its authorization parameters prepared before this deadline. Once prepared, login uses the client's state_payload_max_age, bounded by owner expiry; the handoff deadline does not shorten consent time.

Details

Initial entry is untrusted. It selects an already configured client by exact FHIR base, performs no discovery, and establishes no healthcare identity. A short-lived encrypted record binds the opaque launch handle to the browser owner and client. A clean continuation proves that owner before a fresh OAuth state/PKCE transaction can begin. Consumed handles cannot be reused to reconnect.

EHR entry supports top-level GET navigation with browser retention in one R process. Account/session-only retention and iframe deployments are not supported for EHR entry. Standalone launch supports all manager retention choices. Up to eight unconsumed launch records are retained per browser owner across all of a manager's launch routes, with a total limit of 1,000 per manager. Further entries are rejected without evicting existing tickets. Consuming or expiring a ticket releases its capacity. Raw query size is limited to 8 KiB, launch handles to 2 KiB, and the only allowed initial parameters are iss and launch.

The owner quota limits each browser's pending work. Unauthenticated callers can obtain additional browser owners, so network-level ingress limits belong to the hosting layer.

The HTTP response uses no-store and no-referrer policy and redirects to an opaque, owner-bound continuation. The package's external JavaScript removes that ticket from browser history before Shiny connects.

Value

A plain route configuration list, with no credentials or live state.

References

SMART EHR launch

See Also

smart_client(), oauth_connections_ui()

Examples

```
## Not run:
# hospital is an EHR-mode SMART client; manager uses browser retention.
ui <- oauth_connections_ui(app_ui, "health", manager,
  launch_routes = list(smart_launch_route("/smart/launch", "hospital")))

## End(Not run)
```

---

use_shinyOAuth *Add JavaScript dependency to the UI of a Shiny app*

Description

Adds shinyOAuth's JavaScript so `oauth_module_server()` can redirect the browser and manage its temporary login cookie. This low-level helper supplies the browser dependency for custom HTTP integrations.

`oauth_ui()`, `oauth_form_post_ui()` and `oauth_connections_ui()` already include this dependency, callback handling and response headers.

Usage

```
use_shinyOAuth(inject_referrer_meta = TRUE)
```

Arguments

`inject_referrer_meta`
: If TRUE (default), adds a meta tag to the page: an instruction asking the browser not to share the page's address when loading images, scripts, or other files. Some files may start loading before the browser reads this instruction. The UI wrappers also set the HTTP header, which applies from the start of page loading.

Details

The UI wrappers process callbacks before rendering application HTML, redirect to a clean URL, and send `Referrer-Policy: no-referrer`, `Cache-Control: no-store` and `Pragma: no-cache`. A custom HTTP integration provides that handling separately. This dependency cannot set HTTP headers; its optional referrer meta tag takes effect only after the browser reads it.

Value

A `tagList` that loads the browser code once.

See Also

`oauth_ui()`, `oauth_form_post_ui()`, `oauth_connections_ui()`, `oauth_module_server()`

Examples

```
# Low-level browser setup for a custom HTTP integration.
# Supply equivalent callback handling and response headers separately.
# For ordinary apps, use oauth_ui(ui, id = "auth", client = client).
ui <- shiny::fluidPage(
  use_shinyOAuth()
  # ...
)
```

Index

base::print(), *113*

cachem::cache_disk(), *8*
cachem::cache_mem(), *8*
check_oauth21, 3
client_bearer_req, 5
custom_cache, 6
custom_cache(), *22*, *46*, *63*, *93*, *100*, *119*

get_userinfo, 10
get_userinfo(), *61*, *118*

handle_callback, 11
handle_callback(), *24*, *98*, *123*, *131, 132*
httr2, *34*, *112*, *128, 129*, *136*, *138*, *145*
httr2::req_body_form(), *112*
httr2::req_body_json(), *112*
httr2::req_body_raw(), *112*
httr2::req_headers(), *112*
httr2::req_method(), *130*
httr2::req_perform(), *138*
httr2::request(), *112*, *126*, *128*
httr2::resp_body_json(), *128*

introspect_token, 13
is_ok_host, 15
is_ok_host(), *6*, *50*, *127*, *129*, *137, 138*

oauth_account_owner *(*oauth_browser_owner*)*, 16
oauth_account_owner(), *37*
oauth_browser_owner, 16
oauth_browser_owner(), *37*, *44*
oauth_client, 19
oauth_client(), *7*, *13*, *21*, *25, 26*, *31–34*, *36, 37*, *48, 49*, *52*, *57*, *60*, *62*, *67–76*, *82*, *84*, *86–91*, *93*, *99, 100*, *104, 105*, *117, 118*, *122*, *143*
oauth_client_mtls_registration, 30
oauth_client_secret_apple, 32
oauth_client_secret_apple(), *67*
oauth_connection, 33
oauth_connection(), *29*, *36*, *103*, *107*, *109*, *113*
oauth_connection_store_memory, 35
oauth_connection_store_memory(), *19*, *38*
oauth_connections, 37
oauth_connections(), *16*, *29*, *34, 35*, *40*, *43*, *103*, *141*
oauth_connections_server, 40
oauth_connections_server(), *34*, *37*, *39*, *43*, *106*, *134*
oauth_connections_ui, 43
oauth_connections_ui(), *17*, *37*, *39, 40*, *147–149*
oauth_form_post_ui, 45
oauth_form_post_ui(), *12*, *24*, *48*, *50*, *52*, *67*, *92, 93*, *98*, *149*
oauth_module_server, 48
oauth_module_server(), *7, 8*, *11, 12*, *19*, *24*, *27*, *34*, *44–46*, *92, 93*, *98*, *102*, *105, 106*, *133, 134*, *141*, *149*
oauth_provider, 57
oauth_provider(), *7*, *61–64*, *69, 70*, *72*, *81*, *83–85*, *90*, *113*, *117, 118*, *120*
oauth_provider_apple, 67
oauth_provider_auth0, 68
oauth_provider_authentik, 69
oauth_provider_cognito, 70
oauth_provider_github, 71
oauth_provider_gitlab, 72
oauth_provider_globus, 73
oauth_provider_google, 74
oauth_provider_google(), *21*, *95*, *113*
oauth_provider_huggingface, 74
oauth_provider_keycloak, 75
oauth_provider_microsoft, 76
oauth_provider_oidc, 80
oauth_provider_oidc(), *62, 63*, *118*, *120*
oauth_provider_oidc_discover, 82

oauth_provider_oidc_discover(), *21*, *57*, *63*, *80*, *95*, *113*, *120*, *146*, *147*
oauth_provider_okta, 86
oauth_provider_orcid, 87
oauth_provider_slack, 88
oauth_provider_spotify, 88
oauth_provider_surfconext, 89
oauth_token_target, 90
oauth_token_target(), *29*, *105*
oauth_ui, 92
oauth_ui(), *44–46*, *48*, *52*, *149*
OAuthClient, *3*, *6*, *10*, *12*, *14*, *30*, *31*, *34*, *37*, *46*, *59*, *64*, *92*, 93, *108*, *113*, *116*, *120*, *127*, *129*, *131*, *132*, *134*, *137*, *140*, *141*, *144*
OAuthConnection, *34*, *41*, *51*, 106, *145*
OAuthProvider, *3*, *10*, *65*, *67*, *68*, *70–77*, *80–82*, *84*, *86–90*, 113
OAuthToken, *5*, *10–12*, *14*, *34*, *51*, *52*, *61*, *77*, *108*, *110*, *118*, 123, *126*, *128*, *134*, *135*, *137*, *140*

perform_client_bearer_req, 126
perform_resource_req, 128
perform_resource_req(), *112*, *113*, *123*, *127*, *136*, *138*
prepare_authorization_request, 131
prepare_authorization_request(), *29*, *104*, *133*, *143*
prepare_call, 132
prepare_call(), *11*, *12*, *52*, *131*, *132*
print.OAuthConnections *(*oauth_connections*)*, 37
print.OAuthConnectionStore *(*oauth_connection_store_memory*)*, 35
print.OAuthOwnerPolicy *(*oauth_browser_owner*)*, 16

refresh_token, 134
resource_req, 136
resource_req(), *6*
revoke_token, 139

shiny::reactiveValues(), *50*
shiny::shinyApp(), *46*, *92*
smart_client, 141
smart_client(), *29*, *34*, *37*, *39*, *104*, *131*, *145*, *147*, *148*
smart_context, 144
smart_context(), *112*, *125*
smart_discover, 146
smart_discover(), *141*, *142*
smart_fhir_user *(*smart_context*)*, 144
smart_fhir_user(), *112*
smart_launch_route, 147
smart_launch_route(), *41*, *43*
smart_patient *(*smart_context*)*, 144
smart_patient(), *112*

use_shinyOAuth, 149
use_shinyOAuth(), *43*, *92*, *93*