

Package ‘shinyOAuth’

September 18, 2026

Title OIDC Authentication and OAuth Authorization for 'shiny' Applications

Version 0.6.1

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 (>= 1.0), openssl (>= 2.0.0), httr2 (>= 1.1.0), curl, urltools (>= 1.7.3), cachem (>= 1.1.0), 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, mirai (>= 2.5.1), future, withr, later, callr, processx, pkgload, chromote, 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 8.0.0

NeedsCompilation no

Author Luka Koning [aut, cre, cph]

Maintainer Luka Koning <koningluka@gmail.com>

Repository CRAN

Date/Publication 2026-09-18 06:40:17 UTC

Contents

check_oauth21	3
client_bearer_req	4
custom_cache	6
get_userinfo	9
handle_callback	11
introspect_token	13
is_ok_host	15
OAuthClient	16
OAuthConnection	28
OAuthProvider	33
OAuthToken	43
oauth_browser_owner	46
oauth_client	48
oauth_client_mtls_registration	59
oauth_client_secret_apple	61
oauth_connection	62
oauth_connections	64
oauth_connections_server	67
oauth_connections_ui	69
oauth_connection_store_memory	72
oauth_form_post_ui	73
oauth_module_server	76
oauth_provider	85
oauth_provider_apple	95
oauth_provider_auth0	96
oauth_provider_github	97
oauth_provider_google	97
oauth_provider_keycloak	98
oauth_provider_microsoft	99
oauth_provider_oidc	102
oauth_provider_oidc_discover	105
oauth_provider_okta	109
oauth_provider_slack	110
oauth_provider_spotify	110
oauth_ui	111
perform_client_bearer_req	113
perform_resource_req	115
prepare_authorization_request	118
prepare_call	120
refresh_token	121

resource_req	124
revoke_token	127
smart_client	129
smart_context	132
smart_discover	133
smart_launch_route	135
use_shinyOAuth	137

Index	139
--------------	------------

check_oauth21	<i>Assess an OAuth configuration against a pinned OAuth 2.1 draft</i>
---------------	---

Description

This function inspects a configured [OAuthClient](#) and its [OAuthProvider](#) for compliance with the OAuth 2.1 draft 16 (ruleset 1.1.0) specification. It reports configuration gaps, unresolved external prerequisites, and recommendations without changing the configuration or making requests.

Usage

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

Arguments

config	An OAuthClient or OAuthProvider . Provider-only assessments are partial and cannot establish missing client settings.
context	Optional named list with operations, a character vector selecting additional "userinfo", "introspection" or "revocation" operations, and nonce_exception, a scalar logical declaration. Setting nonce_exception = TRUE declares that the authorization server has the assurance required by draft section 7.5.1.1 for this confidential deployment and specific request's correct OIDC nonce use. It does not supply observed evidence or excuse missing local prerequisites. Prefer S256 PKCE.
draft	Implemented target revision. Currently only "draft-ietf-oauth-v2-1-16" is supported (an Internet-Draft, not an RFC).

Details

Ruleset 1.1.0 covers code/refresh, enabled PAR, required UserInfo and introspection, and the additional operations selected in context. Signing and encryption key retrieval is included when applicable. Future resource URLs, arbitrary request customization, browser/proxy TLS, registered redirect matching, secret custody, and authorization/resource server behavior require separate evidence. Optional DPoP, mTLS, PAR, JAR and JARM are not required as a bundle.

configuration_compliant is FALSE if an applicable mandatory configuration check fails; otherwise NA if a mandatory configuration check is unresolved; otherwise TRUE when a nonempty set of applicable mandatory checks passes. Recommendations and external unknowns do not change that

verdict. In particular, the legacy assertion type JWT is a recommendation finding; the assertion audience is a separate mandatory check for JWT authentication. requirement_source distinguishes OAuth core, OIDC, extension specifications, security guidance and local package/application policy. Callback capacity thresholds are package recommendations, not draft-defined numeric minima; complete encoded requests still need deployment testing. OAuth 2.1 assessment is opt-in and does not change existing OAuth 2.0 configuration or requests.

A positive verdict applies only to the recorded scope and ruleset, with the current configuration, options, runtime and declared context. It is not certification or a test of a live deployment. Rerun after policy changes. Reports contain no client/provider objects, credentials, keys or endpoint URLs. No caches or state stores are read or changed; their method contracts are inspected without invoking them.

Value

A shinyOAuth21_assessment list with configuration_compliant, checks, draft, ruleset_version, package_version, assessed_at, assessment_scope, and operations. checks is a data frame with stable id, scope, status (pass, fail, unknown, not_applicable), requirement (MUST, SHOULD, info), message, remediation, reference, evidence_source, requirement_source, and logical_affects_verdict columns. Only rows with affects_verdict = TRUE enter aggregation; unknown external obligations remain visible separately.

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	<i>Alias for resource_req()</i>
-------------------	---------------------------------

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.
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. Any user-supplied Authorization or DPoP header is ignored to ensure the token authentication set by this function is not overridden.
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 <code>shinyOAuth.allow_redirect</code> is enabled. NULL inherits that global option (disabled by default). Set to TRUE only if you trust all possible redirect targets and understand the security implications.
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 <code>options(shinyOAuth.allowed_hosts)</code> 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. Set to FALSE only if you have already validated the URL and understand the security implications.

client	Optional OAuthClient . Required when the effective token type is DPoP, because the client carries the configured DPoP proof key, and also when using sender-constrained mTLS / certificate-bound tokens so shinyOAuth can attach the configured client certificate and validate any cnf thumbprint from an OAuthToken and observe any cnf thumbprint carried on a raw JWT access-token string.
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. Use exact hostnames for URLs derived from lower-trust input. It constrains the initial URL, not redirect destinations or resolved IPs; retain follow_redirect = FALSE. NULL adds no resource-specific policy.
oauth_client	Compatibility alias for client. Supply only one spelling.

Value

Same value as [resource_req\(\)](#).

custom_cache	<i>Create a custom state store or signing-key cache</i>
--------------	---

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. A shared signing-key cache lets workers reuse keys fetched from the provider rather than maintaining separate caches.

Usage

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

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. Deletes the entry for key. When <code>[["take"]]()</code> is provided, <code>[["remove"]]()</code> serves only as a best-effort cleanup and its return value is ignored. When <code>[["take"]]()</code> is not provided, shinyOAuth falls back to <code>[["get"]]() + [["remove"]]()</code> followed by a post-removal absence check via <code>[["get"]](key, missing = NA)</code> . In this fallback path the return value of <code>[["remove"]]()</code> is not relied upon; the post-check is authoritative.
take	A function(key, missing = NULL) -> value. Optional. An atomic get-and-delete operation. When provided, shinyOAuth uses <code>[["take"]]()</code> instead of separate <code>[["get"]]() + [["remove"]]()</code> calls to enforce single-use state consumption. This prevents TOCTOU (time-of-check / time-of-use) replay attacks in multi-worker deployments with shared state stores. Should return the stored value and atomically remove the entry, or return the missing argument (default NULL) if the key is not present. If your backend supports atomic get-and-delete natively (e.g., Redis GETDEL, SQL DELETE ... RETURNING), wire it through this parameter for replay-safe state stores. When take is not provided and the state store is not a per-process cache (like <code>cachem::cache_mem()</code>), shinyOAuth will error at state consumption time because non-atomic <code>[["get"]]() + [["remove"]]()</code> cannot guarantee single-use under concurrent access in shared stores.
info	Function() -> list(max_age = seconds, ...). Optional TTL information from <code>[["info"]]()</code> is used to align browser cookie max age in <code>oauth_module_server()</code> .
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; <code>cachem::cache_mem()</code> keeps its process-local serialized fallback. Callback bridges use it when available to avoid overwriting concurrent candidates; full partitions then reject callbacks until slots expire or are consumed. Other state-store writes still use set.

Details

This helper adapts your storage functions; it does not create a database, open connections, or make process-local storage shared. Your backend 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. Configure all workers with access to the same external store when routing does not keep the authorization request and callback on the same process.

For a shared `state_store`, implement `take`: it must read and delete a pending login as one indivisible operation, so two requests cannot use it. Redis `GETDEL` and SQL `DELETE ... RETURNING` are examples of backend operations that can do this. Use the same `state_key` and matching provider/client settings on every worker. Separate reads and deletes, including those in `cachem::cache_disk()`, cannot ensure single-use state under concurrent access. See the [deployment guidance](#).

Store values are small R lists; preserve them without interpreting fields. Pending login records in external stores are AES-GCM sealed with a distinct key derived from `state_key` and bound to the client, provider, and state key. The backend receives an opaque `sealed_state_record` string instead of the browser binding, PKCE verifier, and nonce. Existing unsealed external records are rejected; users with logins pending across an upgrade must restart login. The default process-local memory store keeps records within the R trust boundary. Encryption does not replace backend access controls, expiry, or atomic `take`: a backend able to restore consumed entries can still violate single-use state.

With `request_object_mode = "request_uri"`, hosted Request Objects are stored as separate records containing the signed or encrypted JWT and its expiry. These records do not use the pending-login `sealed_state_record` wrapper. A signed, unencrypted JWT has readable claims, including authorization request details. Configure Request Object JWE encryption when those claims need confidentiality, and apply backend access controls and expiry to these records.

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 `jwt_keys_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.

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	<i>Fetch a user's profile (UserInfo)</i>
--------------	--

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; leave NULL when calling directly.
oauth_client	Compatibility alias for client. Supply only one spelling.

Details

The provider must have a `userinfo_url`: the OpenID Connect (OIDC) UserInfo endpoint, or an OAuth provider's profile API. With OIDC, this function checks that userinfo belongs to the same user as a validated ID token when available. If provider policy requires that comparison, an absent validated ID token causes an error. Prefer passing the complete [OAuthToken](#), which carries the ID token needed for this check.

Ordinary JSON profiles and signed JWT UserInfo responses are supported. Signed responses are verified using the provider's signing keys and `userinfo_allowed_algs`; encrypted UserInfo is not supported. Set `userinfo_signed_jwt_required` on the provider to require a signed response, and `userinfo_jwt_required_time_claims` on the client to require time claims such as `exp`. Present time claims are checked even when not required.

For certificate-bound tokens (mTLS) and key-bound tokens (DPoP), the helper uses the client's certificate or signing key and checks the token binding. It handles one DPoP nonce challenge with a fresh-proof retry. See the [advanced security vignette](#) for configuration.

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.

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"]])

  # Requires a refresh token. Keep the returned replacement.
  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	<i>Handle OAuth 2.0 callback: verify state, swap code for token, verify token</i>
-----------------	---

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. This should be the same value that was originally sent in <code>prepare_call()</code> .
browser_token	Browser token present in the user's session. This is usually managed by <code>oauth_module_server()</code> .
shiny_session	Optional captured Shiny session details for audit events. Normally supplied by the module; leave NULL when calling directly.
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 <code>client@enforce_callback_issuer</code> 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 <code>authorization_server_mode = "multi_redirect_uri"</code> must use <code>oauth_module_server()</code> 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	<i>Introspect an OAuth 2.0 token</i>
------------------	--------------------------------------

Description

Ask the provider to check an access or refresh token. This is called token introspection and requires a configured `introspection_url`. Use it when you need the provider's current token status rather than only a locally recorded expiry time: a token may have been revoked before its expiry. 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

<code>client</code>	<code>OAuthClient</code> object
<code>token</code>	<code>OAuthToken</code> object to introspect
<code>token_kind</code>	Which token to introspect: "access" (default) or "refresh".
<code>async</code>	If TRUE, return a promise resolving to the result. Configure mirai daemons or a future plan first; mirai takes priority. Use a non-sequential future plan to move work outside the main R process. Default FALSE waits and returns the result directly.
<code>shiny_session</code>	Optional captured Shiny session details for audit events. Normally supplied by the module; leave NULL when calling directly.

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

Read `result[["active"]]`: TRUE means active, FALSE means inactive, and NA means the result is unknown. Use `isTRUE(result[["active"]])` if your code must require a definite confirmation.

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"]])

  # Requires a refresh token. Keep the returned replacement.
  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.

Call this when checking configured endpoint URLs or diagnosing a URL-policy rejection. The provider and API request helpers apply these checks internally; a direct call lets you inspect the 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

<code>url</code>	Single URL or vector of URLs (character; length 1 or more)
<code>allowed_non_https_hosts</code>	Character vector of hostnames that are allowed to use HTTP instead of HTTPS. Defaults to localhost equivalents. Supports globs
<code>allowed_hosts</code>	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"))

# Caution: a catch-all pattern disables host restrictions
# (only scheme rules remain). Avoid unless you truly intend it
is_ok_host("https://anywhere.example", allowed_hosts = c("*"))
```

 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 = character(0),
resource = character(0),
claims = NULL,
state_store = cachem::cache_mem(max_age = 300),
state_payload_max_age = 300,
state_entropy = 64,
state_key = random_urlsafes(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
)

```

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_supported`, that list should be used to select the algorithm.

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.

`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`.

Set this to `TRUE` for clients that should prefer discovered `mtls_endpoint_aliases` on authorization-server requests even when `token_auth_style` itself is not an mTLS auth style, and present the certificate on token and protected-resource requests. Certificate/key configuration alone does not enable this mode.

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. Leave NULL (default) for a normal callback with parameters in the URL; no response_mode parameter is then sent. Use "query" to request that format explicitly, or "form_post" when your provider needs an HTTP POST. POST callbacks require <code>oauth_form_post_ui()</code> . Signed responses (JWT Secured Authorization Response Mode, JARM) use "jwt", "query.jwt", or "form_post.jwt" and require <code>oauth_module_server()</code> . "jwt" uses the query transport for this authorization-code flow. "form_post.jwt" also needs <code>oauth_form_post_ui()</code> . <code>handle_callback()</code> 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 must have a configured issuer or you must supply an explicit override so the signed Request Object remains audience-bound to the intended authorization server.
request_object_encryption_alg	Optional JWE key-management algorithm override for encrypted Request Objects. Current outbound support is limited to RSA-OAEP. When set, you must

- also set `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). When set, you must also set `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. Leave NULL (the default) to omit 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 fails fast otherwise, independently of the access token's representation. Observed binding data must match the configured key; requiring its presence

	is a separate policy (<code>dpop_require_observed_cnf</code>). In <code>oauth_client()</code>, the default <code>NULL</code> resolves to <code>TRUE</code> when <code>dpop_private_key</code> is configured and to <code>FALSE</code> otherwise. Set <code>FALSE</code> explicitly only when you intentionally want to allow Bearer access tokens, such as deployments where DPoP is used only to bind refresh tokens.
<code>redirect_uri</code>	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.
<code>enforce_callback_issuer</code>	Logical or <code>NULL</code>. When <code>TRUE</code>, enforce that authorization responses handled through this client include an RFC 9207 <code>iss</code> parameter and reject callbacks unless it exactly matches <code>provider@issuer</code>. 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 <code>NULL</code> (the <code>oauth_client()</code> helper default), shinyOAuth auto-enables this check for providers that advertise <code>authorization_response_iss_parameter_supported = TRUE</code> and have a configured issuer, such as OIDC discovery providers that expose RFC 9207 support. Set <code>FALSE</code> to opt out explicitly.
<code>scopes</code>	Character vector of permissions to request. The provider defines the available names. For OIDC (<code>issuer</code> set and <code>infer_oidc_from_issuer = TRUE</code>), shinyOAuth adds "openid" automatically if absent. The resulting set is used in the request and subsequent scope checks.
<code>resource</code>	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 <code>resource</code> parameter on the authorization request, initial token exchange, and token refresh requests. Default is <code>character(0)</code>.
<code>claims</code>	Optional request for specific OIDC user information, beyond scopes. Default <code>NULL</code> sends no request. Supply a list with <code>userinfo</code> and/or <code>id_token</code> members, for example <code>list(userinfo = list(email = list(essential = TRUE)))</code>. Use <code>claims_validation = "strict"</code> if an unmet request must stop login. Lists are JSON-encoded with <code>auto_unbox = TRUE</code>. Use <code>NULL</code> for an unconstrained claim, value for one required value, or values for a set. Wrap a single-element values vector in <code>I()</code> to keep it a JSON array, for example <code>list(values = I("example-acr"))</code>. A pre-encoded JSON string is also accepted. Your provider must support the OIDC claims parameter.
<code>state_store</code>	Storage for pending logins. The default <code>cachem::cache_mem(max_age = 300)</code> is suitable for one R process. For multiple app processes, supply a shared <code>custom_cache()</code> with <code>atomic[["take"]]()</code> and use the same <code>state_key</code> on every process. Plain <code>cachem::cache_disk()</code> is unsafe for shared login state because its separate read and delete operations do not prevent simultaneous reuse. See <code>custom_cache()</code> for method and stored-value requirements.
<code>state_payload_max_age</code>	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. Most apps should keep the default.
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. For multiple R processes, supply the same key and shared state_store on every process. Accepts a character string or raw vector of at least 32 bytes. Generate it from cryptographically random bytes; do not use a memorable password. 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, <code>oauth_client()</code> 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).
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". Default is character(0), which means these claims are validated only when present. Set, for example, userinfo_jwt_required_time_claims = "exp" to require an expiry on signed UserInfo JWTs, or pass multiple values to require additional temporal claims. For security-sensitive deployments that accept signed UserInfo JWTs, prefer requiring at least "exp".
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). (Note that not all providers may return each of these fields in 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 call-back 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. Use this when high-assurance DPoP deployments must fail closed on opaque access tokens that provide no observable binding. 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. The default preserves strict local assurance. Set FALSE for server-enforced opaque bindings that the client cannot observe; this does not disable certificate presentation or mTLS

endpoint selection. Missing confirmation is then allowed, but any observed confirmation is still validated, including mismatches and conflicting claims. This flag does not independently enable mTLS.

`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. Configure only audiences trusted for this application's identity tokens, not arbitrary API audiences.

`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_type`

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. The explicit type is recommended; it does not replace audience validation. This setting does not change JAR, JARM, ID token or DPoP types, or the OAuth form parameter `client_assertion_type`.

`resource_bases` Optional named character vector of approved API base URLs for `oauth_connection()` and `oauth_connections()`. The default `character()` leaves the existing token/request APIs unchanged. 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 "POST". Select POST only after confirming provider support. It submits form fields instead of a long URL query. Use the module's <code>request_login()</code> or <code>prepare_authorization_request()</code> URL-only helpers reject POST. This does not select the callback response_mode or replace a provider's PAR or signed Request Object requirements.
scope_policy	Internal versioned scope policy. Leave the default for ordinary OAuth clients. SMART adapters install their own policy, including required permissions; these checks cannot be disabled by scope_validation. This parameter is not an argument to <code>oauth_client()</code> .
smart	Internal SMART configuration installed by <code>smart_client()</code> . Leave the empty default for ordinary clients. This is not an argument to <code>oauth_client()</code> .
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.

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.

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

Make API requests using a Shiny session's current OAuth credentials and the client/API configuration supplied by an `OAuthClient`. For example, a hospital connection selects that hospital's API address and reads the session's current token for each request. Create it inside `server()` with `oauth_connection()` or the `connection(id)` method of `oauth_connections_server()`. Use `[[ "request" ]]()` to call an approved API, `[[ "is_usable" ]]()` to check local availability and `[[ "summary" ]]()` for status without credentials.

Details

The existing reactive token already updates on refresh; this object combines that lookup with client selection, API-address restrictions and session checks. With `oauth_connection()`, the application supplies the matching module's token source; the manager resolves its own stored records. These are optional shinyOAuth conveniences, not SMART or FHIR protocol objects.

Each operation resolves the current credentials, so refresh and logout are reflected without replacing the reference. `oauth_module_server()` owns the lifecycle of ordinary references; `oauth_connections_server()` owns managed references and supplies `[[ "refresh" ]]()`. Every reference expires when its Shiny session closes. A manager can retain the underlying grant across redirects; a new session obtains a new reference after verifying the local owner.

Call `[[ "is_usable" ]]()`, `[[ "summary" ]]()` and `[[ "request" ]]()` in the owning session's reactive context. If the connection cannot be resolved, `[[ "is_usable" ]]()` returns FALSE; `[[ "summary" ]]()` and `[[ "request" ]]()` raise an error. The ID is read-only and cloning is disabled. Managed resource and status reads do not count as owner activity. Record user actions with the manager's `touch()` method in an input event handler; automatic reactive updates must not prolong an idle owner's session.

Active bindings

`id` Read-only opaque character string identifying this reference. A manager uses the stored grant's ID across sessions and refreshes; `oauth_connection()` generates an ID lasting only for that reference. The ID is never an access token and does not authorize access by itself.

Methods

Public methods:

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

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

Usage:

```
OAuthConnection$new(id, client, resolve, refresh = 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 a manager's coordinated refresh. Legacy session references leave this NULL.

Returns: A new `OAuthConnection` instance.

`OAuthConnection$is_usable()`: Check whether the current token is locally usable. This checks token presence, known unexpired lifetime and the client's required scopes. It does not refresh the token, contact the provider or guarantee remote authorization. Request-specific scopes are checked by `[["request"]]()`.

Usage:

```
OAuthConnection$is_usable()
```

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

`OAuthConnection$refresh()`: Refresh a connection created by `oauth_connections_server()`. The manager coordinates refresh and verifies ownership before updating credentials. References created with `oauth_connection()` use their existing module's refresh lifecycle and cannot invoke this method.

Usage:

```
OAuthConnection$refresh(scopes = 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.

Details: After explicit narrowing succeeds, subsequent refreshes (including automatic refreshes and refreshes in another retained Shiny session) request the accepted scope limit. Widening requires a new authorization. This is a local connection policy: OAuth refresh-token scope itself is not reduced by requesting a narrower access token. Ordinary OAuth connections explicitly request their retained granted scopes when known, including when no explicit narrowing was selected. SMART omits request scope while its permissions equal the original launch grant. Providers may reject requested scopes; there is no retry without them. OIDC clients that require UserInfo must retain `openid`; narrowing that removes it is rejected before exchange. Include any additional scopes needed by the provider's profile endpoint in the client's `required_scopes`.

Returns: TRUE after a successful commit, or a promise resolving to TRUE when the manager 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()
```

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:

- `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 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.

`OAuthConnection$identity()`: Read explicitly selected OIDC identity fields from the current usable connection. Requires `openid` and a cryptographically validated ID token. This method never returns raw tokens or fetches profile data.

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: Call inside the owning session's reactive context. The result contains sensitive identity data: select only what the application needs and keep it out of logs and generic status displays. `[["summary"]]()` and printing continue to omit identity. Ordinary OAuth connections without validated OIDC identity cannot use this accessor.

These are the last validated identity/profile snapshots; an OAuth refresh can retain earlier ID-token claims and does not establish fresh user authentication. This accessor does not log the user into your application or establish an account-retention owner. It does not count as owner activity.

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
)

```

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.

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\$smart_resource(): [Experimental]

Fetch the contextual Patient or validated `fhirUser` through the approved FHIR base, using current read permissions. Prefer `smart_patient()` and `smart_fhir_user()` in application code.

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. This does not resolve the current token.

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)
  connection <- oauth_connection(client, shiny::reactive(auth[["token"]]))
  output[["status"]] <- shiny::renderText(connection[["summary"]][["status"]])
  records <- shiny::reactive({
    shiny::req(connection[["is_usable"]])()
    response <- connection[["request"]]( "api", "records", required_scopes = "read")
    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 = FALSE,
  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
  )

```

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, clients should use request_object_mode = "request" or request_object_mode = "request_uri". This setting enforces local construction only; it does not configure the authorization server. Register require_signed_request_object = true (or the server's equivalent) and verify unsigned requests are rejected before relying on downgrade-resistant request integrity.
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(). When this is TRUE, make sure the provider has a matching public request URI or wildcard prefix registered for the client. shinyOAuth stores this metadata for caller awareness, but it cannot verify provider-side registration state automatically.

token_endpoint_auth_signing_alg_values_supported	Optional vector of JWS algorithms that the provider advertises for JWT-based client authentication (<code>client_secret_jwt</code> / <code>private_key_jwt</code>) at the token endpoint. This metadata is used for early validation of <code>OAuthClient@client_assertion_alg</code> 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 <code>OAuthClient@dpop_signing_alg</code> and inferred outbound DPoP signing defaults.
authorization_response_iss_parameter_supported	Logical. Whether the provider advertises RFC 9207 support for returning an <code>iss</code> parameter on the authorization response. When <code>TRUE</code> , the <code>oauth_client()</code> helper can auto-enable callback issuer enforcement when the caller leaves <code>enforce_callback_issuer</code> unset and the provider also has a configured issuer.
response_modes_supported	Optional character vector of OAuth/OIDC <code>response_mode</code> values advertised by the provider. Discovery-backed providers use the discovery metadata value, defaulting to <code>c("query", "fragment")</code> 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; clients still fail fast if they request one of those unsupported modes.
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. • <code>"url"</code> (default): require the issuer used for discovery to match the discovery metadata exactly, including any trailing slash. • <code>"host"</code>: compare only scheme + host. • <code>"none"</code>: do not validate discovery issuer consistency. In most cases, keep the default <code>"url"</code>. Use <code>"host"</code> only for providers that publish tenant-independent metadata with a templated issuer, such as some Microsoft aliases.
use_nonce	Whether to tie the ID token to this login using a random nonce. Keep enabled for OIDC. The nonce is sent in the request and checked in the returned ID token.
use_pkce	Whether to protect the code exchange using Proof Key for Code Exchange (PKCE). Leave enabled; public clients require it. It sends a <code>code_challenge</code> with the login request and a matching secret <code>code_verifier</code> during token exchange.
pkce_method	PKCE code challenge method (<code>"S256"</code> or <code>"plain"</code>). <code>"S256"</code> is recommended. Use <code>"plain"</code> only if you are working with a provider that does not support <code>"S256"</code> .
userinfo_required	Whether to fetch a user profile after token exchange. The result is stored in <code>token@userinfo</code> ; a failed required fetch stops login. In <code>oauth_provider()</code> , this defaults to <code>TRUE</code> when <code>userinfo_url</code> is supplied and <code>FALSE</code> 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`

Whether fetched userinfo requires a validated ID token for comparison. When both are available, their actual sub values are always compared. TRUE also stops login if the validated ID token is absent. Requires `userinfo_required` and either `id_token_validation` or `use_nonce`. `oauth_provider()` enables this by default when those requirements are met.

`userinfo_signed_jwt_required`

Whether to require the user profile to arrive as a signed JWT (application/jwt). Default FALSE; ordinary JSON userinfo is accepted. When TRUE, requires `userinfo_required` and issuer; the signature must validate with an asymmetric algorithm from `userinfo_allowed_algs`. Unsigned, HMAC-signed, and encrypted userinfo JWTs are not accepted by the normal configuration. Discovery does not enable this automatically: provider support does not mean your app's registration requests signed userinfo.

`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. Set an explicit FALSE only when intentionally opting out of 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 <code>oauth_client(endpoint_auth = ...)</code> 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 <code>token_endpoint</code> , <code>userinfo_endpoint</code> , <code>introspection_endpoint</code> , <code>revocation_endpoint</code> , <code>par_endpoint</code> , or <code>pushed_authorization_request_endpoint</code> , 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 <code>cnf</code> claim with an <code>x5t#S256</code> 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 (<code>client_secret_basic</code>) • "body": Form body (<code>client_secret_post</code>) • "public": Public-client form body (none in discovery metadata); sends <code>client_id</code> but never <code>client_secret</code>, 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 <code>client_secret</code> (RFC 7523) • "private_key_jwt": JWT client assertion signed with an asymmetric key (RFC 7523)
jwt_cache	Storage for the provider's public signing keys. Defaults to <code>cache::cache_mem(max_age = 3600)</code> , an in-memory cache lasting one hour. A <code>custom_cache()</code> 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 <code>no-store</code> are not retained, and <code>no-cache</code> 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.
jwt_pins	Optional character vector of RFC 7638 JWK thumbprints (<code>base64url</code>) to pin against. If non-empty, fetched JWKS must contain keys whose thumbprints match these values depending on <code>jwt_pin_mode</code> . 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; <code>jwt_pin_mode</code> controls whether the surrounding JWK Set may also contain unpinned keys.

`jwt_pin_mode` Pinning policy when `jwt_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)

`jwt_host_issuer_match`

When TRUE, enforce that the discovery `jwt_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 for safer config. 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). Set explicitly to FALSE to opt out. For providers that legitimately publish JWKS on a different host (for example Google), prefer setting `jwt_host_allow_only` to the exact hostname rather than disabling this check.

`jwt_host_allow_only`

Optional explicit hostname that the `jwt_uri` must match. When provided, `jwt_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 `jwt_host_issuer_match`.

`id_token_allowed_algs`

Optional vector of allowed JWT algorithms for ID tokens. Use to restrict acceptable alg values on a per-provider basis. Supported asymmetric algorithms include RS256, RS384, RS512, ES256, ES384, ES512, and Ed25519 or legacy EdDSA with Ed25519 OKP keys (including at_hash validation). Ed448 verification is unsupported and fails closed. Symmetric HMAC algorithms HS256, HS384, HS512 are also supported but require that you supply a `client_secret` and explicitly enable HMAC verification via the option `options(shinyOAuth.allow_hs = TRUE)`. Defaults to `c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512", "Ed25519", "EdDSA")`, which intentionally excludes HS*. Each RSA verification key is bound to one algorithm: its JWK alg, if supplied, or the sole RSA algorithm in this allowlist. When several RSA algorithms are allowed, an unlabelled key is bound to RS256 (and rejected if RS256 is excluded). To use unlabelled keys with RS384 or RS512, configure only that RSA algorithm. EC curves already select one supported algorithm; legacy EdDSA with an Ed25519 key uses the Ed25519 operation. Only include HS* if you are certain the `client_secret` is stored strictly server-side and is never shipped to, or derivable by, the browser or other untrusted environments.

`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) still fail fast rather than being misused. 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: helps 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.
jwt_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. Opt in only for a provider known to issue Bearer tokens while omitting token_type from its token responses, contrary to OAuth 2.0. 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; other token and binding checks remain enforced.
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 an interoperability escape hatch 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 still fail closed. 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.

```

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"
)

# Configure a OIDC provider via OIDC 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.

Read properties with `@`, for example `auth[["token"]>@userinfo`. Profile fields depend on the provider. Keep access and refresh tokens out of the UI and logs.

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 confirmation claim set returned alongside a sender-constrained access token or observed on another token surface. 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; introspection or another provider proof surface is stronger assurance.
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. For stronger proof, configure introspection_checks = "scope".
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.
<code>original_id_token</code>	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 <code>id_token</code> if omitted. Treat this property as credential material.
<code>extra_fields</code>	List of additional parameters from the latest successful token endpoint response. Excludes <code>access_token</code> , <code>token_type</code> , <code>refresh_token</code> , <code>id_token</code> , <code>expires_in</code> , <code>scope</code> , and <code>cnf</code> , which have dedicated token properties. Defaults to an empty list. Successful refresh replaces this list, including when the response contains no extra fields.
<code>initial_extra_fields</code>	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 <code>extra_fields</code> .
<code>smart_context</code>	Internal interpreted SMART context. Empty for ordinary tokens; populated only by SMART token processing. Use <code>smart_context()</code> on a connection to read it. Includes sensitive patient and identity references.
<code>original_granted_scopes</code>	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

The `id_token_claims` property is a read-only computed property that returns the decoded JWT payload of the ID token as a named list. This surfaces all standard and optional OIDC claims (e.g., `sub`, `iss`, `aud`, `acr`, `amr`, `auth_time`, `nonce`, `at_hash`, etc.) without requiring manual JWT decoding. Returns an empty list when no ID token is present or if the token cannot be decoded.

Note: `id_token_claims` always decodes the JWT payload regardless of whether the ID token's signature was verified. Check the `id_token_validated` property to determine whether the claims were cryptographically validated. 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. Both lists can contain sensitive data; keep them out of the UI and logs.

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"]]
```

oauth_browser_owner	<i>Configure ownership of retained OAuth connections</i>
---------------------	--

Description

Choose who may use saved OAuth connections: the browser that created them, or an account already authenticated by your application. Supply the policy to [oauth_connections\(\)](#) with the matching retention mode. These factories do not set cookies, authenticate users or enable retention on [oauth_module_server\(\)](#) calls by themselves.

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. Use the server manager's <code>touch()</code> from a user input event handler; connecting, explicit refresh and disconnect also count.
--------------	---

<code>absolute_timeout</code>	Maximum owner lifetime in seconds, independent of activity. Must be at least <code>idle_timeout</code> .
<code>same_site</code>	Owner-cookie policy, "Lax" for top-level authorization navigation or "Strict". Embedded cross-site ownership is not supported. With "Strict", oauth_connections_ui() serves an intermediate same-origin document after validating a callback, before checking the existing owner.
<code>allow_http_loopback</code>	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.
<code>max_entries</code>	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 <code>max_entries</code> .
<code>resolver</code>	Trusted application function accepting the current Shiny session. On every call it must validate the application's local login and return NULL if unauthenticated, otherwise a plain list containing <code>subject</code> , <code>session_id</code> , <code>generation</code> , <code>authenticated_at</code> and <code>expires_at</code> . The first three are non-empty strings; the timestamps are finite Unix seconds. <code>Subject</code> is the stable local account ID. <code>Session ID</code> and <code>generation</code> identify the current local authentication session. Do not derive these from unverified Shiny inputs, URL values, email addresses or the external provider's token response.
<code>reauth_after_seconds</code>	Maximum age of the verified local authentication, in seconds. Required for account retention; refresh cannot reset this age.
<code>x</code>	An <code>OAuthOwnerPolicy</code> to print.
<code>...</code>	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 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.

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. Size `max_entries` for that entire window, not only simultaneous Shiny sessions.

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. The application remains responsible for validating its local session, including signature, expiry, revocation and account changes. Supplying this configuration is not proof that an arbitrary user ID is authenticated.

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.

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 = character(0),
  resource = character(0),
  claims = NULL,
  state_store = cachem::cache_mem(max_age = 300),
  state_payload_max_age = 300,
  state_entropy = 64,
  state_key = random_ur-safe(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(),
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
)

```

Arguments

provider	The service configuration, created with a provider helper such as <code>oauth_provider_google()</code> or <code>oauth_provider_oidc_discover()</code> .
client_id	The identifier assigned when you register your app with the provider.
client_secret	The secret issued for your app, preferably read with <code>Sys.getenv()</code>. Omit it for registrations that do not use a secret. It is required for <code>token_auth_style = "header"</code>. 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 <code>iss</code> parameter and reject callbacks unless it exactly matches <code>provider@issuer</code>. 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 <code>oauth_client()</code> helper default), shinyOAuth auto-enables this check for providers that advertise <code>authorization_response_iss_parameter_supported = TRUE</code> 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 (<code>issuer</code> set and <code>infer_oidc_from_issuer = TRUE</code>), shinyOAuth adds "openid" automatically if absent. The resulting set is used in the request and subsequent scope checks.
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 autho-

	rization request, initial token exchange, and token refresh requests. Default is <code>character(0)</code>.
<code>claims</code>	Optional request for specific OIDC user information, beyond scopes. Default <code>NULL</code> sends no request. Supply a list with <code>userinfo</code> and/or <code>id_token</code> members, for example <code>list(userinfo = list(email = list(essential = TRUE)))</code>. Use <code>claims_validation = "strict"</code> if an unmet request must stop login. Lists are JSON-encoded with <code>auto_unbox = TRUE</code>. Use <code>NULL</code> for an unconstrained claim, value for one required value, or values for a set. Wrap a single-element values vector in <code>I()</code> to keep it a JSON array, for example <code>list(values = I("example-acr"))</code>. A pre-encoded JSON string is also accepted. Your provider must support the OIDC claims parameter.
<code>state_store</code>	Storage for pending logins. The default <code>cachem::cache_mem(max_age = 300)</code> is suitable for one R process. For multiple app processes, supply a shared <code>custom_cache()</code> with <code>atomic[["take"]]()</code> and use the same <code>state_key</code> on every process. Plain <code>cachem::cache_disk()</code> is unsafe for shared login state because its separate read and delete operations do not prevent simultaneous reuse. See <code>custom_cache()</code> for method and stored-value requirements.
<code>state_payload_max_age</code>	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.
<code>state_entropy</code>	Length in characters of the random state identifier, from 22 to 128. Default 64. Most apps should keep the default.
<code>state_key</code>	Secret used to encrypt and protect pending login details. A random key is generated when omitted. This is separate from <code>client_secret</code> and is also used for public clients. For multiple R processes, supply the same key and shared <code>state_store</code> on every process. Accepts a character string or raw vector of at least 32 bytes. Generate it from cryptographically random bytes; do not use a memorable password. State uses AES-GCM authenticated encryption.
<code>client_assertion_private_key</code>	Optional private key for <code>private_key_jwt</code> client authentication at the token endpoint. Can be an <code>openssl::key</code> or a PEM string containing a private key. Required when the provider's <code>token_auth_style = 'private_key_jwt'</code>. 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.
<code>client_assertion_private_key_kid</code>	Optional key identifier (kid) to include in the JWT header for <code>private_key_jwt</code> assertions and JAR Request Objects. Useful when the authorization server uses kid to select the correct verification key.
<code>client_assertion_alg</code>	Optional JWT signing algorithm to use for client assertions. When omitted, defaults to HS256 for <code>client_secret_jwt</code>. For <code>private_key_jwt</code>, 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_supported` 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.

`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`.

Set this to `TRUE` for clients that should prefer discovered `mtls_endpoint_aliases` on authorization-server requests even when `token_auth_style` itself is not an mTLS auth style, and present the certificate on token and protected-resource requests. Certificate/key configuration alone does not enable this mode.

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. Leave NULL (default) for a normal callback with parameters in the URL; no response_mode parameter is then sent. Use "query" to request that format explicitly, or "form_post" when your provider needs 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 must have a configured issuer

or you must supply an explicit override so the signed Request Object remains audience-bound to the intended authorization server.

`request_object_encryption_alg`

Optional JWE key-management algorithm override for encrypted Request Objects. Current outbound support is limited to RSA-OAEP. When set, you must also set `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). When set, you must also set `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. Leave NULL (the default) to omit 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 fails fast otherwise, 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 <code>oauth_client()</code>, the default NULL resolves to TRUE when dpop_private_key is configured and to FALSE otherwise. Set FALSE explicitly only when you intentionally want to allow Bearer access tokens, such as deployments where DPoP is used only to bind refresh tokens.
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, <code>oauth_client()</code> 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).
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". Default is character(0), which means these claims are validated only when present. Set, for example, userinfo_jwt_required_time_claims = "exp" to require an expiry on signed UserInfo JWTs, or pass multiple values to require additional temporal claims. For security-sensitive deployments that accept signed UserInfo JWTs, prefer requiring at least "exp".
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)`. (Note that not all providers may return each of these fields in 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. Use this when high-assurance DPoP deployments must fail closed on opaque access tokens that provide no observable binding. 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. The default preserves strict local assurance. Set FALSE for server-enforced opaque bindings that

the client cannot observe; this does not disable certificate presentation or mTLS endpoint selection. Missing confirmation is then allowed, but any observed confirmation is still validated, including mismatches and conflicting claims. This flag does not independently enable mTLS.

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. Configure only audiences trusted for this application's identity tokens, not arbitrary API audiences.

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_type

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. The explicit type is recommended; it does not replace audience validation. This setting does not change JAR, JARM, ID token or DPoP types, or the OAuth form parameter `client_assertion_type`.

authorization_method

Browser method for sending the authorization request: "GET" (default) or "POST". Select POST only after confirming provider support. It submits form fields instead of a long URL query. Use the module's `request_login()` or `prepare_authorization_request()` URL-only helpers reject POST. This does not select the callback `response_mode` or replace a provider's PAR or signed Request Object requirements.

resource_bases

Optional named character vector of approved API base URLs for `oauth_connection()` and `oauth_connections()`. The default `character()` leaves the existing token/request APIs unchanged. 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 com-

	parison 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.

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

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, ready to include in a registration request. Submit it through your 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

<code>client</code>	OAuthClient configured for RFC 8705 mutual TLS client authentication or for certificate-bound access tokens.
<code>tls_client_auth_type</code>	For <code>tls_client_auth</code> , which RFC 8705 certificate identifier field to emit. One of "subject_dn", "san_dns", "san_uri", "san_ip", or "san_email".
<code>tls_client_auth_value</code>	Optional explicit value for the selected <code>tls_client_auth_type</code> . 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.
<code>jwks_uri</code>	Optional absolute URL of a JWKS document to publish for <code>self_signed_tls_client_auth</code> . When omitted, the helper returns an inline jwks object with the configured client certificate chain in x5c.
<code>oauth_client</code>	Compatibility alias for <code>client</code> . 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()`. Unlike a fixed password, this secret expires; replace it before its expiry.

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

<code>client_id</code>	Apple Services ID or App ID used as the OAuth client id
<code>team_id</code>	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 lifetime in seconds. Must be 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

Make API requests with a Shiny session's current OAuth credentials

Description

Combine one module's reactive token with its client and approved API addresses configured on `oauth_client()`. Call `[["request"]]()` on the returned connection instead of assembling a token, client and URL for each request. 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.

Usage

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

Arguments

<code>client</code>	An OAuthClient with non-empty <code>resource_bases</code> , created by oauth_client() or smart_client() .
<code>token_reactive</code>	A Shiny reactive expression returning the current OAuthToken or NULL, usually <code>shiny::reactive(auth[["token"]])</code> . It must come from the module using <code>client</code> . Supplying that association is trusted server application wiring; a resource policy cannot prove an opaque token's audience.
<code>session</code>	The owning Shiny session; defaults to the current session.

Details

Create the reference once inside `server()`. Access it only in 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. The legacy module continues to own refresh and logout. Retained connection storage and its independent lifecycle are available through [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_connections

Configure several independently managed OAuth connections

Description

Create one manager outside `server()` for a named set of client/API configurations from `oauth_client()` or `smart_client()`. Use `oauth_connections_ui()` to handle callbacks and `oauth_connections_server()` for each Shiny session. Each successful authorization creates a separate connection, including repeated authorizations at the same client. Connections are independent local records. Providers may reuse an upstream grant, so revoking one can invalidate tokens held by other connections. Within a manager, refreshes sharing one credential are serialized. Rotation or an uncertain refresh invalidates other records holding the same credential; authorize those records again. Their identity and permissions are never replaced by another connection's response. Successful revocation also invalidates known copies of that credential. Distinct tokens may still share an upstream grant whose revocation effects the manager cannot predict. Retired credential digests survive connection replacement and expiry for at least the store's maximum age and the longest client state lifetime. A separate registry holds at most 10,000 digests or pending reservations. A full registry rejects refresh or skips remote revocation before sending credentials; it does not evict retirement evidence. Existing unrelated access remains usable.

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

<code>clients</code>	Non-empty named list of OAuthClient objects, at most 64. Each client must configure non-empty <code>resource_bases</code> . 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 does not request refresh tokens or extend provider authorization.
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. Keep these keys outside credential storage.
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

A manager is bound to one UI/server module ID and one public origin. Create another manager for another namespace. The memory store and owner registries survive Shiny sessions, not R restarts; copies in another R process fail closed.

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.

These are optional package interfaces, not SMART protocol objects. This manager supports generic OAuth clients and the SMART discovery, scope and launch policies configured by `smart_client()`.

Value

An OAuthConnections server-side configuration object. Printing shows only the retention mode and client count. It contains client configuration and deployment keys and must never be sent to the browser.

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

Call once inside `server()` with the manager and ID used by `oauth_connections_ui()`. The returned methods select stored connections by opaque IDs and recheck the local owner for every operation. Tokens are kept on the server and are not returned by summaries.

Usage

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

Arguments

<code>id</code>	Module ID shared with <code>oauth_connections_ui()</code> .
<code>manager</code>	Configuration from <code>oauth_connections()</code> .
<code>async</code>	Whether authorization and refresh use the existing async worker transport. Owner checks and store mutations stay in the original R process.
<code>refresh_proactively</code>	Refresh before expiry when a refresh credential is available. Otherwise the manager attempts refresh at expiry.
<code>refresh_lead_seconds</code>	Non-negative number of seconds before expiry used for proactive refresh. Background checks never extend owner inactivity limits.
<code>refresh_check_interval_ms</code>	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

References expire with this Shiny session even when their stored grants survive. A new session obtains new references after owner verification. The manager coordinates refresh across its connections. Refresh preserves the original authentication time and retention expiry. Reactive connection reads also recheck expiry at `refresh_check_interval_ms`, including references used without `connections()` or `errors()`. These checks notify dependent expressions when lifecycle state

changes; unchanged polling does not rerun application requests or extend owner inactivity limits. Notifications to application code reflect only this owner's record changes. Resource requests and status reads never reset owner inactivity, including when reactive expressions rerun after automatic refresh. Call `touch()` from a user input event handler to count an application action as activity. Do not call it from polling observers or ordinary reactive readers. 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. Use `revoke = FALSE` when preserving those authorizations is required. 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()` do not detect such remote changes: handle API authorization failures and obtain a new authorization.

Use this API inside 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.

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)`: an `OAuthConnection` for requests and refresh.
- `touch()`: record explicit user activity after checking the current owner. Call from an input event handler; returns TRUE invisibly.
- `disconnect(connection_id, revoke = TRUE)`: remove local usability first, then return separate local and remote revocation results.
- `disconnect_all(revoke = TRUE)`: cancel pending authorizations and disconnect this owner's stored connections; return a list of results.
- `logout(revoke = TRUE, reload = TRUE)`: invalidate the local owner/session generation first, 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`.

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	<i>Handle callbacks and establish the connection owner's browser session</i>
----------------------	--

Description

Wrap the application's UI with the same manager and module ID used by `oauth_connections_server()`. Browser retention establishes its HttpOnly owner cookie on an ordinary page request before Shiny starts. OAuth callbacks use the existing validated callback bridge and clean continuation.

Usage

```

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

```

Arguments

<code>base_ui</code>	A Shiny UI object or request-dependent UI function.
<code>id</code>	Module ID shared with <code>oauth_connections_server()</code> .
<code>manager</code>	Configuration from <code>oauth_connections()</code> .
<code>request_uri_resolver</code>	Optional trusted function mapping a Rook request to its public absolute URI. Required when a trusted reverse proxy changes the apparent scheme/host. Do not trust arbitrary forwarded headers. The resolved origin must match the manager's configured origin.
<code>app_base_path</code>	Public path at which the Shiny application is hosted, default <code>"/"</code> . Must begin and end with <code>/</code> . The wrapper inserts a document base before scripts so Shiny dependencies load from the app root even on nested callback pages. Do not supply a separate HTML base element. Managed JAR Request Object URLs are also published beneath this path.
<code>launch_routes</code>	List of <code>smart_launch_route()</code> configurations, empty by default. EHR entry requires browser retention and top-level navigation.
<code>additional_clients</code>	Optional named list of ordinary OAuth/OIDC clients used by separate <code>oauth_module_server()</code> modules. Names are their full module IDs. These clients keep their existing login lifecycle and are not managed connections. Use this single UI wrapper instead of nesting <code>oauth_ui()</code> . Their callback routes must be distinct from managed callbacks and SMART launch routes, on this app's origin and inside <code>app_base_path</code> . All clients must select an appropriate multi-server authorization mode, even when the manager contains one client.

Details

Raw GET and form POST callbacks never create or rotate an owner. A POST may lack a SameSite owner cookie. With Strict owner cookies, a validated callback first serves an inert same-origin document that navigates to the clean continuation, allowing the browser to send its existing cookie. The document does not establish an owner or exchange credentials. A managed continuation must carry a still-valid owner before credentials can be exchanged. A validated ordinary callback for `additional_clients` can establish a new empty manager owner independently. An invalid cookie on an ordinary page is cleared using an HTTP response and a same-origin redirect; the next request establishes a new empty browser owner.

The cookie is scoped to its host. Separate applications on that host must be trusted; different ports or paths do not isolate their cookies. The wrapper adds no owner cookie for session-only or account retention.

Value

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

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_connection_store_memory
```

Create a process-local store for retained OAuth connections

Description

Stores encrypted credential envelopes and coordinates their lifecycle across Shiny sessions in one R process. Create one store outside `server()`. This adapter does not survive process restarts and rejects use from a copied worker process. It is a connection-manager building block, not an authentication API.

Usage

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

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

Arguments

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

Details

Methods are trusted server-side operations. The manager must validate the current owner session and generation before calling them. The opaque owner identifier scopes records and can survive session rotation; knowledge of an owner or connection ID does not authenticate a user. Records contain ciphertext only. Encryption keys stay with the manager, outside this adapter. Never expose these methods or credential imports as HTTP routes.

- `[["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. Other owners are unaffected.

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_form_post_ui	<i>Wrap a Shiny UI to enable OAuth 2.0/OIDC form_post callbacks</i>
--------------------	---

Description

Accept the provider's login callback as an HTTP POST and continue login with [oauth_module_server\(\)](#). Use this when you select `response_mode = "form_post"` or `"form_post.jwt"`, either because the provider requires it or to keep callback parameters out of the browser URL. The POST arrives before a Shiny session exists; this wrapper receives it and makes the validated callback available to the server module. For query-string callbacks, use [oauth_ui\(\)](#).

Usage

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

Arguments

<code>base_ui</code>	Existing Shiny UI object, or a UI function accepting req.
<code>id</code>	Shiny module id used by <code>oauth_module_server()</code> . This must match the <code>id</code> argument passed to the server module.
<code>client</code>	<code>OAuthClient</code> object used by <code>oauth_module_server()</code> .
<code>callback_path</code>	Optional URL path to accept POST callbacks on. Defaults to the path component of <code>client@redirect_uri</code> and must match it when supplied. This is the public callback path; use <code>request_uri_resolver</code> to map a trusted proxy's backend path to the registered public URI.
<code>request_uri_resolver</code>	Optional function accepting the Rook req environment and returning the trusted, public absolute request URI without relying on query parameters. Return <code>NULL</code> 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 <code>callback_path</code> . Registered fixed query parameters must also occur unchanged in the incoming request. Continuation URLs preserve only registered application parameters.
<code>clients</code>	Optional client registry as in <code>oauth_ui()</code> . With a registry, omit <code>id</code> , <code>client</code> , and <code>callback_path</code> ; each client's redirect URI sets its route, and both query and form-post clients are supported.

Details

Set `response_mode` on your `oauth_client()`, wrap your UI here, and use the same `id` and `client` in `oauth_module_server()`. The wrapper includes the browser setup and privacy header supplied by `oauth_ui()`. For a callback path below the app root, pass `uiPattern = ".*"` to `shiny::shinyApp()` so Shiny routes the callback to this wrapper.

The wrapper checks the incoming address and login state, stores the callback temporarily, and redirects the browser to a normal Shiny page with a single-use handle. Raw callback values do not appear in that redirected URL. For `"form_post.jwt"`, it also validates the signed response using JWT Secured Authorization Response Mode (JARM). Every POST receives its own random handle. Handles expire after 120 seconds or the smaller of `state_payload_max_age` and the state store lifetime. The module still verifies the browser binding before consuming login state. Each client/provider/module namespace has a separate bounded pool in the state store: 256 partitions of eight slots, with each login assigned to one partition. When a partition is full, a new POST

replaces its oldest response. An expired or replaced handle fails validation; it can never select the replacement response. Completed logins invalidate their remaining candidates and remove locally known candidates where possible. Shared stores still require atomic take. With `set_if_absent`, slots are claimed atomically with a short TTL; a full partition rejects new callbacks until a slot is consumed or expires. Otherwise, concurrent writers can also replace a candidate; handle validation fails closed in that case. Pending form-post handles issued by an older version must be restarted after upgrading.

Value

A Shiny UI function. Pass it to `shiny::shinyApp()` and, for non-root callback paths, use `uiPattern = ".*"` so Shiny routes the callback path to this UI function.

Deployment behind a proxy

If a proxy receives HTTPS and forwards HTTP to Shiny, supply a `request_uri_resolver` that reconstructs the public address after verifying the request came from your trusted proxy. The default resolver does not trust forwarded headers. The resulting address must match 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	<i>OAuth 2.0 authorization and OIDC authentication module for Shiny</i>
---------------------	---

Description

Call `oauth_module_server()` inside your Shiny `server()` function to manage login for each user. It sends users to the provider, checks their return, and gives your app reactive login status and user information. Create client with `oauth_client()` outside `server()`, and wrap your complete UI with `oauth_ui()`.

This uses the OAuth 2.0 Authorization Code flow, with OpenID Connect (OIDC) identity checks when configured for an OIDC provider.

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

<code>id</code>	A name for this Shiny module, such as "auth".
<code>client</code>	The app configuration created with <code>oauth_client()</code> .
<code>auto_redirect</code>	If TRUE (default), start login automatically for unauthenticated sessions. If FALSE, call <code>auth[["request_login"]]()</code> to start it.
<code>async</code>	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. <code>future::sequential()</code> runs in the main process. See Asynchronous execution for operations that remain synchronous.
<code>indefinite_session</code>	If TRUE, the module will not automatically clear the token due to access-token expiry or the <code>reauth_after_seconds</code> window, and it will not trigger automatic reauthentication when a token expires or a refresh fails. This effectively makes sessions "indefinite" from the module's perspective once a user has logged in. Note that your API calls may still fail once the provider considers the token expired; this option only affects the module's automatic clearing and redirect behavior.
<code>reauth_after_seconds</code>	Optional maximum interactive-authentication age in seconds. If set, the module removes the token (and thus sets <code>authenticated</code> to FALSE) after this many seconds. Token refresh does not reset the timer. For OIDC providers, reauthentication requests send <code>max_age=0</code> ; the returned ID token must contain a valid <code>auth_time</code> , which is used as the next authentication start. OAuth-only providers have no standard way to require active user authentication, so for them this is a hard local session lifetime followed by an ordinary authorization request. By default this is NULL (no forced reauthentication).
<code>refresh_proactively</code>	If TRUE, obtain a replacement access token before expiry when a refresh token is available. Default FALSE. The module schedules refresh at approximately <code>expires_at - refresh_lead_seconds</code> .
<code>refresh_lead_seconds</code>	Number of seconds before expiry to attempt proactive refresh (default: 60)
<code>refresh_check_interval_ms</code>	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, automatically revokes provider tokens when the Shiny session ends (e.g., browser tab closed, session timeout). This is a best-effort operation. Revocation runs asynchronously only when the module is configured with `async = TRUE` (otherwise it runs synchronously). Requires the provider to have a `revocation_url` configured. Default is FALSE. Note that session-end revocation may not always succeed (e.g., network issues, provider unavailable), so combine with appropriate token lifetimes on the provider side.

`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. Keep "Strict" unless the deployment needs these broader cookie-sending rules.

`refresh_check_interval`

Compatibility alias for `refresh_check_interval_ms`. Supply only one spelling.

Details

Login starts automatically by default. Use `auto_redirect = FALSE` and `auth[["request_login"]]()` to start it from a button. Read `auth[["authenticated"]]` in reactive code, and use `req(auth[["authenticated"]])` before server operations that require login. Your app must also enforce its own access rules.

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.
- `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 an indefinite session keeps an expired token or one whose refresh failed. Resets after successful login, refresh, or logout.

The object also supplies:

- `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. Inside `observeEvent()`, register the promise handler and then return `invisible(NULL)` so Shiny can process the browser acknowledgment. Do not return the pending promise from the observer itself.
- `auth[["has_browser_token"]]()`: reports whether the browser token is available. Use it before building a custom login URL; it does not report whether the user is authenticated.

- `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. An existing token is left unchanged.
- `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`, configure `mirai::daemons()` or a non-sequential `future::plan()` before starting the app. Slow provider requests can then run outside the main R process. Without this, network waits can delay all Shiny sessions sharing that process.

Advanced operations sent to workers include PAR, signed Request Object preparation, and query JARM verification. State-store operations and Shiny Request Object publication stay in the main process. Discovery during app setup, standalone `prepare_call()`, and JARM verification in `oauth_form_post_ui()` remain synchronous. Use timeouts on those network and storage operations; see the [package options reference](#).

Browser setup

Open the app at its registered return address in a regular browser with cookies, session storage, and Web Crypto enabled. Embedded IDE viewers may prevent login. The binding token stays in origin- and tab-scoped session storage; the cookie contains an independent marker, which must match the stored record. The temporary browser cookie follows the state store's `max_age`, with a 300-second fallback when that lifetime is unavailable. The separate `state_payload_max_age` client setting limits the age of the login request. Each new login uses a fresh server-selected browser binding and its own marker cookie. Application callback routes and module namespaces identify the storage record. Separate tabs can complete logins independently; complete a login in the tab that started it. Starting another login in the same tab and module replaces that tab's pending binding. Pending logins must be restarted after upgrading from versions that used local storage. Private browser-binding inputs are excluded from Shiny bookmarks. Do not copy `auth[["browser_token"]]` into custom bookmark values, URLs, or logs. Treat the entire hostname as a trust boundary: cookies are shared across ports, even with `__Host-`, `Secure`, or `HttpOnly`. Use a dedicated hostname when other services are not trusted. The origin-scoped check prevents cookie adoption across ports, but co-hosted services can still disrupt cookies.

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)

```

```

    )
    http2::resp_check_status(resp)
    http2::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
  )

```

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, clients should use request_object_mode = "request" or request_object_mode = "request_uri". This setting enforces local construction only; it does not configure the authorization server. Register require_signed_request_object = true (or the server's equivalent) and verify unsigned requests are rejected before relying on downgrade-resistant request integrity.

`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()`. When this is TRUE, make sure the provider has a matching public request URI or wildcard prefix registered for the client. shinyOAuth stores this metadata for caller awareness, but it cannot verify provider-side registration state automatically.

`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; clients still fail fast if they request one of those unsupported modes.

`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. In most cases, keep the default "url". Use "host" only for providers that publish tenant-independent metadata with a templated issuer, such as some Microsoft aliases.
use_nonce	Whether to tie the ID token to this login using a random nonce. Keep enabled for OIDC. The nonce is sent in the request and checked in the returned ID token.
use_pkce	Whether to protect the code exchange using Proof Key for Code Exchange (PKCE). Leave enabled; public clients require it. 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" or "plain"). "S256" is recommended. Use "plain" only if you are working with a provider that does not support "S256".
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 <code>oauth_provider()</code> , this defaults to TRUE when userinfo_url is supplied and FALSE otherwise.
userinfo_id_token_match	Whether fetched userinfo requires a validated ID token for comparison. When both are available, their actual sub values are always compared. TRUE also stops login if the validated ID token is absent. Requires userinfo_required and either id_token_validation or use_nonce. <code>oauth_provider()</code> enables this by default when those requirements are met.
userinfo_signed_jwt_required	Whether to require the user profile to arrive as a signed JWT (application/jwt). Default FALSE; ordinary JSON userinfo is accepted. When TRUE, requires userinfo_required and issuer; the signature must validate with an asymmetric algorithm from userinfo_allowed_algs. Unsigned, HMAC-signed, and encrypted userinfo JWTs are not accepted by the normal configuration. Discovery does not enable this automatically; provider support does not mean your app's registration requests signed userinfo.

`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`. Set an explicit FALSE only when intentionally opting out of 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 <code>cachemem(max_age = 3600)</code> , an in-memory cache lasting one hour. A <code>custom_cache()</code> 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.
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 <code>jwks_pin_mode</code> . 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; <code>jwks_pin_mode</code> controls whether the surrounding JWK Set may also contain unpinned keys.
jwks_pin_mode	Pinning policy when <code>jwks_pins</code> 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 <code>jwks_uri</code> host matches the issuer host exactly. Defaults to FALSE at the class level, but helper constructors for OIDC (e.g., <code>oauth_provider_oidc()</code> and <code>oauth_provider_oidc_discover()</code>) enable this by default for safer config. The generic helper <code>oauth_provider()</code> will also automatically set this to TRUE when an issuer is provided and either <code>id_token_validation</code> or <code>id_token_required</code> is TRUE (OIDC-like configuration). Set explicitly to FALSE to opt out. For providers that legitimately publish JWKS on a different host (for example Google), prefer setting <code>jwks_host_allow_only</code> to the exact hostname rather than disabling this check.
jwks_host_allow_only	Optional explicit hostname that the <code>jwks_uri</code> must match. When provided, <code>jwks_uri</code> 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 <code>jwks_host_issuer_match</code> .
id_token_allowed_algs	Optional vector of allowed JWT algorithms for ID tokens. Use to restrict acceptable alg values on a per-provider basis. Supported asymmetric algorithms include RS256, RS384, RS512, ES256, ES384, ES512, and Ed25519 or legacy EdDSA with Ed25519 OKP keys (including at_hash validation). Ed448 verification is unsupported and fails closed. Symmetric HMAC algorithms HS256, HS384, HS512 are also supported but require that you supply a <code>client_secret</code> and explicitly enable HMAC verification via the option <code>options(shinyOAuth.allow_hs = TRUE)</code> . Defaults to <code>c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512", "Ed25519", "EdDSA")</code> , which intentionally excludes HS*. Each RSA verification key is bound to one

algorithm: its JWK alg, if supplied, or the sole RSA algorithm in this allowlist. When several RSA algorithms are allowed, an unlabelled key is bound to RS256 (and rejected if RS256 is excluded). To use unlabelled keys with RS384 or RS512, configure only that RSA algorithm. EC curves already select one supported algorithm; legacy EdDSA with an Ed25519 key uses the Ed25519 operation. Only include HS* if you are certain the `client_secret` is stored strictly server-side and is never shipped to, or derivable by, the browser or other untrusted environments.

`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) still fail fast rather than being misused. 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.

`jwtks_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 an interoperability escape hatch 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 still fail closed. 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. Opt in only for a provider known to issue Bearer tokens while omitting token_type from its token responses, contrary to OAuth 2.0. 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; other token and binding checks remain enforced.

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"
)

# Configure a OIDC provider via OIDC 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.

Read identity information 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`; do not rely on this helper to retrieve that payload. 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 = "ABCDEFGHJI",
    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_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.

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_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](#). Configure the client ID & secret in your [OAuthClient](#).

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. Issuer comparison remains exact, as for other OIDC providers.

Value

[OAuthProvider](#) object for use with a Google OAuth 2.0 app

Examples

```
oauth_provider_google()
```

```
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

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

`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, which works for many common setups. Use "public" if you need to suppress client_secret even when it is set in the environment. If you pass NULL, discovery will infer 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 fail closed on duplicate top-level iss members instead of applying this interoperability workaround.

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
)
```

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

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.

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 and fetches userinfo from Microsoft Graph.

Setting `id_token_validation = FALSE` disables ID token and nonce checks and leaves OAuth plus profile retrieval. Keep the default for OIDC sign-in. Tenant domains and other unrecognized tenant identifiers require this explicit opt-out; otherwise use the directory GUID to retain OIDC validation.

Value

[OAuthProvider](#) object configured for Microsoft identity platform

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)
})
}

# Need to open app in 'localhost:8100' to match with redirect_uri
# of the public Azure CLI app (above). Browser must use 'localhost'
# too to properly set the browser cookie. But Shiny only redirects to
# '127.0.0.1' & blocks process once it runs. So we disable browser
# launch by Shiny & then use 'later::later()' to open the browser
# ourselves a short moment after the app starts
later::later(
  function() {
    utils::browseURL("http://localhost:8100")
  },
  delay = 0.25
)

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

```

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"
)
```

Arguments

<code>name</code>	Friendly name for the provider
<code>base_url</code>	Base URL for OIDC endpoints
<code>auth_path</code>	Authorization endpoint path (default: "/authorize")
<code>token_path</code>	Token endpoint path (default: "/token")
<code>userinfo_path</code>	User info endpoint path (default: "/userinfo")
<code>introspection_path</code>	Token introspection endpoint path (default: "/introspect")
<code>use_nonce</code>	Logical, whether to use OIDC nonce. Defaults to TRUE
<code>id_token_validation</code>	Logical, whether to validate ID tokens automatically for this provider. Defaults to TRUE
<code>jwks_host_issuer_match</code>	When TRUE (default), enforce that the JWKS host discovered from the provider matches the issuer host exactly. For providers that serve JWKS from a different host (e.g., Google), set <code>jwks_host_allow_only</code> to the exact hostname instead of disabling this. Disabling (FALSE) is not recommended unless you also pin JWKS via <code>jwks_host_allow_only</code> or <code>jwks_pins</code>
<code>allowed_token_types</code>	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".

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"
)

# Configure a OIDC provider via OIDC 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 run-time 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'
jwt_host_issuer_match	When TRUE (default), enforce that the JWKS host discovered from the provider matches the issuer host exactly. For providers that serve JWKS from a different host, set jwt_host_allow_only to the exact hostname instead of disabling this. Disabling (FALSE) is not recommended unless you also pin JWKS via jwt_host_allow_only or jwt_pins.
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 (recommended). For a full discovery URL in-

put, require the discovered issuer's standard metadata location to match that URL instead.

- "host": compare only scheme + host (explicit opt-out; not recommended).
- "none": do not validate issuer consistency.

Prefer "url" and tighten hosts via `options(shinyOAuth.allowed_hosts)` when feasible.

... Additional fields passed to `oauth_provider()` (for example, `pkce_method = "plain"` when a provider explicitly advertises only plain PKCE support and you intentionally want to allow that downgrade).

`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 what a service supports. Your app's registration may require a particular `token_auth_style`, `secret`, or `key`; configure those to match the registration. Automatic selection prefers "header", then "body", then "public" when none is advertised and PKCE is enabled. For a public client registration, set `token_auth_style = "public"` explicitly. JWT and mTLS methods must also be selected explicitly.

Value

`OAuthProvider` object configured from discovery

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_1 = 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 `jwt_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 `jwt_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"
)

# Configure a OIDC provider via OIDC 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_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

<code>name</code>	Optional provider name (default "slack")
<code>profile</code>	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. Select the public profile only after enabling PKCE for that Slack app. Slack marks the app public, and 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; enable only during migration.

Details

Spotify requires scopes to be included in the authorization request. Set requested scopes on the client with `oauth_client(..., scopes = ...)`. Identity uses Spotify's immutable `account_id`. Existing installations must migrate stored account mappings and audit digests from `id` before upgrading. Link the old and new identifiers only from a successfully authenticated profile; do not use display names or email to merge accounts. To temporarily preserve old mappings, explicitly replace `provider@userinfo_id_selector` with `function(userinfo) userinfo[["id"]]` while completing the migration.

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_ui

Set up a Shiny UI for shinyOAuth

Description

Wrap your complete UI in `oauth_ui()` when using [oauth_module_server\(\)](#). It adds the browser code needed for login and protects callback responses from caching and referrer disclosure. Supply `id` and `client` to accept query callbacks before rendering any application UI or scripts.

Usage

```

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

```

Arguments

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

Details

Build the page as usual, for example with `fluidPage()`, then use `ui <- oauth_ui(ui, id = "auth", client = client)`, using the same module ID and client as the server. Pass the result to `shiny::shinyApp()`. UI functions are supported too, including functions accepting the Shiny request. This wrapper includes [use_shinyOAuth\(\)](#) setup. With `client`, it also serves client-hosted Request Objects at the app root using independent, single-use handles. Shared-worker apps need a shared `client@state_store` with `atomic take()`; a memory store supports one process.

For `response_mode = "form_post"` or `"form_post.jwt"`, use [oauth_form_post_ui\(\)](#) instead; it includes this setup and accepts POST callbacks.

GET callbacks are validated and sealed into short-lived, single-use bridge handles in the client's state store, then redirected to a clean URL before application HTML is rendered. Logical state is consumed only after the Shiny module verifies browser binding. The storage requirements and quotas are the same as [oauth_form_post_ui\(\)](#). Register any fixed application query parameters in `client@redirect_uri`; other inbound parameters are discarded. For non-root callback paths use `uiPattern = ".*"` in `shiny::shinyApp()`.

For multiple providers, supply `clients = list(auth_a = client_a, auth_b = client_b)` instead of `id` and `client`. Names are the server module IDs. The registry accepts query and form-post callbacks on configured routes. Each client 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 complete callback is then verified before a bridge handle is stored. Encrypted JARM on a shared route requires an outer iss identifying one distinct configured issuer. The decrypted, signed response must match it. Do not nest wrappers to route multiple providers.

Without `id` and `client`, ordinary pages still render, but raw OAuth GET callbacks fail closed with a setup error. Earlier `oauth_ui(ui)` query-flow applications must add those arguments. When integrating `use_shinyOAuth()` directly, provide an equivalent dedicated callback endpoint: third-party or application scripts must not execute on an unsanitized callback page.

HTML responses include `Cache-Control: no-store`, `Pragma: no-cache`, and `Referrer-Policy: no-referrer`. The browser reads these headers before loading page resources. The meta tag from `use_shinyOAuth()` takes effect only once the browser reads that tag, so it may miss early resource requests. You can also set the same HTTP header at your web server.

Value

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

See Also

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

Examples

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

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

`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	Either the absolute URL to call or an httr2::request() object to authorize and perform. When you pass a request object, shinyOAuth uses it as the base request, still applies token authentication and request defaults, and then layers any explicit method, headers, query, and follow_redirect overrides on top. Inherited httr2 authentication, caching, and retry policies, and curl authentication or method-changing options are rejected. Use httr2::req_method() and httr2 body helpers to configure the request. HEAD requests with bodies are rejected because httr2 can transmit them as POST despite the explicit method. Authenticated response caching is unsupported. shinyOAuth owns retries; configure them with idempotent and the shinyOAuth.retry_* options.
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. Any user-supplied Authorization or DPoP header is ignored to ensure the token authentication set by this function is not overridden.
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). Set to TRUE only if you trust all possible redirect targets and understand the security implications.
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. Set to FALSE only if you have already validated the URL and understand the security implications.
client	Optional OAuthClient . Required when the effective token type is DPoP, because the client carries the configured DPoP proof key, and also when using sender-

	constrained mTLS / certificate-bound tokens so shinyOAuth can attach the configured client certificate and validate any cnf thumbprint from an OAuthToken and observe any cnf thumbprint carried on a raw JWT access-token string.
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. Use exact hostnames for URLs derived from lower-trust input. It constrains the initial URL, not redirect destinations or resolved IPs; retain follow_redirect = FALSE. 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	Either the absolute URL to call or an httr2::request() object to authorize and perform. When you pass a request object, shinyOAuth uses it as the base request, still applies token authentication and request defaults, and then layers any explicit method, headers, query, and follow_redirect overrides on top. Inherited httr2 authentication, caching, and retry policies, and curl authentication or method-changing options are rejected. Use httr2::req_method() and httr2 body helpers to configure the request. HEAD requests with bodies are rejected because httr2 can transmit them as POST despite the explicit method. Authenticated response caching is unsupported. shinyOAuth owns retries; configure them with idempotent and the shinyOAuth.retry_* options.
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. Any user-supplied Authorization or DPoP header is ignored to ensure the token authentication set by this function is not overridden.
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). Set to TRUE only if you trust all possible redirect targets and understand the security implications.
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. Set to FALSE only if you have already validated the URL and understand the security implications.
client	Optional OAuthClient . Required when the effective token type is DPoP, because the client carries the configured DPoP proof key, and also when using sender-constrained mTLS / certificate-bound tokens so shinyOAuth can attach the configured client certificate and validate any cnf thumbprint from an OAuthToken and observe any cnf thumbprint carried on a raw JWT access-token string.

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. Use exact hostnames for URLs derived from lower-trust input. It constrains the initial URL, not redirect destinations or resolved IPs; retain follow_redirect = FALSE. NULL adds no resource-specific policy.
oauth_client	Compatibility alias for client. Supply only one spelling.

Details

Only send a token to an API you intend to authorize. 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, also supply client so the request uses the matching key or certificate.

Value

An [httr2](#) response object.

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"]]

  # Recommended for most callers: build + perform in one step.
  response <- perform_resource_req(
    token,
    "https://api.example.com/resource",
    query = list(limit = 5)
  )

  # Build only when you need to inspect the request yourself.
  request <- resource_req(
    token,
    "https://api.example.com/resource",
```

```

    query = list(limit = 5)
  )

  # Inspect request settings without printing authentication headers.
  # httr2::req_perform(request) sends it when ready.

  # Or start from your own httr2 request and still let shinyOAuth perform it
  # so 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)

  # Constrain dynamic URLs before attaching a token. check_url alone does not
  # restrict HTTPS hosts unless a global allowed_hosts policy is configured.
  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, navigate the browser to `url`. For POST, create a form with `url` as its action, method POST, and `application/x-www-form-urlencoded` encoding. Add one hidden input per fields entry, assigning its name and value through DOM properties or an HTML escaping library, then submit the form in the current browser window. Repeated names (such as OAuth resource) must remain repeated inputs. Do not send the authorization request from R: the provider needs to interact with the user's browser and login cookies.

The client selects `authorization_method = "POST"` explicitly. Ordinary OAuth providers may not support POST; confirm their documentation first. `smart_client()` additionally requires the `authorize-post` capability. The outgoing method is independent of the callback `response_mode`. Configured PAR and Request Object requirements still apply. Each result belongs to one login attempt; do not cache it or reuse it after logout.

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. Existing state and callback size limits also apply. 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: do not log it or expose it to other browser sessions.

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

<code>client</code>	An OAuthClient object.
<code>browser_token</code>	Browser-bound token used to tie the login attempt to the current browser session.
<code>request_uri_publisher</code>	Optional function used when <code>request_object_mode = "request_uri"</code> . It must accept <code>request_object</code> , <code>request_handle_id</code> , <code>expires_at</code> , and <code>oauth_client</code> arguments and return an absolute HTTPS request-object URL that the provider can fetch.
<code>oauth_client</code>	Compatibility alias for <code>client</code> . 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

Use a refresh token to obtain a new access token without sending the user through login again. Call this when your application manages token lifetime itself, for example before continuing API requests with an expiring token. Assign the returned [OAuthToken](#) to keep the updated credentials. [oauth_module_server\(\)](#) can manage refresh during a Shiny session with `refresh_proactively = TRUE` when a refresh token is available.

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. Configure mirai daemons or a future plan first; mirai takes priority. Use a non-sequential future plan to move 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; leave NULL when calling directly.
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 does not establish a new interactive login. Use the module's reauth_after_seconds argument when a fresh login is required. 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; await the existing promise instead. Separate R processes require coordination by the

application. The input token is a value object: store the returned token for subsequent refreshes, and use an application generation check when assigning results after logout or a new login. Completed results are not cached.

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. After any other outcome (including an unavailable worker result), discard that credential and require a new login. This does not accept unvalidated access or identity data. The module applies this rule even with indefinite sessions.

Value

An updated [OAuthToken](#) object with refreshed credentials.

What changes:

- `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`

Validation failures cause errors: If the provider returns a new ID token that fails validation (wrong issuer, audience, expired, or subject mismatch with original), or if `userinfo` subject doesn't match the new ID token, the refresh fails with an error. 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"]])

  # Requires a refresh token. Keep the returned replacement.
  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 [http2](#) 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.
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. Any user-supplied Authorization or DPoP header is ignored to ensure the token authentication set by this function is not overridden.
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). Set to TRUE only if you trust all possible redirect targets and understand the security implications.
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. Set to FALSE only if you have already validated the URL and understand the security implications.
client	Optional OAuthClient . Required when the effective token type is DPoP, because the client carries the configured DPoP proof key, and also when using sender-constrained mTLS / certificate-bound tokens so shinyOAuth can attach the configured client certificate and validate any cnf thumbprint from an OAuthToken and observe any cnf thumbprint carried on a raw JWT access-token string.
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. Use exact hostnames for URLs derived from lower-trust input. It constrains the initial URL, not redirect destinations or resolved IPs; retain follow_redirect = FALSE. NULL adds no resource-specific policy.
oauth_client	Compatibility alias for client. Supply only one spelling.

Details

Only send a token to an API you intend to authorize. 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, also supply `client` so the request uses the matching key or certificate.

Managed Authorization credentials cannot be combined with an `access_token` query parameter or form field. Inspection covers URL/query inputs and prebuilt `httr2` form, raw and string bodies labelled `application/x-www-form-urlencoded`. JSON business fields are unaffected. File, multipart and streaming bodies are not parsed; callers must ensure these contain no additional OAuth credential transport. Later request changes outside these helpers require a new check.

Value

An `httr2` request object, ready to be performed with `httr2::req_perform()`. Callers may still add headers or query parameters, but when the effective token type is DPoP they must not change the request method or base URL after calling `resource_req()` because the proof is already bound to those values.

DPoP note

DPoP proofs bind the current HTTP method and target URI (without query or fragment). 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.

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"]]

  # Recommended for most callers: build + perform in one step.
  response <- perform_resource_req(
    token,
    "https://api.example.com/resource",
    query = list(limit = 5)
  )

  # Build only when you need to inspect the request yourself.
  request <- resource_req(
    token,
    "https://api.example.com/resource",
    query = list(limit = 5)
  )

  # Inspect request settings without printing authentication headers.
  # httr2::req_perform(request) sends it when ready.

  # Or start from your own httr2 request and still let shinyOAuth perform it
  # so 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)

# Constrain dynamic URLs before attaching a token. check_url alone does not
# restrict HTTPS hosts unless a global allowed_hosts policy is configured.
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

<code>client</code>	OAuthClient object
<code>token</code>	OAuthToken object containing tokens to revoke
<code>token_kind</code>	Which token to revoke: "refresh" (default) or "access"
<code>async</code>	If TRUE, return a promise resolving to the result. Configure mirai daemons or a future plan first; mirai takes priority. Use a non-sequential future plan to move work outside the main R process. Default FALSE waits and returns the result directly.
<code>shiny_session</code>	Optional captured Shiny session details for audit events. Normally supplied by the module; leave NULL when calling directly.
<code>oauth_client</code>	Compatibility alias for <code>client</code> . 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. Check the returned status: an absent endpoint or token, or an unsuccessful HTTP response, leaves 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"]])

  # Requires a refresh token. Keep the returned replacement.
  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]**

Combine a reviewed [smart_discover\(\)](#) snapshot with an existing app registration. The result is an [OAuthClient](#) for [oauth_connections\(\)](#), with the resource ID "fhir". Discovery does not register the app or grant access.

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 fail closed, 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 fail closed.
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/_history". 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. This does not apply to refresh responses or change ordinary OAuth defaults.
label	Display label, default "FHIR server"; no credentials or context.
state_store, state_key, state_payload_max_age	See oauth_client() .

Details

This constructor selects SMART 2.2 scope rules, S256 PKCE and the exact FHIR base as the authorization request's aud. Identity is opt-in: "openid" requests and requires openid, signed ID-token validation and nonce binding. "fhirUser" additionally requests and requires the fhirUser scope and claim. "none" does not enable OIDC because an issuer happens to be present. A patient in context is independent of the authenticated user and local owner.

Only direct authorization requests and query/form POST callbacks are supported. Claims requests (including auth_time), JAR, PAR, JARM, DPoP, mTLS and remote freshness requirements are not supported by this constructor. EHR clients require a fresh registered launch transaction. Configure standalone and EHR registrations as separate clients when both are needed. No launch handle is stored in shared provider configuration. Local usability policy requires a positive lifetime. An initial response may omit expires_in only when initial_expires_in_fallback is configured explicitly; refresh responses must include it. The generic assumed lifetime is not used. SMART back-channel and resource requests require TLS 1.2 or newer. A stronger configured TLS minimum is preserved; ordinary clients keep their defaults.

Value

An [OAuthClient](#), usable by the existing module or the separate connection manager. @resource_bases contains the approved fhir base; @required_scopes and @label use the ordinary client properties. @smart describes the selected profile. The client contains registration settings, never a user's token or launch context. Configure it outside server().

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]

Return interpreted patient/encounter and validated user references for the current accepted token. This is sensitive data; keep it out of general status tables and logs. 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 <code>smart_client()</code> , 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. Neither changes the local connection owner. General summaries omit all of this data. The application remains responsible for displaying patient identity clearly and discarding data from an older context revision. Experimental `fhirContext` and styling extensions remain raw data and are not automatically fetched or interpreted. 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; key patient-dependent application data by connection ID and this revision. 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. This returns server metadata for application setup; it does not register an app, choose client credentials, or start authorization.

Usage

```
smart_discover(fhir_base, endpoint_hosts = NULL, allow_http_loopback = FALSE)
```

Arguments

<code>fhir_base</code>	Trusted FHIR base URL, including its complete path, for example "https://ehr.example/fhir/R4".
<code>endpoint_hosts</code>	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.
<code>allow_http_loopback</code>	Logical, default FALSE. Explicit development exception permitting HTTP only at localhost, 127.0.0.1, or ::1. This applies to the base and metadata URLs and does not change global options. It does not establish production TLS interoperability.

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, so discovery does not change protocol identifiers.

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 `jwt_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`. No client, token, or live cache is stored in the result. 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

Supply a trusted, deployment-configured base, never an arbitrary browser query parameter. 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`, `jwt_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.

The request carries no OAuth credentials. Redirects are refused even when the generic redirect option is enabled. Existing package TLS, timeout, response-size, and retry protections apply. 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. No automatic metadata cache is used: every call reads a new snapshot. Applications should review metadata changes before replacing configuration; do not rediscover endpoints during a pending authorization or to reinterpret a retained grant.

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	<i>Register a SMART EHR launch entry route</i>
--------------------	--

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 <code>"/smart/launch"</code> . Register <code>paste0(manager[["app_origin"]], path)</code> with the EHR. It must be inside the UI's <code>app_base_path</code> 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 <code>client_id</code> values. Each must select an EHR-mode <code>smart_client()</code> . 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 <code>state_payload_max_age</code> , 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`.

Apply ingress rate limits to both launch entry and ordinary pages that create browser owners. The owner quota isolates an existing browser's pending work; it is not a per-person or per-network-client rate limit, since unauthenticated callers can obtain additional browser owners. Configure these controls at a trusted reverse proxy using its verified client address, not arbitrary inbound forwarded headers. Size the owner registry for the admitted traffic window.

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. The ticket alone cannot authorize a connection. No inline script permission or additional CSP nonce is needed for the handoff. Application access logs must also avoid recording raw launch query strings. Normal callback issuer checks and single-use browser/state checks still apply.

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

Add shinyOAuth's JavaScript to a page so [oauth_module_server\(\)](#) can redirect the browser and manage its temporary login cookie. Use this inside an existing `fluidPage()` or `tagList()` when you integrate the browser dependency directly and configure response headers elsewhere, such as in your web server or another UI wrapper.

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. Use [oauth_ui\(\)](#) to provide this protection from the start of page loading.

Details

[oauth_ui\(\)](#) combines this browser setup with the HTTP header `Referrer-Policy: no-referrer`, which prevents callback URLs from being sent as referrers when page resources load. When using [use_shinyOAuth\(\)](#) directly, set that header in your HTTP response configuration for protection from the start of page loading; the optional meta tag takes effect later. [oauth_ui\(\)](#) and [oauth_form_post_ui\(\)](#) already include this dependency. The dependency alone does not provide a callback bridge. Do not load application or third-party scripts on raw OAuth callback pages; use `oauth_ui(ui, id, client)` or a dedicated equivalent endpoint to redirect to a clean URL before rendering the app. Callback responses must also send `Cache-Control: no-store` and `Pragma: no-cache`.

Value

A `tagList` that loads the browser code once.

See Also[oauth_module_server\(\)](#)**Examples**

```
ui <- shiny::fluidPage(  
  use_shinyOAuth()  
  # ...  
)
```

Index

`base::print()`, 33

`cachem::cache_disk()`, 8

`cachem::cache_mem()`, 7, 8

`check_oauth21`, 3

`client_bearer_req`, 4

`custom_cache`, 6

`custom_cache()`, 22, 39, 51, 91

`future::plan()`, 80

`get_userinfo`, 9

`handle_callback`, 11

`handle_callback()`, 20, 43, 53, 118, 120

`httr2`, 32, 63, 115, 117, 124, 126, 133

`httr2::req_body_form()`, 32

`httr2::req_body_json()`, 32

`httr2::req_body_raw()`, 32

`httr2::req_headers()`, 32

`httr2::req_method()`, 114, 116

`httr2::req_perform()`, 126

`httr2::request()`, 32, 114, 116

`httr2::resp_body_json()`, 115

`I()`, 22, 51

`introspect_token`, 13

`is_ok_host`, 15

`is_ok_host()`, 5, 6, 78, 114–117, 125

`mirai::daemons()`, 80

`oauth_account_owner`
 (`oauth_browser_owner`), 46

`oauth_account_owner()`, 65

`oauth_browser_owner`, 46

`oauth_browser_owner()`, 65, 71

`oauth_client`, 48

`oauth_client()`, 6, 13, 16, 21–23, 27, 28, 37, 38, 42, 50, 54, 55, 60–64, 73, 74, 76, 77, 80, 85, 88, 90, 95–99, 105, 107, 109, 110, 130, 131

`oauth_client_mtls_registration`, 59

`oauth_client_secret_apple`, 61

`oauth_client_secret_apple()`, 95

`oauth_connection`, 62

`oauth_connection()`, 26, 28–30, 33, 58, 73

`oauth_connection_store_memory`, 72

`oauth_connection_store_memory()`, 48, 65

`oauth_connections`, 64

`oauth_connections()`, 26, 46, 58, 63, 67, 70, 129

`oauth_connections_server`, 67

`oauth_connections_server()`, 28–30, 63, 64, 66, 69, 70

`oauth_connections_ui`, 69

`oauth_connections_ui()`, 47, 64, 66, 67, 135, 137

`oauth_form_post_ui`, 73

`oauth_form_post_ui()`, 12, 20, 53, 78, 80, 95, 112, 113, 137

`oauth_module_server`, 76

`oauth_module_server()`, 7, 8, 11, 12, 20, 24, 28, 29, 46, 48, 53, 56, 70, 73, 74, 111, 113, 120, 121, 137, 138

`oauth_provider`, 85

`oauth_provider()`, 6, 33, 37, 38, 40, 89–92, 104, 106, 107

`oauth_provider_apple`, 95

`oauth_provider_auth0`, 96

`oauth_provider_github`, 97

`oauth_provider_google`, 97

`oauth_provider_google()`, 18, 33, 50

`oauth_provider_keycloak`, 98

`oauth_provider_microsoft`, 99

`oauth_provider_oidc`, 102

`oauth_provider_oidc()`, 38, 40, 90, 91

`oauth_provider_oidc_discover`, 105

`oauth_provider_oidc_discover()`, 18, 33,

[40, 50, 85, 91, 103, 134, 135](#)
 oauth_provider_okta, [109](#)
 oauth_provider_slack, [110](#)
 oauth_provider_spotify, [110](#)
 oauth_ui, [111](#)
 oauth_ui(), [70, 71, 73, 74, 76, 80, 137](#)
 OAuthClient, [3, 6, 10, 11, 13, 16, 28, 29, 33, 36, 40, 59, 60, 63, 64, 74, 87, 92, 98, 112, 114, 116, 118, 120, 122, 125, 127, 129, 131](#)
 OAuthConnection, [28, 63, 68, 132, 133](#)
 OAuthProvider, [3, 10, 33, 93, 95–100, 102, 104, 105, 107, 109–111](#)
 OAuthToken, [5, 6, 10–13, 29, 31, 43, 63, 79, 80, 114–116, 121–125, 127](#)

 perform_client_bearer_req, [113](#)
 perform_resource_req, [115](#)
 perform_resource_req(), [32, 33, 43, 115, 124](#)
 prepare_authorization_request, [118](#)
 prepare_authorization_request(), [27, 58, 120, 131](#)
 prepare_call, [120](#)
 prepare_call(), [11, 12, 80, 118, 119](#)
 print.OAuthConnections
 (oauth_connections), [64](#)
 print.OAuthConnectionStore
 (oauth_connection_store_memory), [72](#)
 print.OAuthOwnerPolicy
 (oauth_browser_owner), [46](#)

 refresh_token, [121](#)
 resource_req, [124](#)
 resource_req(), [6, 126](#)
 revoke_token, [127](#)

 shiny::reactiveValues(), [79](#)
 shiny::shinyApp(), [74, 75, 112, 113](#)
 smart_client, [129](#)
 smart_client(), [26, 27, 58, 63–65, 119, 132, 133, 135–137](#)
 smart_context, [132](#)
 smart_context(), [32, 45](#)
 smart_discover, [133](#)
 smart_discover(), [129](#)
 smart_fhir_user (smart_context), [132](#)
 smart_fhir_user(), [32](#)

 smart_launch_route, [135](#)
 smart_launch_route(), [68, 70](#)
 smart_patient (smart_context), [132](#)
 smart_patient(), [32](#)

 use_shinyOAuth, [137](#)
 use_shinyOAuth(), [112, 113, 137](#)