Package: shinyOAuth (via r-universe)

May 31, 2026

Title Provider-Agnostic OAuth Authentication for 'shiny' Applications

Version 0.5.0.9000

Description Provides a simple, configurable, provider-agnostic 'OAuth 2.0' and 'OpenID Connect' (OIDC) authentication framework for '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, asynchronous flows, and hooks for audit logging.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

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), 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 testthat (>= 3.0.0), knitr, rmarkdown, webfakes, promises, mirai (>= 2.0.0), future, withr, later, callr, 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

Config/pak/sysreqs cmake make libuv1-dev libssl-dev zlib1g-dev

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

Date/Publication 2026-05-31 21:22:13 UTC

RemoteUrl https://github.com/lukakoning/shinyoauth

RemoteRef HEAD

RemoteSha 42cd3523e3d7fe35116e1509327e0388fa86b906

Contents

client_bearer_req *Alias for* resource_req()

Description

[Deprecated]

Deprecated alias for `resource_req()` to avoid a breaking change in the public API. 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,
  oauth_client = NULL,
  token_type = NULL,
  dpop_nonce = 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.
`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. If `FALSE` (the default), HTTP redirects are disabled to prevent leaking the access token to unexpected hosts. 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. Set to `FALSE` only if you have already validated the URL and understand the security implications.

oauth_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.

Value

Same value as resource_req().

---

custom_cache *Create a custom cache backend (cachem-like)*

Description

Builds a small cachem-like backend object with methods compatible with what shinyOAuth needs: $get(key, missing), $set(key, value), $remove(key), and optional $info().

Use this helper when you want to plug a custom state store or JWKS cache into shinyOAuth, when cachem::cache_mem() is not suitable (e.g., multi-process deployments with non-sticky workers). In such cases, you may want to use a shared external cache (e.g., database, Redis, Memcached).

The resulting object can be used in both places where shinyOAuth accepts a cache-like object:

- OAuthClient@state_store (requires $get, $set, $remove; optional $info)
- OAuthProvider@jwks_cache (requires $get, $set; optional $remove, $info)

For OAuthClient@state_store, stored values are small lists. browser_token must always round-trip as a non-empty string. pkce_code_verifier and nonce are required only when the provider enables PKCE or nonce validation; otherwise stores may preserve them as NULL or omit them when serializing.

The $info() method is optional, but if provided and it returns a list with max_age (seconds), shinyOAuth will align browser cookie max-age in oauth_module_server() to that value.

Usage

```
custom_cache(get, set, remove, take = NULL, info = 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 $take() is provided, $remove() serves only as a best-effort cleanup and its return value is ignored. When $take() is not provided, shinyOAuth falls back to $get() + $remove() followed by a post-removal absence check via $get(key, missing = NA). In this fallback path the return value of $remove() 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 $take() instead of separate $get() + $remove() 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 cachem::cache_mem()), shinyOAuth will **error** at state consumption time because non-atomic $get() + $remove() cannot guarantee single-use under concurrent access in shared stores.
info	Function() -> list(max_age = seconds, ...). Optional TTL information from $info() is used to align browser cookie max age in oauth_module_server().

Value

An R6 object exposing cachem-like $get/$set/$remove/$info methods

Examples

```
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)
  },

  # Atomic get-and-delete: preferred for state stores in multi-worker
  # deployments to prevent TOCTOU replay attacks. For per-process caches
  # (like cachem::cache_mem()) this is optional; for shared backends (Redis,
  # database) it should map to the backend's atomic primitive (e.g., GETDEL).
  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 = 600)
)
```

`get_userinfo` *Get user info from OAuth 2.0 provider*

Description

Fetches user information from the provider's userinfo endpoint using the supplied access token. Emits an audit event with redacted details. When a validated ID token baseline is available, or when provider policy requires one, this helper also enforces OIDC UserInfo subject binding before returning.

Usage

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

Arguments

`oauth_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 pre-captured Shiny session context (from `capture_shiny_session_context()`) to include in audit events and span attributes. Used when calling from async workers that lack access to the reactive domain.

Value

A list containing the user information returned by the provider.

Examples

```
# Please note: `get_userinfo()`, `introspect_token()`, and `refresh_token()`
# are typically not called by users of this package directly, but are called
# internally by `oauth_module_server()`. These functions are exported
# nonetheless for advanced use cases. Most users will not need to
# call these functions directly

# Example requires a real token from a completed OAuth flow
# (code is therefore not run; would error with placeholder values below)
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"
  )

  # Have a valid OAuthToken object; fake example below
  # (typically provided by `oauth_module_server()` or `handle_callback()`)
  token <- handle_callback(client, "<code>", "<payload>", "<browser_token>")

  # Get userinfo
  user_info <- get_userinfo(client, token)

  # Introspect token (if supported by provider)
  introspection <- introspect_token(client, token)

  # Refresh token
  new_token <- refresh_token(client, token, introspect = TRUE)
}
```

---

handle_callback *Handle OAuth 2.0 callback: verify state, swap code for token, verify token*

Description

Completes the callback step of the login flow. It validates the callback state, exchanges the returned code for tokens, and verifies the result. This low-level helper accepts only the classic authorization-code callback shape for non-JARM clients: a `code`, the sealed `state` payload returned as `payload`, and an optional RFC 9207 `iss` callback parameter. It does not accept a raw JARM `response` JWT, and it also does not provide a public way to resume a JARM callback after separate validation. For clients configured with `response_mode` = `"jwt"`, `"query.jwt"`, or `"form_post.jwt"`, use `oauth_module_server()` (and `oauth_form_post_ui()` for `form_post.jwt`) so shinyOAuth validates the callback JWT and resumes through its internal prevalidated callback path.

Usage

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

Arguments

oauth_client	An OAuthClient object.
code	Authorization code received from the provider on a classic direct callback.
payload	Encrypted state payload returned by the provider on a classic direct callback. This should be the same value that was originally sent in prepare_call().
browser_token	Browser token present in the user's session. This is usually managed by oauth_module_server().
shiny_session	Optional pre-captured Shiny session context (from capture_shiny_session_context()) to include in audit events. Used when calling from async workers that lack access to the reactive domain.
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 oauth_client@enforce_callback_issuer is TRUE, this parameter is required and must match the configured provider issuer before any token exchange occurs.

Value

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

Examples

```
# Please note: `prepare_call()` & `handle_callback()` are typically
# not called by users of this package directly, but are called
# internally by `oauth_module_server()`. These functions are exported
# nonetheless for advanced use cases. Most users will not need to
# call these functions directly

# Below code shows generic usage of `prepare_call()` and `handle_callback()`
# (code is not run because it would require user interaction)
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 authorization URL and and store state in client's state store
  # `<browser_token>` is a token that identifies the browser session
  #  and would typically be stored in a browser cookie
  #  (`oauth_module_server()` handles this typically)
  authorization_url <- prepare_call(client, "<browser_token>")

  # Redirect user to authorization URL; retrieve code & payload from query;
  # read also `<browser_token>` from browser cookie
  # (`oauth_module_server()` handles this typically)
  code <- "..."
  payload <- "..."
  browser_token <- "..."

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

`introspect_token` *Introspect an OAuth 2.0 token*

Description

Introspects an access or refresh token when the provider exposes an introspection endpoint (RFC 7662). Returns a small result object describing whether introspection is supported and, when known, whether the token is active.

Authentication to the introspection endpoint mirrors the provider's `token_auth_style`:

- "header" (default): HTTP Basic with `client_id`/`client_secret`.
- "body": form fields `client_id` and (when available) `client_secret`.
- "public": form field `client_id` only; `client_secret` is never sent.
- "client_secret_jwt" / "private_key_jwt": a signed JWT client assertion is generated (RFC 7523) and sent via `client_assertion_type` and `client_assertion`, with aud resolved via `resolve_client_assertion_audience()` (so `client_assertion_audience` overrides are honored).

Usage

```
introspect_token(
  oauth_client,
  oauth_token,
  which = c("access", "refresh"),
  async = FALSE,
  shiny_session = NULL
)
```

Arguments

oauth_client	OAuthClient object
oauth_token	OAuthToken object to introspect
which	Which token to introspect: "access" (default) or "refresh".
async	Logical, default FALSE. If TRUE and an async backend is configured, the operation is dispatched through shinyOAuth's async promise path and this function returns a promise-compatible async result that resolves to the result list. mirai is preferred when daemons are configured via `mirai::daemons()`; otherwise the current future plan is used. Non-sequential future plans run off the main R session; `future::sequential()` stays in-process.
shiny_session	Optional pre-captured Shiny session context (from `capture_shiny_session_context()`) to include in audit events. Used when calling from async workers that lack access to the reactive domain.

Details

Best-effort semantics:

- If the provider does not expose an introspection endpoint, the function returns `supported = FALSE`, `active = NA`, and `status = "introspection_unsupported"`.
- If the endpoint responds with an HTTP error (e.g., 404/500) or the body cannot be parsed or does not include a usable `active` field, the function does not throw. It returns `supported = TRUE`, `active = NA`, and a descriptive `status` (for example, `"http_404"`, `"invalid_json"`, `"missing_active"`). In this context, `NA` means "unknown" and will not break flows unless your code explicitly requires a definitive result (i.e., `isTRUE(result$active)`).
- Providers vary in how they encode the RFC 7662 `active` field (logical, numeric, or character variants like "true"/"false", 1/0). These are normalized to logical `TRUE`/`FALSE` when possible; otherwise `active` is set to `NA`.

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

```
# Please note: `get_userinfo()`, `introspect_token()`, and `refresh_token()`
# are typically not called by users of this package directly, but are called
# internally by `oauth_module_server()`. These functions are exported
# nonetheless for advanced use cases. Most users will not need to
```

```
# call these functions directly

# Example requires a real token from a completed OAuth flow
# (code is therefore not run; would error with placeholder values below)
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"
  )

  # Have a valid OAuthToken object; fake example below
  # (typically provided by `oauth_module_server()` or `handle_callback()`)
  token <- handle_callback(client, "<code>", "<payload>", "<browser_token>")

  # Get userinfo
  user_info <- get_userinfo(client, token)

  # Introspect token (if supported by provider)
  introspection <- introspect_token(client, token)

  # Refresh token
  new_token <- refresh_token(client, token, introspect = TRUE)
}
```

`is_ok_host` *Check if URL(s) are HTTPS and/or in allowed hosts lists*

Description

Returns `TRUE` if every input URL passes shinyOAuth's scheme and host policy. In practice, each URL must be either:

- a syntactically valid HTTPS URL, and (if set) whose host matches `allowed_hosts`, or
- an HTTP URL whose host matches `allowed_non_https_hosts` (e.g. localhost, 127.0.0.1, ::1), and (if set) also matches `allowed_hosts`.

If the input omits the scheme (e.g., "localhost:8080/cb"), this function will first attempt to validate it as HTTP (useful for loopback development), and if that fails, as HTTPS. This mirrors how helpers normalize inputs for convenience while still enforcing the same host and scheme policies.

`allowed_hosts` is the allowlist of hosts or domains that are permitted, while `allowed_non_https_hosts` defines which hosts are allowed to use HTTP instead of HTTPS. If `allowed_hosts` is `NULL` or length 0, all hosts are allowed subject to the scheme rules above.

Since `allowed_hosts` supports globs, a value like "*" matches any host and therefore effectively disables endpoint host restrictions. Only use a catch-all pattern when you truly intend to allow any host. In most deployments you should pin to your expected domain(s), e.g. `c(".example.com")` or a specific host name.

Wildcards: `allowed_hosts` and `allowed_non_https_hosts` support globs: `*` = any chars, ? = one char. A leading `.example.com` matches the domain itself and any subdomain.

Any non-URLs, NAs, or empty strings cause a FALSE result.

Usage

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

Arguments

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

Details

This function is used internally to validate redirect URIs in OAuth clients, but can also be used directly to test whether URLs would be accepted. Internally, the defaults come from the options `shinyOAuth.allowed_non_https_hosts` and `shinyOAuth.allowed_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("*"))
```

`oauth_client` *Create generic OAuthClient*

Description

Main helper for creating a validated OAuthClient configuration before `oauth_module_server()` starts login or callback handling.

Usage

```
oauth_client(
  provider,
  client_id,
  client_secret = character(0),
  redirect_uri,
  scopes = character(0),
  response_mode = NULL,
  resource = character(0),
  claims = NULL,
  enforce_callback_issuer = NULL,
  scope_validation = c("warn", "strict", "none"),
  claims_validation = c("none", "warn", "strict"),
  required_acr_values = character(0),
  userinfo_jwt_required_time_claims = character(0),
  introspect = FALSE,
  introspect_elements = character(0),
  state_store = cachem::cache_mem(max_age = 300),
  state_payload_max_age = 300,
  state_entropy = 64,
  state_key = random_urlsafe(128),
  client_assertion_private_key = NULL,
  client_assertion_private_key_kid = NULL,
  client_assertion_alg = NULL,
  client_assertion_audience = NULL,
  mtls_client_cert_file = NULL,
  mtls_client_key_file = NULL,
  mtls_client_key_password = NULL,
  mtls_client_ca_file = NULL,
  mtls_certificate_bound_access_tokens = FALSE,
  dpop_private_key = NULL,
  dpop_private_key_kid = NULL,
  dpop_signing_alg = NULL,
  dpop_require_access_token = NULL,
  dpop_require_observed_cnf = FALSE,
  request_object_mode = c("parameters", "request", "request_uri"),
  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,
  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,
  ...
)
```

Arguments

`provider`	OAuthProvider object
`client_id`	OAuth client ID
`client_secret`	OAuth client secret. Validation rules: • Required (non-empty) when the provider authenticates the client with HTTP Basic auth at the token endpoint (`token_auth_style = "header"`, also known as `client_secret_basic`). • Optional when the provider uses form-body client authentication at the token endpoint (`token_auth_style = "body"`, also known as `client_secret_post`) and `use_pkce = TRUE`. In that configuration, the secret is omitted only when it is empty. • Ignored for token-endpoint authentication when the provider uses `token_auth_style = "public"` (or the alias `"none"`). Public auth sends `client_id` only and never sends `client_secret`, even if one is configured explicitly. Note: If your provider issues HS256 ID tokens and `id_token_validation` is enabled, a non-empty `client_secret` is required for signature validation.
`redirect_uri`	Redirect URI registered with provider
`scopes`	Vector of scopes to request. For OIDC providers (those with an `issuer`), shinyOAuth automatically prepends `openid` when it is missing; that effective scope set is what gets sent in the authorization request and used for later state and token-scope validation.
`response_mode`	Authorization response mode for authorization-code callbacks. Supported values are `"query"`, `"form_post"`, `"jwt"`, `"query.jwt"`, and `"form_post.jwt"`. The effective default is always `"query"`: omitting this argument keeps the normal query-parameter callback flow and shinyOAuth does not send a `response_mode` parameter. Pass `"query"` only if you need to explicitly request the query response mode from the provider. Set `"form_post"` only when the provider requires or explicitly recommends POSTing the authorization response to the redirect URI. Shiny apps using `"form_post"` must wrap their UI with `oauth_form_post_ui()`. Prefer this argument over setting `extra_auth_params$response_mode` on the

provider. When the provider advertises response_modes_supported, the resolved mode must be included in that set. "jwt" requests the JARM-defined default callback transport for the response type; for the authorization-code flow that still means a query callback, but shinyOAuth preserves and sends "jwt" when you configure it explicitly. "fragment.jwt" is not currently supported because shinyOAuth does not implement fragment callback transport.

JARM callbacks are currently module-only. For "jwt", "query.jwt", and "form_post.jwt", use oauth_module_server() and, for "form_post.jwt", wrap the app UI with oauth_form_post_ui(). The exported handle_callback() helper still accepts only the classic direct code + sealed state callback shape and does not expose a public JARM validation/resume API.

resource

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

claims

OIDC claims request parameter (OIDC Core section 5.5). Allows requesting specific claims from the UserInfo Endpoint and/or in the ID Token. Can be:

- NULL (default): no claims parameter is sent
- A list: automatically JSON-encoded (via jsonlite::toJSON() with auto_unbox = TRUE) and URL-encoded into the authorization request. The list should have top-level members userinfo and/or id_token, each containing named lists of claims. Use NULL to request a claim without parameters (per spec). Example: list(userinfo = list(email = NULL, given_name = list(essential = TRUE)), id_token = list(auth_time = list(essential = TRUE))) Note on single-element arrays: because auto_unbox = TRUE is used, single-element R vectors are serialized as JSON scalars, not arrays. The OIDC spec defines values as an array. To force array encoding for a single-element vector, wrap it in I(), e.g., acr = list(values = I("urn:mace:incommon:iap:silver")) produces {"values":["urn:mace:incommon:iap:silver"]}. Multi-element vectors are always encoded as arrays. shinyOAuth warns when it sees a single-element values entry that is not wrapped in I(), because that common input pattern serializes incorrectly for OIDC.
- A character string: pre-encoded JSON string (advanced use). Must be valid JSON. Use this when you need full control over JSON encoding. Note: The claims parameter is OPTIONAL per OIDC Core section 5.5. Not all providers support it; consult your provider's documentation.

enforce_callback_issuer

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

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

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

Controls validation of requested claims supplied via the claims parameter (OIDC Core section 5.5). When claims includes entries with essential = TRUE for id_token or userinfo, or explicit value / values constraints for individual claims, this setting determines what happens if the returned ID token or userinfo response does not satisfy those requests.

- "none": Skips claims validation entirely. This remains the effective default when the supplied claims request has no enforceable essential, value, or values constraints, and when you explicitly set claims_validation = "none".
- "warn": Emits a warning but continues authentication if requested essential claims are missing or requested claim values are not satisfied.
- "strict": Throws an error if any requested essential claims are missing or requested claim value / values constraints are not satisfied by the response.

If claims_validation is omitted and the supplied claims request does include enforceable essential, value, or values constraints, oauth_client() promotes the effective default to "warn" so those mismatches are surfaced by default.

Enforceable requests under claims$id_token require a validated ID token. Configure the provider with id_token_validation = TRUE or use_nonce = TRUE so shinyOAuth validates the ID token before checking those claims.

required_acr_values

Optional character vector of acceptable Authentication Context Class Reference values (OIDC Core sections 2 and 3.1.2.1). When non-empty, the ID token returned by the provider must contain an acr claim whose value is one of the specified entries; otherwise the login fails with a shinyOAuth_id_token_error.

Additionally, when non-empty, the authorization request automatically includes an acr_values query parameter (space-separated) as a voluntary hint to the provider (OIDC Core section 3.1.2.1). Note that the provider is not required to honour this hint; the client-side validation is the authoritative enforcement.

Requires an OIDC-capable provider with id_token_validation = TRUE and an issuer configured. Default is character(0) (no enforcement).

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"`.

`introspect` If TRUE, the login flow will call the provider's token introspection endpoint (RFC 7662) to validate the access token. The login is not considered complete unless introspection succeeds and returns `active = TRUE`; otherwise the login fails and `authenticated` remains FALSE. When `oauth_module_server()` later performs proactive refresh, it also forwards this setting so refreshed access tokens are introspected through the same client policy. Default is FALSE. Requires the provider to have an `introspection_url` configured.

`introspect_elements`
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.)

`state_store` State storage backend. Defaults to `cachem::cache_mem(max_age = 300)`. Alternative backends should use `custom_cache()` with an atomic `$take()` method for replay-safe single-use state consumption. The backend must implement cachem-like methods `$get(key, missing)`, `$set(key, value)`, and `$remove(key)`; `$info()` is optional.

Stored values must round-trip `browser_token` as a non-empty string. `pkce_code_verifier` and `nonce` are required only when the provider enables PKCE or nonce validation; otherwise backends may keep those fields as `NULL` or omit them.

`cachem::cache_mem()` is a good default for a single Shiny process. For multi-process deployments, use `custom_cache()` with an atomic `$take()` backed by a shared store (for example Redis `GETDEL` or SQL `DELETE ... RETURNING`). Plain `cachem::cache_disk()` is **not safe** as a shared state store because its `$get()` + `$remove()` operations are not atomic.

The client automatically generates, persists (in `state_store`), and validates the OAuth `state` parameter (and OIDC `nonce` when applicable) during the authorization code flow.

`state_payload_max_age`
Positive number of seconds. Maximum allowed age for the decrypted state payload's `issued_at` timestamp during callback validation.

This is the freshness window for the sealed `state` payload itself. It is separate

from the state_store TTL, which controls how long the one-time server-side state entry can exist.

Default is 300 seconds.

state_entropy Integer. The length (in characters) of the randomly generated state parameter. Higher values provide more entropy and better security against CSRF attacks. Must be between 22 and 128 (to align with validate_state()'s default minimum which targets ~128 bits for base64url-like strings). Default is 64.

state_key Optional per-client secret used as the state sealing key for AES-GCM AEAD (authenticated encryption) of the state payload that travels via the state query parameter. This provides confidentiality and integrity (via authentication tag) for the embedded data used during callback verification. If you omit this argument, a random value is generated via random_urlsafe(128). This key is distinct from the OAuth client_secret and may be used with public clients.

Type: character string (>= 32 bytes when encoded) or raw vector (>= 32 bytes). Raw keys enable direct use of high-entropy secrets from external stores. Both forms are normalized internally by cryptographic helpers.

Multi-process deployments: if your app runs with multiple R workers or behind a non-sticky load balancer, configure a shared state_store and the same state_key across all workers. Otherwise callbacks that land on a different worker will fail state validation.

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'. Ignored for other auth styles. Current outbound private-key JWT signing supports RSA and EC private keys. For RSA keys, outbound signing is currently limited to RS256; RS384, RS512, and RSA-PSS (PS256, PS384, PS512) are not supported. Ed25519/Ed448 keys are also not currently supported.

client_assertion_private_key_kid

Optional key identifier (kid) to include in the JWT header for private_key_jwt assertions. 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). 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, ES256, ES384, ES512) for private keys. RS384, RS512, PS256, PS384, PS512, and EdDSA 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

exact token endpoint request URL. Some identity providers require a different audience value; set this to the exact value your IdP expects.

`mtls_client_cert_file`
: Optional path to the PEM-encoded client certificate (or certificate chain) used for RFC 8705 mutual TLS client authentication and certificate-bound protected-resource requests. Required when `provider@token_auth_style` is `"tls_client_auth"` or `"self_signed_tls_client_auth"`.

`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 that should fail closed if the returned access token omits `cnf.x5t#S256`.

  Requires `mtls_client_cert_file` and `mtls_client_key_file`, and the provider must be configured with `mtls_client_certificate_bound_access_tokens = TRUE`.

`dpop_private_key`
: Optional private key used to generate DPoP proofs (RFC 9449). Can be an `openssl::key` or a PEM string containing an asymmetric private key. When provided, shinyOAuth can attach `DPoP` proofs to token endpoint requests and use DPoP-bound access tokens in downstream request helpers. In `oauth_client()`, configuring this key also makes `dpop_require_access_token` default to `TRUE`, so access-token responses reject `token_type = "Bearer"` unless you explicitly set `dpop_require_access_token = FALSE`. Current outbound DPoP signing supports RSA and EC private keys. For RSA keys, outbound signing is currently limited to `RS256`; `RS384`, `RS512`, and RSA-PSS (`PS256`, `PS384`, `PS512`) are not supported. Ed25519/Ed448 keys are also not currently supported. This is an advanced setting; most clients do not need DPoP unless their provider or resource server asks for it.

`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). RS384, RS512, PS256, PS384, PS512, and EdDSA 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. When shinyOAuth can observe token binding data from a JWT access token or an introspection response, this strict mode also requires cnf$jkt to be present and match the configured dpop_private_key. Opaque access tokens that expose no cnf data still pass this check unless introspection later reveals the binding. In oauth_client(), 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.

dpop_require_observed_cnf

Logical. When TRUE, shinyOAuth rejects token_type = "DPoP" access tokens unless it can observe cnf$jkt locally, either from the access token itself or from a token introspection response. Use this when high-assurance DPoP deployments must fail closed on opaque access tokens that provide no observable binding. Default is 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.

Most users can keep the default. Request mode is an advanced option that requires 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. If a caller-managed request_uri uses HTTP and the configured host policy explicitly allows it, shinyOAuth still publishes it but warns once per R session because RFC 9101 Section 5.2 expects client-provided request_uri values to use HTTPS. 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.

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, ES256, ES384,

or ES512). RS384, RS512, PS256, PS384, PS512, and EdDSA 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.

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, 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.

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

Value

OAuthClient object

Examples

```
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"
  )

  # 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 <- fluidPage(
  use_shinyOAuth(),
  uiOutput("login")
)

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 <- fluidPage(
  use_shinyOAuth(),
  actionButton("login_btn", "Login"),
  uiOutput("login")
)

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

  observeEvent(input$login_btn, {
    auth$request_login()
  })

  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 <- fluidPage(
  use_shinyOAuth(),
  uiOutput("ui")
```

```
)

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

  repositories <- reactiveVal(NULL)

  observe({
    req(auth$authenticated)

    # Example additional API request using the access token
    # (e.g., fetch user repositories from GitHub)
    resp <- perform_resource_req(
      auth$token,
      "https://api.github.com/user/repos"
    )

    if (httr2::resp_is_error(resp)) {
      repositories(NULL)
    } else {
      repos_data <- httr2::resp_body_json(resp, simplifyVector = TRUE)
      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$h4("Your repositories:"),
        if (!is.null(repos)) {
          tags$ul(
            Map(
              function(url, name) {
                tags$li(tags$a(href = url, target = "_blank", 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_client_mtls_registration` *Build RFC 8705 mTLS registration metadata*

Description

Returns a JSON-ready list of client metadata for registering an OAuthClient that uses RFC 8705 mutual TLS or requests certificate-bound access tokens.

For `token_auth_style = "tls_client_auth"`, this helper returns `token_endpoint_auth_method = "tls_client_auth"` plus exactly one RFC 8705 certificate identifier field: `tls_client_auth_subject_dn`, `tls_client_auth_san_dns`, `tls_client_auth_san_uri`, `tls_client_auth_san_ip`, or `tls_client_auth_san_email`.

For `token_auth_style = "self_signed_tls_client_auth"`, this helper returns `token_endpoint_auth_method = "self_signed_tls_client_auth"` plus either an inline `jwks` document built from the configured client certificate and certificate chain (published via `x5c`), or a caller-supplied `jwks_uri`.

For clients that request RFC 8705 certificate-bound access tokens without mTLS OAuth client authentication, this helper returns the runtime `token_auth_style` mapped back to the dynamic-registration metadata value (for example, `public` becomes `none`) and emits `tls_client_certificate_bound_access_toke = TRUE`.

This helper prepares metadata only. It does not make a registration HTTP call.

Usage

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

Arguments

`oauth_client`	OAuthClient configured for RFC 8705 mutual TLS client authentication or for certificate-bound access tokens.
`tls_client_auth_type`	For `tls_client_auth`, which RFC 8705 certificate identifier field to emit. One of `"subject_dn"`, `"san_dns"`, `"san_uri"`, `"san_ip"`, or `"san_email"`.
`tls_client_auth_value`	Optional explicit value for the selected `tls_client_auth_type`. When omitted, shinyOAuth derives the subject DN or, when possible, a unique matching SAN value from the configured client certificate. Auto-derived IP SAN values are normalized to dotted-decimal IPv4 or RFC 5952 IPv6 text. If the certificate exposes no unambiguous SAN for the chosen type, pass the exact registration value explicitly.
`jwks_uri`	Optional absolute URL of a JWKS document to publish for `self_signed_tls_client_auth`. When omitted, the helper returns an inline `jwks` object with the configured client certificate chain in `x5c`.

Value

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

`oauth_client_secret_apple` *Create an Apple client secret JWT*

Description

Builds the ES256-signed JWT that Apple expects in the token-request `client_secret` form field for Sign in with Apple.

Usage

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

Arguments

client_id	Apple Services ID or App ID used as the OAuth client id
team_id	Apple Developer Team ID. Apple documents this as a 10-character identifier
key_id	Apple Sign in with Apple private-key identifier (kid). Apple documents this as a 10-character identifier
private_key	Apple private key as an openssl::key or PEM string. The key must be compatible with ES256 (P-256 ECDSA)
expires_in	Positive 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

Apple currently requires the following JWT shape for Sign in with Apple token requests:

- JOSE header alg = ES256 and kid = <Apple key id>
- iss = <Apple Developer Team ID>
- sub = <client_id>
- aud = "https://appleid.apple.com"
- exp no more than 15777000 seconds (six months) after iat

The resulting string can be supplied directly to oauth_client() as the client_secret for oauth_provider_apple().

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

Description

`oauth_form_post_ui()` enables the OpenID Foundation OAuth 2.0 Form Post Response Mode for apps that use `oauth_module_server()`. It wraps your existing Shiny UI so a provider can POST an authorization response to the app's redirect URI. The POST body is stored server-side under a short-lived one-time handle, and the browser is redirected back to the app with only that opaque handle in the query string.

For most apps, this helper is not needed because the default transport for authorization responses is the query string, which works without this UI wrapper. You only need to use this helper if your provider requires or strongly recommends form_post response mode.

To request form_post response mode from the provider, wrap your UI with this helper, configure your OAuthClient with `response_mode = "form_post"`, and ensure the `redirect_uri` is set to a URL that routes to this UI wrapper (e.g., the app's root URL or a specific callback path). This helper handles the plain form_post response mode, where the POST body contains authorization response parameters such as `code`, `state`, `error`, and `iss`. When `response_mode = "form_post.jwt"`, the helper validates the inbound JARM `response`, decrypts and validates the enclosed state, and then stores the accepted callback payload under the same one-time handle so the main callback flow can resume from a prevalidated POST boundary.

Usage

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

Arguments

`base_ui`	Existing Shiny UI object, or a UI function accepting `req`.
`id`	Shiny module id used by `oauth_module_server()`. This must match the `id` argument passed to the server module.
`client`	OAuthClient object used by `oauth_module_server()`.
`callback_path`	Optional URL path to accept POST callbacks on. Defaults to the path component of `client@redirect_uri`.

Details

When this wrapper is used, it also injects `use_shinyOAuth()` automatically for the wrapped GET UI, so you do not need a separate top-level `use_shinyOAuth()` call.

The server-side callback handle is single-use and is rejected if it is older than the smaller of `client@state_payload_max_age` and the configured `state_store` TTL. The raw POST body and transient handle query parameters are also bounded by the `shinyOAuth.callback_max_form_post_*` options described in the usage vignette.

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.

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)

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

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

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

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

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

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

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

`oauth_module_server` *OAuth 2.0 & OIDC authentication module for Shiny applications*

Description

This function implements a Shiny module server that manages OAuth 2.0/OIDC authentication for Shiny applications. It handles the OAuth 2.0/OIDC flow, including redirecting users to the authorization endpoint, securely processing the callback, exchanging authorization codes for tokens, verifying tokens, and managing token refresh. It also provides options for automatic or manual login flows, session expiry, and proactive token refresh.

Note: when using this module, you must include `shinyOAuth::use_shinyOAuth()` in your UI definition to load the necessary JavaScript dependencies.

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 = 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")
)
```

Arguments

`id`	Shiny module id
`client`	OAuthClient object

`auto_redirect` If TRUE (default), unauthenticated sessions will immediately initiate the OAuth flow by redirecting the browser to the authorization endpoint. If FALSE, the module will not auto-redirect; instead, the returned object exposes helpers for triggering login manually (use `$request_login()`).

`async` If TRUE, dispatches token exchange and refresh through shinyOAuth's async promise path and updates values when the promise resolves. mirai is preferred when daemons are configured with `mirai::daemons()`. Otherwise, if promises and future are installed, the current future plan is used. Non-sequential future plans run off the main R session; `future::sequential()` stays in-process. If FALSE (default), token exchange and refresh are performed synchronously (which may block the Shiny event loop). For production apps, `async = TRUE` is usually the better choice.

`indefinite_session`
If TRUE, the module will not automatically clear the token due to access-token expiry or the `reauth_after_seconds` window, and it will not trigger automatic reauthentication when a token expires or a refresh fails. 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.

`reauth_after_seconds`
Optional maximum session age in seconds. If set, the module will remove the token (and thus set `authenticated` to FALSE) after this many seconds have elapsed since authentication started. By default this is `NULL` (no forced re-authentication). If a value is provided, the timer is reset after each successful refresh so the knob is opt-in and counts rolling session age.

`refresh_proactively`
If TRUE, will automatically refresh tokens before they expire (if refresh token is available). The refresh is scheduled adaptively so that it executes approximately at `expires_at - refresh_lead_seconds` rather than on a coarse polling loop.

`refresh_lead_seconds`
Number of seconds before expiry to attempt proactive refresh (default: 60)

`refresh_check_interval`
Fallback check interval in milliseconds for expiry/refresh (default: 10000 ms). When expiry is known, the module uses adaptive scheduling to wake up exactly when needed; this interval is used as a safety net or when expiry is unknown/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 not include a query string or fragment. Non-HTTPS hosts still follow the same `?is_ok_host` policy as other package URLs, but shinyOAuth warns once per R session because RFC 9101 Section 5.2 expects client-provided `request_uri` values to use HTTPS.

`browser_cookie_path`
: Optional cookie Path to scope the browser token cookie. By default (`NULL`), the path is fixed to "/" for reliable clearing across route changes. Provide an explicit path (e.g., "/app") to narrow the cookie's scope to a sub-route. Explicit values must start with `/` and must not contain semicolons or control characters. Note: when the path is "/" and the page is served over HTTPS, the cookie name uses the `__Host-` prefix (Secure, Path=/) for additional hardening; when the path is not "/", a regular cookie name is used.

  For apps deployed under nested routes or where the OAuth callback may land on a different route than the initial page, keeping the default (root path) ensures the browser token cookie is available and clearable across app routes. If you deliberately scope the cookie to a sub-path, make sure all relevant routes share that prefix.

`browser_cookie_samesite`
: SameSite value for the browser-token cookie. One of "Strict", "Lax", or "None". Defaults to "Strict" for maximum protection against cross-site request forgery. Use "Lax" only when your deployment requires the cookie to accompany top-level cross-site navigations (for example, because of reverse-proxy flows), and document the associated risk. If set to "None", the cookie will be marked `SameSite=None; Secure` in the browser, and authentication will error on non-HTTPS origins because browsers reject `SameSite=None` cookies without the `Secure` attribute

Details

Most apps only need to decide whether login starts automatically, whether to enable async mode, and whether token refresh should happen proactively. The remaining arguments are mainly for deployments that need tighter control over session lifetime, logout behavior, or browser cookie settings.

- Blocking vs. async behavior: when `async = FALSE` (the default), network operations like token exchange and refresh are performed on the main R thread. Transient errors are retried

by the package's internal `req_with_retry()` helper, which currently uses `Sys.sleep()` for backoff. In Shiny, `Sys.sleep()` blocks the event loop for the entire worker process, potentially freezing UI updates for all sessions on that worker during slow provider responses or retry backoff. To keep the UI responsive: set `async = TRUE` and configure an async backend that runs off the main process, such as mirai daemons (`mirai::daemons(n)`) or a non-sequential future plan, or reduce/block retries (see `vignette("usage", package = "shinyOAuth")`).

- Browser requirements: the module relies on the browser's Web Crypto API to generate a secure, per-session browser token used for state double-submit protection. Specifically, the login flow requires `window.crypto.getRandomValues` to be available. If it is not present (for example, in some very old or highly locked-down browsers), the module will be unable to proceed with authentication. In that case a client-side error is emitted and surfaced to the server as `shinyOAuth_cookie_error` containing the message `"webcrypto_unavailable"`. Use a modern browser (or enable Web Crypto) to resolve this.
- Browser cookie lifetime: the opaque browser token cookie lifetime mirrors the client's `state_store` TTL. Internally, the module reads `client@state_store$info()$max_age` and uses that value for the cookie's `Max-Age/Expires`. When the cache does not expose a finite `max_age`, a conservative default of 5 minutes (300 seconds) is used to align with the built-in `cachem::cache_mem(max_age = 300)` default. Separately, the state payload `issued_at` freshness window is controlled by the client's `state_payload_max_age` (default 300 seconds).

Value

A reactiveValues object with `token`, `error`, `error_description`, `error_uri`, and `authenticated`, plus additional fields used by the module.

The returned reactiveValues contains the following fields:

- `authenticated`: logical TRUE when there is no error and a token is present and valid (matching the verifications enabled in the client provider); FALSE otherwise. Exception: when `indefinite_session = TRUE`, errors do not affect this flag so `authenticated` remains TRUE even if refresh or other operations fail.
- `token`: OAuthToken object, or NULL if not yet authenticated. This contains the access token, refresh token (if any), ID token (if any), userinfo (if fetched), and the decoded ID token claims via `token@id_token_claims` (a read-only named list exposing all JWT payload claims such as `sub`, `acr`, `amr`, `auth_time`, etc.). See OAuthToken for details. Because OAuthToken is a S7 object, you access its fields with `@`, e.g., `token@userinfo` or `token@id_token_claims$acr`.
- `error`: error code string when the OAuth flow fails. Be careful about showing this directly to users, because it may contain sensitive information.
- `error_description`: human-readable error detail when available. Be extra careful about showing this directly to users, because it may contain even more sensitive information.
- `error_uri`: URI identifying a human-readable web page with information about the error (per RFC 6749 section 4.1.2.1). Treat this as untrusted navigation input; shinyOAuth only surfaces absolute HTTPS values here when they stay on a provider host or another host already allowlisted via `options(shinyOAuth.allowed_hosts = ...)`, and returns NULL when the provider omits or sends an unsafe value.
- `browser_token`: internal opaque browser cookie value; used for state double-submit protection; NULL if not yet set

- pending_callback: internal deferred callback payload; stores either list(type = "code", code, state, iss) for authorization-code callbacks or list(type = "error", error, error_description, error_uri, state, iss) for provider error callbacks. Used to defer callback handling until browser_token is available; NULL otherwise.
- pending_login: internal logical; TRUE when a login was requested but must wait for browser_token to be set, FALSE otherwise.
- auto_redirected: internal logical; TRUE once the module has initiated an automatic redirect in this session to avoid duplicate redirects.
- reauth_triggered: internal logical; TRUE once a reauthentication attempt has been initiated (after expiry or failed refresh), to avoid loops.
- auth_started_at: internal numeric timestamp (as from Sys.time()) when authentication started; NA if not yet authenticated. Used to enforce reauth_after_seconds if set.
- token_stale: logical; TRUE when the token was kept despite a refresh failure because indefinite_session = TRUE, or when the access token is past its expiry but indefinite_session = TRUE prevents automatic clearing. This lets UIs warn users or disable actions that require a fresh token. It resets to FALSE on successful login, refresh, or logout.
- last_login_async_used: internal logical; TRUE if the last login attempt used async = TRUE, FALSE if it was synchronous. This is only used for testing and diagnostics.
- refresh_in_progress: internal logical; TRUE while a token refresh is currently in flight (async or sync). Used to prevent concurrent refresh attempts when proactive refresh logic wakes up multiple times.

It also contains the following helper functions, mainly useful when auto_redirect = FALSE and you want to start login from your own UI (for example, from a button):

- request_login(): initiates login by redirecting to the authorization endpoint, with cookie-ensure semantics: if browser_token is missing, the module sets the cookie and defers the redirect until browser_token is present, then redirects. If the module is already authenticated, the request is ignored and no new OAuth state is created. This is the main entry point for login when auto_redirect = FALSE.
- logout(): if a token is present, makes best-effort revocation requests for the refresh token and access token when the provider exposes a revocation endpoint. This may perform network I/O, can revoke refresh tokens, and follows the module's async setting. It then clears the current token, sets authenticated to FALSE, and rotates the browser token cookie. You might call this when the user clicks a logout button.
- build_auth_url(): internal; builds and returns the authorization URL, also storing the relevant state in the client's state_store (for validation during callback). Note that this requires browser_token to be present, so it will throw an error if called too early. When the module is already authenticated it returns NA and does not mint new state (verify with has_browser_token() first). When PAR is used, the returned string keeps shinyOAuth.par_request_uri, shinyOAuth.par_expires_in, and shinyOAuth.par_expires_at attributes so manual link-style flows can decide when to regenerate it. Typically you would not call this directly, but use request_login() instead, which calls it internally.
- set_browser_token(): internal; injects JS to set the browser token cookie if missing. Normally called automatically on first load, but you can call it manually if needed. If a token is already present, it will return immediately without changing it (call clear_browser_token() if

you want to force a reset). Typically you would not call this directly, but use `request_login()` instead, which calls it internally if needed.

- `clear_browser_token()`: internal; injects JS to clear the browser token cookie and clears `browser_token`. You might call this to reset the cookie if you suspect it's stale or compromised. Typically you would not call this directly.
- `has_browser_token()`: internal; returns TRUE if `browser_token` is present (non-NULL, non-empty), FALSE otherwise. Typically you would not call this directly

See Also

`use_shinyOAuth()`

Examples

```
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"
  )

  # 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 <- fluidPage(
  use_shinyOAuth(),
  uiOutput("login")
)

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 <- fluidPage(
  use_shinyOAuth(),
  actionButton("login_btn", "Login"),
  uiOutput("login")
)

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

```
  )

  observeEvent(input$login_btn, {
    auth$request_login()
  })

  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 <- fluidPage(
  use_shinyOAuth(),
  uiOutput("ui")
)

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

  repositories <- reactiveVal(NULL)

  observe({
    req(auth$authenticated)

    # Example additional API request using the access token
    # (e.g., fetch user repositories from GitHub)
    resp <- perform_resource_req(
```

```
      auth$token,
      "https://api.github.com/user/repos"
    )

    if (httr2::resp_is_error(resp)) {
      repositories(NULL)
    } else {
      repos_data <- httr2::resp_body_json(resp, simplifyVector = TRUE)
      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$h4("Your repositories:"),
        if (!is.null(repos)) {
          tags$ul(
            Map(
              function(url, name) {
                tags$li(tags$a(href = url, target = "_blank", 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 *Create generic OAuthProvider*

Description

Helper to create an OAuthProvider object with sensible defaults. It is the main user-facing constructor for generic providers and is also used by the built-in provider helpers.

Usage

```
oauth_provider(
  name,
  auth_url,
  token_url,
  issuer = NA_character_,
  issuer_match = "url",
  token_auth_style = "header",
  use_pkce = TRUE,
  pkce_method = "S256",
  use_nonce = NULL,
  userinfo_url = NA_character_,
  userinfo_required = NULL,
  userinfo_id_selector = function(userinfo) {
     userinfo[["sub"]]
 },
  userinfo_id_token_match = NULL,
  userinfo_signed_jwt_required = FALSE,
  id_token_required = NULL,
  id_token_validation = NULL,
  id_token_at_hash_required = FALSE,
  introspection_url = NA_character_,
  revocation_url = NA_character_,
  extra_auth_params = list(),
  extra_token_params = list(),
  extra_token_headers = character(),
  jwks_uri = NA_character_,
  jwks_cache = NULL,
  jwks_pins = character(),
  jwks_pin_mode = "any",
  jwks_host_issuer_match = NULL,
  jwks_host_allow_only = NULL,
 allowed_algs = c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512", "EdDSA"),
  allowed_token_types = c("Bearer"),
  leeway = getOption("shinyOAuth.leeway", 30),
  par_url = NA_character_,
  par_required = FALSE,
  signed_request_object_required = FALSE,
```

```
  request_parameter_supported = NA,
  request_uri_parameter_supported = NA,
  request_uri_registration_required = NA,
  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,
  authorization_request_front_channel_mode = "compat",
  authorization_response_iss_parameter_supported = FALSE,
  response_modes_supported = character(),
  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,
  token_endpoint_auth_signing_alg_values_supported = character(),
  dpop_signing_alg_values_supported = character(),
  mtls_endpoint_aliases = list(),
  mtls_client_certificate_bound_access_tokens = FALSE,
  ...
)
```

Arguments

`name`	Provider name (e.g., "github", "google"). Cosmetic only; used in logging and audit events
`auth_url`	Authorization endpoint URL
`token_url`	Token endpoint URL
`issuer`	Optional OIDC issuer URL. You need this when you want ID token validation. shinyOAuth uses it to verify the ID token `iss` claim and to locate the provider's signing keys (JWKS), typically through the OIDC discovery document at `/.well-known/openid-configuration`.
`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 exactly after removing one trailing slash, if present, from both the configured issuer and the discovery metadata value. • `"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.
`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)

use_pkce
: Whether to use PKCE. This adds a code_challenge parameter to the authorization request and requires a code_verifier when exchanging the authorization code for tokens. This helps protect against authorization code interception attacks.

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".

use_nonce
: Whether to use OIDC nonce. This adds a nonce parameter to the authorization request and validates the nonce claim in the ID token. For OIDC providers, leaving this enabled is usually the right choice.

userinfo_url
: User info endpoint URL (optional)

userinfo_required
: Whether to fetch userinfo after token exchange. User information will be stored in the userinfo field of the returned OAuthToken object. This requires a valid userinfo_url to be set. If fetching userinfo fails, login fails.

  For the low-level constructor oauth_provider(), when not explicitly supplied, this is inferred from the presence of a non-empty userinfo_url: if a userinfo_url is provided, userinfo_required defaults to TRUE, otherwise it defaults to FALSE. This avoids unexpected validation errors when userinfo_url is omitted (since it is optional).

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 user identifier, such as audit fields and UserInfo-to-ID-token subject matching. If you configure a selector other than function(x) x$sub, that selector also defines which UserInfo value is compared against the validated ID token sub. Helper constructors like oauth_provider() and oauth_provider_oidc() provide a default selector that extracts the sub field.

userinfo_id_token_match
: Whether to fail closed if UserInfo cannot be bound to a validated ID token subject. Whenever both UserInfo and a validated ID token are available, shinyOAuth compares the validated ID token sub to the value returned by userinfo_id_selector(userinfo). Setting this field to TRUE additionally requires a validated ID token baseline whenever UserInfo is fetched. This requires userinfo_required, a configured

userinfo_id_selector, plus either id_token_validation or use_nonce to be TRUE.

For oauth_provider(), when not explicitly supplied, this is inferred as TRUE when userinfo_required is TRUE and either id_token_validation or use_nonce is TRUE; otherwise it defaults to FALSE.

userinfo_signed_jwt_required

Whether to require that the userinfo endpoint returns a signed JWT (Content-Type: application/jwt) whose signature can be verified against the provider's JWKS. This is an advanced hardening option. When TRUE:

- If the userinfo response is not application/jwt, authentication fails.
- If the JWT uses alg=none or an algorithm not in the asymmetric subset of allowed_algs (RS*, ES*, or EdDSA), authentication fails. HS* algorithms are not accepted for UserInfo JWTs on this surface even if they appear in allowed_algs.
- If signature verification fails (JWKS fetch error, no compatible keys, or invalid signature), authentication fails.

This prevents unsigned or weakly signed userinfo payloads from being treated as trusted identity data. Requires userinfo_required = TRUE and a valid issuer (for JWKS). Defaults to FALSE.

Note: oauth_provider_oidc_discover() does not auto-enable this flag. Discovery's userinfo_signing_alg_values_supported indicates provider capability, not that every client actually receives signed JWTs. Pass userinfo_signed_jwt_required = TRUE explicitly if you need this behavior.

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.

Note: At the S7 class level, this defaults to FALSE so that pure OAuth 2.0 providers can be configured without OIDC. Helper constructors like oauth_provider() and oauth_provider_oidc() will enable this when an issuer is supplied or OIDC is explicitly requested.

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).

Note: At the S7 class level, this defaults to FALSE. Helper constructors like oauth_provider() and oauth_provider_oidc() turn this on when an issuer is provided or when OIDC is used.

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.

introspection_url

Token introspection endpoint URL (optional; RFC 7662)

`revocation_url` Token revocation endpoint URL (optional; RFC 7009)

`extra_auth_params`
: Extra parameters for authorization URL

`extra_token_params`
: Extra parameters for token exchange

`extra_token_headers`
: Extra headers for back-channel token-style requests (named character vector). shinyOAuth applies these headers to token exchange, refresh, introspection, revocation, and PAR requests. Use this only for headers you intentionally want on that full set of authorization-server calls.

`jwks_uri` Optional explicit URL of the provider's JWK Set document. Use this when a generic OAuth 2.0 or JARM deployment publishes signing keys outside OIDC discovery, or when you intentionally want to override runtime metadata-based JWKS resolution.

In most cases, a TTL between 15 minutes and 2 hours is reasonable. Shorter TTLs pick up new keys faster but do more network work; longer TTLs reduce traffic but may take longer to notice key rotation. If a new `kid` appears, shinyOAuth will also do a one-time refresh automatically.

`jwks_cache` Cache used for the provider's signing keys (JWKS). If not provided, shinyOAuth creates an in-memory cache for 1 hour with `cachem::cache_mem(max_age = 3600)`. You can also use another cachem-compatible backend, including a shared cache created with `custom_cache()`.

`jwks_pins` Optional character vector of RFC 7638 JWK thumbprints (base64url) to pin against. If non-empty, fetched JWKS must contain keys whose thumbprints match these values depending on `jwks_pin_mode`. This is an advanced hardening option that lets you pre-authorize expected keys.

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

`jwks_host_issuer_match`
: When TRUE, enforce that the discovery `jwks_uri` host matches the issuer host exactly. Defaults to FALSE at the class level, but helper constructors for OIDC (e.g., `oauth_provider_oidc()` and `oauth_provider_oidc_discover()`) enable this by default 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 `jwks_host_allow_only` to the exact hostname rather than disabling this check.

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

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 EdDSA for OKP-backed signatures. When ID token at_hash validation is in play, Ed25519 is supported. Ed448 at_hash cannot be validated with the current crypto bindings, so shinyOAuth skips that optional check unless id_token_at_hash_required = TRUE, in which case Ed448 ID tokens fail fast. 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","EdDSA"), which intentionally excludes HS*. 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 always include token_type; 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.

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. Most users only need this when their provider specifically supports or requires PAR.

par_required
: Logical. Whether the provider requires authorization requests to be sent via PAR. When TRUE, par_url must also be configured.

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".

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.

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.

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 the current OIDC-compatible shape 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.

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.

`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`.

`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.

`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.

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

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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

Ready-to-use OAuthProvider settings for Sign in with Apple.

Usage

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

Arguments

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

Details

This helper resolves Sign in with Apple's current metadata from Apple's OIDC discovery document at `https://appleid.apple.com/.well-known/openid-configuration`.

Apple does not publish a userinfo endpoint, so this helper relies on the validated ID token for subject and claim data and leaves `userinfo_required = FALSE`.

When configuring your OAuthClient:

- use your Services ID or App ID as `client_id`
- supply `client_secret` as an Apple-signed ES256 JWT, for example via `oauth_client_secret_apple()`
- use an HTTPS redirect URI with a domain name; Apple does not allow IP literals or `localhost`
- if you request `email` or `name`, configure `oauth_client(..., response_mode = "form_post")` and wrap your UI with `oauth_form_post_ui()`

Apple can return a one-time `user` JSON payload on the front-channel form_post callback when `email` or `name` are requested. shinyOAuth does not currently map that transient payload into the returned OAuthToken `userinfo` field, so this helper leaves `userinfo_required = FALSE` and relies on ID token claims.

Because this helper delegates to `oauth_provider_oidc_discover()`, any discovery-backed metadata Apple publishes in the future is picked up automatically. When a particular discovery field is omitted, shinyOAuth keeps the same defaults documented for `oauth_provider_oidc_discover()`.

Value

OAuthProvider object configured for Sign in with Apple

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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_auth0` *Create an Auth0 OAuthProvider (via OIDC discovery)*

Description

Create an Auth0 OAuthProvider (via OIDC discovery)

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

```
# 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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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_github` *Create a GitHub OAuthProvider*

Description

Ready-to-use OAuth 2.0 provider settings for GitHub.

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 'Developer Settings'.

Value

OAuthProvider object for use with a GitHub OAuth 2.0 app

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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_google` *Create a Google OAuthProvider*

Description

Ready-to-use OAuthProvider settings for Google.

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.

Value

OAuthProvider object for use with a Google OAuth 2.0 app

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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_keycloak` *Create a Keycloak OAuthProvider (via OIDC discovery)*

Description

Create a Keycloak OAuthProvider (via OIDC discovery)

Usage

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

Arguments

`base_url`	Base URL of the Keycloak server, e.g., "http://localhost:8080"
`realm`	Keycloak realm name, e.g., "myrealm"
`name`	Optional provider name. Defaults to `paste0('keycloak-', realm)`
`token_auth_style`	Optional override for token endpoint authentication method. One of "header" (client_secret_basic), "body" (client_secret_post), "public" (send `client_id` only; `"none"` alias also accepted), "private_key_jwt", or "client_secret_jwt". Defaults to "body" for Keycloak, 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

```
# 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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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_microsoft` *Create a Microsoft (Entra ID) OAuthProvider*

Description

Ready-to-use OAuthProvider settings for Microsoft Entra ID (formerly Azure AD) using the v2.0 endpoints. Accepts a tenant identifier and configures the authorization, token, and userinfo endpoints directly.

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

Most users only need to choose the tenant and decide whether to keep ID token validation enabled. The remaining details below explain how the helper behaves for Microsoft's different tenant styles.

The `tenant` can be one of the special values "common", "organizations", or "consumers", or a specific directory (tenant) ID GUID (e.g., "00000000-0000-0000-0000-000000000000").

When `tenant` is a specific GUID, the provider enables strict ID token validation with the tenant-specific issuer.

For `tenant = "common"` or `tenant = "organizations"`, the helper enables Microsoft Entra's tenant-independent validation mode by default: ID tokens are checked against Microsoft's `{tenantid}` issuer template and the signing key's own `issuer` scope, as documented by Microsoft for multi-tenant metadata. Runtime JWKS discovery for these aliases also uses host-only discovery issuer matching because Microsoft's tenant-independent metadata publishes a templated issuer rather than echoing the alias URL exactly.

For `tenant = "consumers"`, the helper resolves the stable consumer tenant issuer (`9188040d-6c67-4c5b-b112-36a304b66d`) and performs normal exact- issuer validation.

Set `id_token_validation = FALSE` to opt out of ID token and nonce validation for these aliases, which falls back to OAuth 2.0 plus userinfo identity only.

Microsoft issues RS256 ID tokens; `allowed_algs` is restricted accordingly. The userinfo endpoint is provided by Microsoft Graph (https://graph.microsoft.com/oidc/userinfo).

When configuring your OAuthClient, if you do not have the option to register an app or simply wish to test during development, you may be able to use the default Azure CLI public app, with `client_id` '04b07795-8ddb-461a-bbee-02f9e1bf7b46' (uses `redirect_uri` 'http://localhost:8100').

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")
) {
  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")
    ),
    # Default Azure CLI app ID (public client; activated in many tenants):
    client_id = "04b07795-8ddb-461a-bbee-02f9e1bf7b46",
    client_secret = "",
    redirect_uri = "http://localhost:8100",
    scopes = c("openid", "profile", "email")
  )
```

```
# UI
ui <- fluidPage(
  use_shinyOAuth(),
  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")
)

# 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$user_info <- renderPrint({
    req(auth$token)
    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)
}
```

---

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

Description

Helper for providers that follow a standard OpenID Connect endpoint layout. It builds the usual OIDC endpoints from one base URL and then calls `oauth_provider()` with OIDC-friendly defaults.

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"),
  ...
)
```

Arguments

`name`	Friendly name for the provider
`base_url`	Base URL for OIDC endpoints
`auth_path`	Authorization endpoint path (default: "/authorize")
`token_path`	Token endpoint path (default: "/token")
`userinfo_path`	User info endpoint path (default: "/userinfo")
`introspection_path`	Token introspection endpoint path (default: "/introspect")
`use_nonce`	Logical, whether to use OIDC nonce. Defaults to TRUE
`id_token_validation`	Logical, whether to validate ID tokens automatically for this provider. Defaults to TRUE
`jwks_host_issuer_match`	When TRUE (default), 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 `jwks_host_allow_only` to the exact hostname instead of disabling this. Disabling (`FALSE`) is not recommended unless you also pin JWKS via `jwks_host_allow_only` or `jwks_pins`
`allowed_token_types`	Character vector of allowed token types for access tokens issued by this provider. Defaults to 'Bearer'
`...`	Additional arguments passed to `oauth_provider()`

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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

Builds an OAuthProvider from the provider's OpenID Connect discovery document at `/.well-known/openid-configurati`
When present, `introspection_endpoint` is also wired into the resulting provider.

Usage

```
oauth_provider_oidc_discover(
  issuer,
  name = NULL,
  use_pkce = TRUE,
  use_nonce = TRUE,
  id_token_validation = TRUE,
  token_auth_style = NULL,
 allowed_algs = c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512", "EdDSA"),
  allowed_token_types = c("Bearer"),
  jwks_host_issuer_match = TRUE,
  issuer_match = c("url", "host", "none"),
  ...
)
```

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 and normalized back to the issuer base URL before validation and fetch.
`name`	Optional friendly provider name. Defaults to the issuer hostname

use_pkce
: Logical, whether to use PKCE for this provider. Defaults to TRUE. If the discovery document indicates token_endpoint_auth_methods_supported includes "none", PKCE is required unless use_pkce is explicitly set to FALSE (not recommended)

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. When PKCE is enabled and the provider advertises support for public clients via none, discovery selects "public". Otherwise, the helper prefers "header" (client_secret_basic) when available, then "body" (client_secret_post). JWT-based methods are not auto-selected unless explicitly requested.

allowed_algs
: Character vector of allowed ID token signing algorithms. Defaults to a broad set of common algorithms, including RSA (RS*), ECDSA (ES*), and EdDSA. If the discovery document advertises supported algorithms, the intersection of advertised and caller-provided algorithms is used to avoid runtime mismatches. If there's no overlap, discovery fails with a configuration error (no fallback).

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

jwks_host_issuer_match
: When TRUE (default), enforce that the JWKS host discovered from the provider matches the issuer host exactly. For providers that serve JWKS from a different host, set jwks_host_allow_only to the exact hostname instead of disabling this. Disabling (FALSE) is not recommended unless you also pin JWKS via jwks_host_allow_only or jwks_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 after normalizing a full discovery-document input back to its issuer base URL and removing one trailing slash, if present, from both values (recommended).
  - "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).

Details

Most users can accept the defaults here. The points below are mainly reference for advanced provider setups or for understanding why discovery might fail early.

- ID token algorithms: by default this helper accepts common asymmetric algorithms RSA (RS*), ECDSA (ES*), and EdDSA. When the provider advertises its supported ID token signing algorithms via `id_token_signing_alg_values_supported`, the helper uses the intersection with the caller-provided `allowed_algs`. If there is no overlap, discovery fails with a configuration error. There is no automatic fallback to the discovery-advertised set.
- Token endpoint authentication methods: supports `client_secret_basic` (header), `client_secret_post` (body), public clients using `none` (mapped to `token_auth_style = "public"` when PKCE is enabled), as well as JWT-based methods `private_key_jwt` and `client_secret_jwt` per RFC 7523. Discovery also preserves RFC 8705 mTLS metadata (`mtls_endpoint_aliases` and `tls_client_certificate_bound_access_tokens`) and supports explicit `tls_client_auth` / `self_signed_tls_client_auth` selection.
- PAR metadata: when the discovery document advertises `pushed_authorization_request_endpoint` or `require_pushed_authorization_requests`, the resulting provider stores that PAR capability and policy metadata in `par_required` so authorization requests can use RFC 9126 PAR and fail fast on PAR-only provider policies.
- Request Object metadata: when the discovery document advertises `request_object_signing_alg_values_supporte` or `require_signed_request_object`, the resulting provider stores that metadata in `signed_request_object_requir` so `OAuthClient` can fail fast when a request-object algorithm is unsupported or when the provider requires signed Request Objects. When the discovery document also advertises `request_object_encryption_alg_values_supported` or `request_object_encryption_enc_values_supported`, the resulting provider stores that encryption metadata so Request Object JWE configuration can be validated early as well.
- Authorization request transport metadata: when the discovery document advertises `request_parameter_supported`, `request_uri_parameter_supported`, or `require_request_uri_registration`, the resulting provider stores that metadata so shinyOAuth can fail fast when a provider explicitly disallows the front-channel `request` transport used by JAR or caller-managed `request_uri` values. The registration requirement itself remains deployment-specific: shinyOAuth stores `request_uri_registration_required` for caller awareness, but it cannot independently verify whether the provider has already registered a matching public `request_uri` or wildcard prefix for the client. When PAR is configured, shinyOAuth sends signed Request Objects to the PAR endpoint and the browser redirect only carries the PAR-issued `request_uri` handle, regardless of `request_uri_parameter_supported` or `request_uri_registration_required`. When discovery omits these booleans, this helper applies the OpenID Connect defaults instead of storing `NA`.
- Response mode metadata: when the discovery document advertises `response_modes_supported`, the resulting provider stores it so explicit `response_mode` requests can fail fast when unsupported. When the metadata is omitted, this helper applies the OAuth/OIDC metadata default of `c("query", "fragment")`.
- Token endpoint JWT auth metadata: when the discovery document advertises `token_endpoint_auth_signing_alg_va` the resulting provider stores that metadata so `OAuthClient` can fail fast when a JWT client assertion algorithm is unsupported.

- DPoP metadata: when the discovery document advertises dpop_signing_alg_values_supported, the resulting provider stores that metadata so OAuthClient can fail fast when an explicit or inferred DPoP proof signing algorithm is unsupported.
- RFC 9207 callback issuer metadata: when the discovery document advertises authorization_response_iss_paramet = true, the resulting provider stores that metadata so oauth_client() can auto-enable callback issuer enforcement unless you explicitly opt out.
- PKCE method discovery: this helper keeps S256 as the default and does not silently downgrade to plain. If discovery metadata explicitly omits S256, discovery fails with a configuration error unless you explicitly opt into pkce_method = "plain".

  Important: discovery metadata lists methods supported across the provider, not per-client provisioning. This helper does not automatically select JWT-based methods just because they are advertised. By default it prefers client_secret_basic (header) when available, otherwise client_secret_post (body), and maps public none to token_auth_style = "public" only for PKCE clients. If a provider advertises only JWT methods, you must explicitly set token_auth_style and configure the corresponding credentials on your OAuthClient (a private key for private_key_jwt, or a sufficiently strong client_secret for client_secret_jwt).
- Host policy: by default, discovered standard endpoints must be absolute URLs whose host matches the issuer host exactly. Subdomains are NOT implicitly allowed. If you want to allow subdomains, add a leading-dot or glob in options(shinyOAuth.allowed_hosts), e.g., .example.com or *.example.com. If a global whitelist is supplied via options(shinyOAuth.allowed_hosts), discovery will restrict endpoints to that whitelist. RFC 8705 mtls_endpoint_aliases are validated separately: they may use a different host or port by default, but an explicit shinyOAuth.allowed_hosts whitelist still constrains them. Scheme policy (https/http for loopback) is delegated to is_ok_host(), so you may allow non-HTTPS hosts with options(shinyOAuth.allowed_non_https_hosts) (see ?is_ok_host).

Value

OAuthProvider object configured from discovery

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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

Create an Okta OAuthProvider (via OIDC discovery)

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

```
# 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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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_slack` *Create a Slack OAuthProvider (via OIDC discovery)*

Description

Create a Slack OAuthProvider (via OIDC discovery)

Usage

```
oauth_provider_slack(name = "slack")
```

Arguments

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

Value

OAuthProvider object configured for Slack

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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_spotify` *Create a Spotify OAuthProvider*

Description

Ready-to-use OAuth 2.0 provider settings for Spotify. It uses `/v1/me` as the user profile endpoint and does not expect ID tokens.

Usage

```
oauth_provider_spotify(name = "spotify")
```

Arguments

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

Details

Spotify requires scopes to be included in the authorization request. Set requested scopes on the client with `oauth_client(..., scopes = ...)`.

Value

OAuthProvider object for use with a Spotify OAuth 2.0 app

See Also

For an example application which using Spotify OAuth 2.0 login to display the user's listening data, see `vignette("example-spotify")`.

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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")
```

```
}
```

`OAuthClient` *OAuthClient S7 class*

Description

S7 class describing an OAuth 2.0 client configuration. It combines the provider, client credentials, redirect URI, requested scopes, and the state handling rules used during login and callback validation.

This is a low-level constructor intended for advanced use. Most users should prefer the helper constructor `oauth_client()`.

Usage

```
OAuthClient(
  provider = NULL,
  client_id = character(0),
  client_secret = character(0),
  redirect_uri = character(0),
  scopes = character(0),
  response_mode = NA_character_,
  resource = character(0),
  claims = NULL,
  enforce_callback_issuer = FALSE,
  scope_validation = "warn",
  claims_validation = "none",
  required_acr_values = character(0),
  userinfo_jwt_required_time_claims = character(0),
  introspect = FALSE,
  introspect_elements = character(0),
  state_store = cachem::cache_mem(max_age = 300),
  state_payload_max_age = 300,
  state_entropy = 64,
  state_key = random_urlsafe(n = 128),
  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,
  dpop_private_key = NULL,
  dpop_private_key_kid = NA_character_,
  dpop_signing_alg = NA_character_,
```

```
  dpop_require_access_token = FALSE,
  dpop_require_observed_cnf = FALSE,
  request_object_mode = "parameters",
  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_,
  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
)
```

Arguments

`provider`	OAuthProvider object
`client_id`	OAuth client ID
`client_secret`	OAuth client secret. Validation rules: • Required (non-empty) when the provider authenticates the client with HTTP Basic auth at the token endpoint (`token_auth_style = "header"`, also known as `client_secret_basic`). • Optional when the provider uses form-body client authentication at the token endpoint (`token_auth_style = "body"`, also known as `client_secret_post`) and `use_pkce = TRUE`. In that configuration, the secret is omitted only when it is empty. • Ignored for token-endpoint authentication when the provider uses `token_auth_style = "public"` (or the alias `"none"`). Public auth sends `client_id` only and never sends `client_secret`, even if one is configured explicitly. Note: If your provider issues HS256 ID tokens and `id_token_validation` is enabled, a non-empty `client_secret` is required for signature validation.
`redirect_uri`	Redirect URI registered with provider
`scopes`	Vector of scopes to request. For OIDC providers (those with an `issuer`), shinyOAuth automatically prepends `openid` when it is missing; that effective scope set is what gets sent in the authorization request and used for later state and token-scope validation.
`response_mode`	Authorization response mode for authorization-code callbacks. Supported values are `"query"`, `"form_post"`, `"jwt"`, `"query.jwt"`, and `"form_post.jwt"`. The effective default is always `"query"`: omitting this argument keeps the normal query-parameter callback flow and shinyOAuth does not send a `response_mode`

parameter. Pass "query" only if you need to explicitly request the query response mode from the provider. Set "form_post" only when the provider requires or explicitly recommends POSTing the authorization response to the redirect URI. Shiny apps using "form_post" must wrap their UI with oauth_form_post_ui(). Prefer this argument over setting extra_auth_params$response_mode on the provider. When the provider advertises response_modes_supported, the resolved mode must be included in that set. "jwt" requests the JARM-defined default callback transport for the response type; for the authorization-code flow that still means a query callback, but shinyOAuth preserves and sends "jwt" when you configure it explicitly. "fragment.jwt" is not currently supported because shinyOAuth does not implement fragment callback transport.

JARM callbacks are currently module-only. For "jwt", "query.jwt", and "form_post.jwt", use oauth_module_server() and, for "form_post.jwt", wrap the app UI with oauth_form_post_ui(). The exported handle_callback() helper still accepts only the classic direct code + sealed state callback shape and does not expose a public JARM validation/resume API.

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

claims
: OIDC claims request parameter (OIDC Core section 5.5). Allows requesting specific claims from the UserInfo Endpoint and/or in the ID Token. Can be:

  - NULL (default): no claims parameter is sent
  - A list: automatically JSON-encoded (via jsonlite::toJSON() with auto_unbox = TRUE) and URL-encoded into the authorization request. The list should have top-level members userinfo and/or id_token, each containing named lists of claims. Use NULL to request a claim without parameters (per spec). Example: list(userinfo = list(email = NULL, given_name = list(essential = TRUE)), id_token = list(auth_time = list(essential = TRUE))) Note on single-element arrays: because auto_unbox = TRUE is used, single-element R vectors are serialized as JSON scalars, not arrays. The OIDC spec defines values as an array. To force array encoding for a single-element vector, wrap it in I(), e.g., acr = list(values = I("urn:mace:incommon:iap:silver")) produces {"values":["urn:mace:incommon:iap:silver"]}. Multi-element vectors are always encoded as arrays. shinyOAuth warns when it sees a single-element values entry that is not wrapped in I(), because that common input pattern serializes incorrectly for OIDC.
  - A character string: pre-encoded JSON string (advanced use). Must be valid JSON. Use this when you need full control over JSON encoding. Note: The claims parameter is OPTIONAL per OIDC Core section 5.5. Not all providers support it; consult your provider's documentation.

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

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

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

Controls validation of requested claims supplied via the claims parameter (OIDC Core section 5.5). When claims includes entries with essential = TRUE for id_token or userinfo, or explicit value / values constraints for individual claims, this setting determines what happens if the returned ID token or userinfo response does not satisfy those requests.

- "none": Skips claims validation entirely. This remains the effective default when the supplied claims request has no enforceable essential, value, or values constraints, and when you explicitly set claims_validation = "none".
- "warn": Emits a warning but continues authentication if requested essential claims are missing or requested claim values are not satisfied.
- "strict": Throws an error if any requested essential claims are missing or requested claim value / values constraints are not satisfied by the response.

If claims_validation is omitted and the supplied claims request does include enforceable essential, value, or values constraints, oauth_client() promotes the effective default to "warn" so those mismatches are surfaced by default.

Enforceable requests under claims$id_token require a validated ID token. Configure the provider with id_token_validation = TRUE or use_nonce = TRUE so shinyOAuth validates the ID token before checking those claims.

required_acr_values

Optional character vector of acceptable Authentication Context Class Reference values (OIDC Core sections 2 and 3.1.2.1). When non-empty, the ID token returned by the provider must contain an acr claim whose value is one of the specified entries; otherwise the login fails with a shinyOAuth_id_token_error.

Additionally, when non-empty, the authorization request automatically includes an acr_values query parameter (space-separated) as a voluntary hint to the provider (OIDC Core section 3.1.2.1). Note that the provider is not required to honour this hint; the client-side validation is the authoritative enforcement.

Requires an OIDC-capable provider with id_token_validation = TRUE and an issuer configured. Default is character(0) (no enforcement).

`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"`.

`introspect`
: If TRUE, the login flow will call the provider's token introspection endpoint (RFC 7662) to validate the access token. The login is not considered complete unless introspection succeeds and returns `active = TRUE`; otherwise the login fails and `authenticated` remains FALSE. When `oauth_module_server()` later performs proactive refresh, it also forwards this setting so refreshed access tokens are introspected through the same client policy. Default is FALSE. Requires the provider to have an `introspection_url` configured.

`introspect_elements`
: 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.)

`state_store`
: State storage backend. Defaults to `cachem::cache_mem(max_age = 300)`. Alternative backends should use `custom_cache()` with an atomic `$take()` method for replay-safe single-use state consumption. The backend must implement cachem-like methods `$get(key, missing)`, `$set(key, value)`, and `$remove(key)`; `$info()` is optional.

  Stored values must round-trip `browser_token` as a non-empty string. `pkce_code_verifier` and `nonce` are required only when the provider enables PKCE or nonce validation; otherwise backends may keep those fields as `NULL` or omit them.

  `cachem::cache_mem()` is a good default for a single Shiny process. For multi-process deployments, use `custom_cache()` with an atomic `$take()` backed by a shared store (for example Redis `GETDEL` or SQL `DELETE ... RETURNING`). Plain `cachem::cache_disk()` is **not safe** as a shared state store because its `$get()` + `$remove()` operations are not atomic.

  The client automatically generates, persists (in `state_store`), and validates the OAuth `state` parameter (and OIDC `nonce` when applicable) during the authorization code flow.

`state_payload_max_age`
: Positive number of seconds. Maximum allowed age for the decrypted state payload's `issued_at` timestamp during callback validation.

  This is the freshness window for the sealed `state` payload itself. It is separate from the `state_store` TTL, which controls how long the one-time server-side state entry can exist.

  Default is 300 seconds.

`state_entropy`
: Integer. The length (in characters) of the randomly generated state parameter. Higher values provide more entropy and better security against CSRF attacks. Must be between 22 and 128 (to align with `validate_state()`'s default minimum which targets ~128 bits for base64url-like strings). Default is 64.

`state_key`
: Optional per-client secret used as the state sealing key for AES-GCM AEAD (authenticated encryption) of the state payload that travels via the `state` query parameter. This provides confidentiality and integrity (via authentication tag) for the embedded data used during callback verification. If you omit this argument, a random value is generated via `random_urlsafe(128)`. This key is distinct from the OAuth `client_secret` and may be used with public clients.

  Type: character string (>= 32 bytes when encoded) or raw vector (>= 32 bytes). Raw keys enable direct use of high-entropy secrets from external stores. Both forms are normalized internally by cryptographic helpers.

  Multi-process deployments: if your app runs with multiple R workers or behind a non-sticky load balancer, configure a shared `state_store` and the same `state_key` across all workers. Otherwise callbacks that land on a different worker will fail state validation.

`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'`. Ignored for other auth styles. Current outbound private-key JWT signing supports RSA and EC private keys. For RSA keys, outbound signing is currently limited to `RS256`; `RS384`, `RS512`, and RSA-PSS (`PS256`, `PS384`, `PS512`) are not supported. Ed25519/Ed448 keys are also not currently supported.

`client_assertion_private_key_kid`
: Optional key identifier (kid) to include in the JWT header for `private_key_jwt` assertions. 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). 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`, `ES256`, `ES384`, `ES512`) for private keys. `RS384`, `RS512`, `PS256`, `PS384`, `PS512`, and `EdDSA` 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 exact token endpoint request URL. Some identity providers require a different audience value; set this to the exact value your IdP expects.

mtls_client_cert_file
: Optional path to the PEM-encoded client certificate (or certificate chain) used for RFC 8705 mutual TLS client authentication and certificate-bound protected-resource requests. Required when provider@token_auth_style is "tls_client_auth" or "self_signed_tls_client_auth".

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 that should fail closed if the returned access token omits cnf.x5t#S256.

  Requires mtls_client_cert_file and mtls_client_key_file, and the provider must be configured with mtls_client_certificate_bound_access_tokens = TRUE.

dpop_private_key
: Optional private key used to generate DPoP proofs (RFC 9449). Can be an openssl::key or a PEM string containing an asymmetric private key. When provided, shinyOAuth can attach DPoP proofs to token endpoint requests and use DPoP-bound access tokens in downstream request helpers. In oauth_client(), configuring this key also makes dpop_require_access_token default to TRUE, so access-token responses reject token_type = "Bearer" unless you explicitly set dpop_require_access_token = FALSE. Current outbound DPoP signing supports RSA and EC private keys. For RSA keys, outbound signing is currently limited to RS256; RS384, RS512, and RSA-PSS (PS256, PS384, PS512) are not supported. Ed25519/Ed448 keys are also not currently supported. This is an advanced setting; most clients do not need DPoP unless their provider or resource server asks for it.

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). RS384, RS512, PS256, PS384, PS512, and EdDSA 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. When shinyOAuth can observe token binding data from a JWT access token or an introspection response, this strict mode also requires cnf$jkt to be present and match the configured dpop_private_key. Opaque access tokens that expose no cnf data still pass this check unless introspection later reveals the binding. In oauth_client(), 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.

dpop_require_observed_cnf
: Logical. When TRUE, shinyOAuth rejects token_type = "DPoP" access tokens unless it can observe cnf$jkt locally, either from the access token itself or from a token introspection response. Use this when high-assurance DPoP deployments must fail closed on opaque access tokens that provide no observable binding. Default is 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.

  Most users can keep the default. Request mode is an advanced option that requires 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. If a caller-managed request_uri uses HTTP and the configured host policy explicitly allows it, shinyOAuth still publishes it but warns once per R session because RFC 9101 Section 5.2 expects client-provided request_uri values to use HTTPS. 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.

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, ES256, ES384, or ES512). RS384, RS512, PS256, PS384, PS512, and EdDSA 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.

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, 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.

Examples

```
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"
  )

  # 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 <- fluidPage(
  use_shinyOAuth(),
  uiOutput("login")
)

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 <- fluidPage(
```

```
  use_shinyOAuth(),
  actionButton("login_btn", "Login"),
  uiOutput("login")
)

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

  observeEvent(input$login_btn, {
    auth$request_login()
  })

  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 <- fluidPage(
  use_shinyOAuth(),
  uiOutput("ui")
)

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

```
  )

  repositories <- reactiveVal(NULL)

  observe({
    req(auth$authenticated)

    # Example additional API request using the access token
    # (e.g., fetch user repositories from GitHub)
    resp <- perform_resource_req(
      auth$token,
      "https://api.github.com/user/repos"
    )

    if (httr2::resp_is_error(resp)) {
      repositories(NULL)
    } else {
      repos_data <- httr2::resp_body_json(resp, simplifyVector = TRUE)
      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$h4("Your repositories:"),
        if (!is.null(repos)) {
          tags$ul(
            Map(
              function(url, name) {
                tags$li(tags$a(href = url, target = "_blank", 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
    )
  }
}
```

OAuthProvider *OAuthProvider S7 class*

Description

S7 class describing an OAuth 2.0 or OpenID Connect provider. It stores the provider's endpoints and the rules shinyOAuth should follow during login, callback handling, token exchange, and optional OIDC checks.

This is a low-level constructor intended for advanced use. Most users should prefer the helper constructors `oauth_provider()` for generic OAuth 2.0 providers or `oauth_provider_oidc()` / `oauth_provider_oidc_discover()` for OpenID Connect providers. Those helpers enable secure defaults based on the presence of an issuer and available endpoints.

Usage

```
OAuthProvider(
  name = character(0),
  auth_url = character(0),
  token_url = character(0),
  issuer = NA_character_,
  issuer_match = "url",
  token_auth_style = "header",
  use_pkce = TRUE,
  pkce_method = "S256",
  use_nonce = FALSE,
  userinfo_url = NA_character_,
  userinfo_required = FALSE,
  userinfo_id_selector = function(userinfo) userinfo[["sub"]],
  userinfo_id_token_match = FALSE,
  userinfo_signed_jwt_required = FALSE,
  id_token_required = FALSE,
  id_token_validation = FALSE,
  id_token_at_hash_required = FALSE,
  introspection_url = NA_character_,
  revocation_url = NA_character_,
  extra_auth_params = list(),
  extra_token_params = list(),
  extra_token_headers = character(0),
  jwks_uri = NA_character_,
```

```
  jwks_cache = cachem::cache_mem(max_age = 3600),
  jwks_pins = character(0),
  jwks_pin_mode = "any",
  jwks_host_issuer_match = FALSE,
  jwks_host_allow_only = NA_character_,
 allowed_algs = c("RS256", "RS384", "RS512", "ES256", "ES384", "ES512", "EdDSA"),
  allowed_token_types = "Bearer",
  par_url = NA_character_,
  par_required = FALSE,
  signed_request_object_required = FALSE,
  request_parameter_supported = NA,
  request_uri_parameter_supported = NA,
  request_uri_registration_required = NA,
  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,
  authorization_request_front_channel_mode = "compat",
  authorization_response_iss_parameter_supported = FALSE,
  response_modes_supported = character(0),
  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,
  token_endpoint_auth_signing_alg_values_supported = character(0),
  dpop_signing_alg_values_supported = character(0),
  mtls_endpoint_aliases = list(),
  mtls_client_certificate_bound_access_tokens = FALSE,
  leeway = getOption("shinyOAuth.leeway", 30)
)
```

Arguments

`name`	Provider name (e.g., "github", "google"). Cosmetic only; used in logging and audit events
`auth_url`	Authorization endpoint URL
`token_url`	Token endpoint URL
`issuer`	Optional OIDC issuer URL. You need this when you want ID token validation. shinyOAuth uses it to verify the ID token `iss` claim and to locate the provider's signing keys (JWKS), typically through the OIDC discovery document at `/.well-known/openid-configuration`.
`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 exactly after removing one trailing slash, if present, from both the configured issuer and the discovery metadata value.

- `"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.

`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)

`use_pkce`
: Whether to use PKCE. This adds a `code_challenge` parameter to the authorization request and requires a `code_verifier` when exchanging the authorization code for tokens. This helps protect against authorization code interception attacks.

`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".

`use_nonce`
: Whether to use OIDC nonce. This adds a `nonce` parameter to the authorization request and validates the `nonce` claim in the ID token. For OIDC providers, leaving this enabled is usually the right choice.

`userinfo_url`
: User info endpoint URL (optional)

`userinfo_required`
: Whether to fetch userinfo after token exchange. User information will be stored in the `userinfo` field of the returned `OAuthToken` object. This requires a valid `userinfo_url` to be set. If fetching userinfo fails, login fails.

  For the low-level constructor `oauth_provider()`, when not explicitly supplied, this is inferred from the presence of a non-empty `userinfo_url`: if a `userinfo_url` is provided, `userinfo_required` defaults to `TRUE`, otherwise it defaults to `FALSE`. This avoids unexpected validation errors when `userinfo_url` is omitted (since it is optional).

`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 user identifier, such as audit fields and UserInfo-to-ID-token subject matching. If you configure a selector other than `function(x) x$sub`, that selector also defines which UserInfo value is compared against the validated ID token `sub`. Helper constructors

like oauth_provider() and oauth_provider_oidc() provide a default selector that extracts the sub field.

userinfo_id_token_match

Whether to fail closed if UserInfo cannot be bound to a validated ID token subject. Whenever both UserInfo and a validated ID token are available, shinyOAuth compares the validated ID token sub to the value returned by userinfo_id_selector(userinfo). Setting this field to TRUE additionally requires a validated ID token baseline whenever UserInfo is fetched. This requires userinfo_required, a configured userinfo_id_selector, plus either id_token_validation or use_nonce to be TRUE.

For oauth_provider(), when not explicitly supplied, this is inferred as TRUE when userinfo_required is TRUE and either id_token_validation or use_nonce is TRUE; otherwise it defaults to FALSE.

userinfo_signed_jwt_required

Whether to require that the userinfo endpoint returns a signed JWT (Content-Type: application/jwt) whose signature can be verified against the provider's JWKS. This is an advanced hardening option. When TRUE:

- If the userinfo response is not application/jwt, authentication fails.
- If the JWT uses alg=none or an algorithm not in the asymmetric subset of allowed_algs (RS*, ES*, or EdDSA), authentication fails. HS* algorithms are not accepted for UserInfo JWTs on this surface even if they appear in allowed_algs.
- If signature verification fails (JWKS fetch error, no compatible keys, or invalid signature), authentication fails.

This prevents unsigned or weakly signed userinfo payloads from being treated as trusted identity data. Requires userinfo_required = TRUE and a valid issuer (for JWKS). Defaults to FALSE.

Note: oauth_provider_oidc_discover() does not auto-enable this flag. Discovery's userinfo_signing_alg_values_supported indicates provider capability, not that every client actually receives signed JWTs. Pass userinfo_signed_jwt_required = TRUE explicitly if you need this behavior.

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.

Note: At the S7 class level, this defaults to FALSE so that pure OAuth 2.0 providers can be configured without OIDC. Helper constructors like oauth_provider() and oauth_provider_oidc() will enable this when an issuer is supplied or OIDC is explicitly requested.

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).

Note: At the S7 class level, this defaults to FALSE. Helper constructors like oauth_provider() and oauth_provider_oidc() turn this on when an issuer is provided or when OIDC is used.

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.

introspection_url
: Token introspection endpoint URL (optional; RFC 7662)

revocation_url
: Token revocation endpoint URL (optional; RFC 7009)

extra_auth_params
: Extra parameters for authorization URL

extra_token_params
: Extra parameters for token exchange

extra_token_headers
: Extra headers for back-channel token-style requests (named character vector). shinyOAuth applies these headers to token exchange, refresh, introspection, revocation, and PAR requests. Use this only for headers you intentionally want on that full set of authorization-server calls.

jwks_uri
: Optional explicit URL of the provider's JWK Set document. Use this when a generic OAuth 2.0 or JARM deployment publishes signing keys outside OIDC discovery, or when you intentionally want to override runtime metadata-based JWKS resolution.

  In most cases, a TTL between 15 minutes and 2 hours is reasonable. Shorter TTLs pick up new keys faster but do more network work; longer TTLs reduce traffic but may take longer to notice key rotation. If a new kid appears, shinyOAuth will also do a one-time refresh automatically.

jwks_cache
: Cache used for the provider's signing keys (JWKS). If not provided, shinyOAuth creates an in-memory cache for 1 hour with cachem::cache_mem(max_age = 3600). You can also use another cachem-compatible backend, including a shared cache created with custom_cache().

jwks_pins
: Optional character vector of RFC 7638 JWK thumbprints (base64url) to pin against. If non-empty, fetched JWKS must contain keys whose thumbprints match these values depending on jwks_pin_mode. This is an advanced hardening option that lets you pre-authorize expected keys.

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

jwks_host_issuer_match
: When TRUE, enforce that the discovery jwks_uri host matches the issuer host exactly. Defaults to FALSE at the class level, but helper constructors for OIDC (e.g., oauth_provider_oidc() and oauth_provider_oidc_discover()) enable this by default 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 jwks_host_allow_only to the exact hostname rather than disabling this check.

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

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 `EdDSA` for OKP-backed signatures. When ID token `at_hash` validation is in play, Ed25519 is supported. Ed448 `at_hash` cannot be validated with the current crypto bindings, so shinyOAuth skips that optional check unless `id_token_at_hash_required = TRUE`, in which case Ed448 ID tokens fail fast. 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","EdDSA")`, which intentionally excludes HS*. 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 always include `token_type`; 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.

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. Most users only need this when their provider specifically supports or requires PAR.

par_required
: Logical. Whether the provider requires authorization requests to be sent via PAR. When `TRUE`, `par_url` must also be configured.

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"`.

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.

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.

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 the current OIDC-compatible shape 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.

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.

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.

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.

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.

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.

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
# See `?oauth_provider_microsoft` for example using a custom tenant ID

# 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()) {
  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

S7 class representing OAuth tokens and (optionally) user information.

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
)
```

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. `Inf` for non-expiring tokens
`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 introspect_elements = "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.

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.

Examples

```
# Please note: `get_userinfo()`, `introspect_token()`, and `refresh_token()`
# are typically not called by users of this package directly, but are called
# internally by `oauth_module_server()`. These functions are exported
# nonetheless for advanced use cases. Most users will not need to
# call these functions directly

# Example requires a real token from a completed OAuth flow
# (code is therefore not run; would error with placeholder values below)
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"
  )

  # Have a valid OAuthToken object; fake example below
  # (typically provided by `oauth_module_server()` or `handle_callback()`)
  token <- handle_callback(client, "<code>", "<payload>", "<browser_token>")
```

```
  # Get userinfo
  user_info <- get_userinfo(client, token)

  # Introspect token (if supported by provider)
  introspection <- introspect_token(client, token)

  # Refresh token
  new_token <- refresh_token(client, token, introspect = TRUE)
}
```

`perform_client_bearer_req` *Alias for* `perform_resource_req()`

Description

[Deprecated]

Deprecated alias for `perform_resource_req()` to avoid a breaking change in the public API. 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,
  oauth_client = NULL,
  token_type = NULL,
  dpop_nonce = NULL,
  idempotent = 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.
`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.

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. If FALSE (the default), HTTP redirects are disabled to prevent leaking the access token to unexpected hosts. 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. Set to FALSE only if you have already validated the URL and understand the security implications.

oauth_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 — Optional logical controlling generic transport and transient-HTTP retries in req_with_retry(). When NULL (the default), shinyOAuth infers this from the final request method using standard HTTP idempotency semantics (GET, HEAD, OPTIONS, TRACE, PUT, DELETE). DPoP nonce challenges are replayed once regardless, as required by RFC 9449.

Value

Same value as perform_resource_req().

---

perform_resource_req *Build and perform an authenticated httr2 request for a protected resource*

Description

This is a helper for calling downstream APIs with an access token. It creates an `httr2::request()` for the given URL, attaches the right authorization header for the token type, applies shinyOAuth's standard HTTP defaults, and performs the request. You can also provide a prebuilt `httr2::request()` object as the `url` argument, in which case this helper will layer token authentication and any explicit overrides on top of the provided request before performing it.

Use `resource_req()` if you want to only build the request (and perform it later).

Compared to `httr2::req_perform()`, this helper adds shinyOAuth-specific handling for DPoP-bound tokens, including retrying once with a fresh proof when a `DPoP-Nonce` challenge is encountered. For non-DPoP tokens, this helper behaves similarly to `httr2::req_perform()` but with the package's standard defaults for retries and redirects.

Usage

```
perform_resource_req(
  token,
  url,
  method = "GET",
  headers = NULL,
  query = NULL,
  follow_redirect = FALSE,
  check_url = TRUE,
  oauth_client = NULL,
  token_type = NULL,
  dpop_nonce = NULL,
  idempotent = 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.
`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.
`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. If `FALSE` (the default), HTTP redirects are disabled to prevent leaking the access token to unexpected hosts. 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. Set to FALSE only if you have already validated the URL and understand the security implications.

oauth_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 — Optional logical controlling generic transport and transient-HTTP retries in req_with_retry(). When NULL (the default), shinyOAuth infers this from the final request method using standard HTTP idempotency semantics (GET, HEAD, OPTIONS, TRACE, PUT, DELETE). DPoP nonce challenges are replayed once regardless, as required by RFC 9449.

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()) {
  # Get an OAuthToken
  # (typically provided as reactive return value by `oauth_module_server()`)
  token <- OAuthToken()

  # 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)
```

```
  )

  httr2::req_dry_run(request)

  # 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)
}
```

`prepare_call` *Prepare a OAuth 2.0 authorization call and build an authorization URL*

Description

Prepares an OAuth 2.0 authorization request and returns the browser redirect URL. It generates the needed state, PKCE, and nonce values, stores the one-time callback data, and builds the final authorization URL.

Usage

```
prepare_call(oauth_client, browser_token, request_uri_publisher = NULL)
```

Arguments

`oauth_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 request-object URL.

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

```
# Please note: `prepare_call()` & `handle_callback()` are typically
# not called by users of this package directly, but are called
# internally by `oauth_module_server()`. These functions are exported
# nonetheless for advanced use cases. Most users will not need to
# call these functions directly

# Below code shows generic usage of `prepare_call()` and `handle_callback()`
# (code is not run because it would require user interaction)
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 authorization URL and and store state in client's state store
  # `<browser_token>` is a token that identifies the browser session
  #  and would typically be stored in a browser cookie
  #  (`oauth_module_server()` handles this typically)
  authorization_url <- prepare_call(client, "<browser_token>")

  # Redirect user to authorization URL; retrieve code & payload from query;
  # read also `<browser_token>` from browser cookie
  # (`oauth_module_server()` handles this typically)
  code <- "..."
  payload <- "..."
  browser_token <- "..."

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

`refresh_token` *Refresh an OAuth 2.0 token*

Description

Refreshes an OAuth session by obtaining a new access token with the refresh token. When configured, shinyOAuth also re-fetches userinfo and validates any new ID token returned by the provider.

Per OIDC Core Section 12.2, providers may omit the ID token from refresh responses. When omitted, the original ID token from the initial login is preserved.

If the provider does return a new ID token during refresh, `refresh_token()` requires that an original ID token from the initial login is available so it can enforce subject continuity (OIDC 12.2: `sub` MUST match). If no original ID token is available, refresh fails with an error.

When `id_token_validation = TRUE`, any refresh-returned ID token is also fully validated (signature and claims) in addition to the OIDC 12.2 `sub` continuity check.

When `userinfo_required = TRUE`, userinfo is re-fetched using the fresh access token. Whenever shinyOAuth has both refreshed userinfo and a validated ID token baseline, it checks that their `sub` claims still match. If `userinfo_id_token_match = TRUE`, the absence of a trustworthy ID token baseline is treated as an error instead of silently accepting unbound userinfo data.

Usage

```
refresh_token(
  oauth_client,
  token,
  async = FALSE,
  introspect = FALSE,
  shiny_session = NULL
)
```

Arguments

`oauth_client`	OAuthClient object
`token`	OAuthToken object containing the refresh token
`async`	Logical, default FALSE. If TRUE and an async backend is configured, the refresh is dispatched through shinyOAuth's async promise path and this function returns a promise-compatible async result that resolves to an updated `OAuthToken`. mirai is preferred when daemons are configured via `mirai::daemons()`; otherwise the current future plan is used. Non-sequential future plans run off the main R session; `future::sequential()` stays in-process.
`introspect`	Logical, default FALSE. After a successful refresh, if the provider exposes an introspection endpoint, introspect the new access token for validation and audit/diagnostics. When enabled, refresh fails if introspection is unsupported, inactive, or missing required `introspect_elements`. The raw introspection result is not stored separately, but a successful introspection response may backfill `token@cnf`.
`shiny_session`	Optional pre-captured Shiny session context (from `capture_shiny_session_context()`) to include in audit events. Used when calling from async workers that lack access to the reactive domain.

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 finite fallback expiry from `resolve_missing_expires_in()`
- `refresh_token`: Updated if the provider rotates it; otherwise preserved

- `id_token`: Updated only if the provider returns one (and it validates); otherwise the original from login is preserved
- `userinfo`: Refreshed if `userinfo_required = TRUE`; otherwise preserved
- `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`.

Examples

```
# Please note: `get_userinfo()`, `introspect_token()`, and `refresh_token()`
# are typically not called by users of this package directly, but are called
# internally by `oauth_module_server()`. These functions are exported
# nonetheless for advanced use cases. Most users will not need to
# call these functions directly

# Example requires a real token from a completed OAuth flow
# (code is therefore not run; would error with placeholder values below)
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"
  )

  # Have a valid OAuthToken object; fake example below
  # (typically provided by `oauth_module_server()` or `handle_callback()`)
  token <- handle_callback(client, "<code>", "<payload>", "<browser_token>")

  # Get userinfo
  user_info <- get_userinfo(client, token)

  # Introspect token (if supported by provider)
  introspection <- introspect_token(client, token)

  # Refresh token
  new_token <- refresh_token(client, token, introspect = TRUE)
}
```

`resource_req` *Build an authenticated httr2 request for a protected resource*

Description

This is a helper for calling downstream APIs with an access token. It creates an `httr2::request()` for the given URL, attaches the right authorization header for the token type, and applies shinyOAuth's standard HTTP defaults.

Use `perform_resource_req()` when you want shinyOAuth to also perform the request and handle DPoP nonce challenges for you (which `httr2::req_perform()` would not do on its own).

Usage

```
resource_req(
  token,
  url,
  method = "GET",
  headers = NULL,
  query = NULL,
  follow_redirect = FALSE,
  check_url = TRUE,
  oauth_client = NULL,
  token_type = NULL,
  dpop_nonce = 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.
`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. If `FALSE` (the default), HTTP redirects are disabled to prevent leaking the access token to unexpected hosts. 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. Set to FALSE only if you have already validated the URL and understand the security implications.

oauth_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.

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). Adding query parameters after resource_req() is fine, but 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()) {
  # Get an OAuthToken
  # (typically provided as reactive return value by `oauth_module_server()`)
  token <- OAuthToken()

  # 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)
```

```
  )

  httr2::req_dry_run(request)

  # 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)
}
```

`revoke_token` *Revoke an OAuth 2.0 token*

Description

Attempts to revoke an access or refresh token when the provider exposes a revocation endpoint (RFC 7009).

Authentication mirrors the provider's `token_auth_style` (same as token exchange and introspection).

Best-effort semantics:

- If the provider does not expose a revocation endpoint, returns `supported = FALSE`, `revoked = NA`, and `status = "revocation_unsupported"`.
- If the selected token value is missing, returns `supported = TRUE`, `revoked = NA`, and `status = "missing_token"`.
- If the endpoint returns a 2xx, returns `supported = TRUE`, `revoked = TRUE`, and `status = "ok"`.
- If the endpoint returns an HTTP error, returns `supported = TRUE`, `revoked = NA`, and `status = "http_<code>"`.

Usage

```
revoke_token(
  oauth_client,
  oauth_token,
  which = c("refresh", "access"),
  async = FALSE,
  shiny_session = NULL
)
```

Arguments

`oauth_client`	OAuthClient object
`oauth_token`	OAuthToken object containing tokens to revoke

which	Which token to revoke: "refresh" (default) or "access"
async	Logical, default FALSE. If TRUE and an async backend is configured, the operation is dispatched through shinyOAuth's async promise path and this function returns a promise-compatible async result that resolves to the result list. mirai is preferred when daemons are configured via `mirai::daemons()`; otherwise the current future plan is used. Non-sequential future plans run off the main R session; `future::sequential()` stays in-process.
shiny_session	Optional pre-captured Shiny session context (from `capture_shiny_session_context()`) to include in audit events. Used when calling from async workers that lack access to the reactive domain.

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>"`.

---

use_shinyOAuth *Add JavaScript dependency to the UI of a Shiny app*

Description

Adds shinyOAuth's client-side JavaScript dependency to your Shiny UI. This is required so the module can handle redirects and manage its browser-side session token.

Without this call in the UI, `oauth_module_server()` will not work unless your app UI is wrapped with `oauth_form_post_ui()`, which injects this dependency automatically for form_post flows.

Usage

```
use_shinyOAuth(inject_referrer_meta = TRUE)
```

Arguments

inject_referrer_meta	If TRUE (default), injects a `<meta name="referrer" content="no-referrer">` tag into the document head. This reduces the risk of leaking OAuth callback query parameters (like `code` and `state`) via the `Referer` header to third-party subresources during the initial callback page load.

Details

Place this near the top-level of your UI (e.g., inside `fluidPage()` or `tagList()`), similar to how you would use `shinyjs::useShinyjs()`. If you wrap the app UI with `oauth_form_post_ui()`, you usually do not need a separate call here because that wrapper injects this dependency for you.

Value

A `tagList` that loads the `inst/www/shinyOAuth.js` dependency once.

See Also

`oauth_module_server()`

Examples

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

Index