Package ‘shinyOAuth’

November 10, 2025

Title Provider-Agnostic OAuth Authentication for 'shiny' Applications

Version 0.1.3

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

RoxygenNote 7.3.3

Imports S7 (>= 0.2.0), R6 (>= 2.0), rlang (>= 1.0.0), shiny (>= 1.7.0), jsonlite (>= 1.0), openssl (>= 2.0.0), httr2 (>= 1.0.0), cachem (>= 1.1.0), jose (>= 1.2.0), cli (>= 3.0.0), htmltools (>= 0.5.0)

Suggests testthat (>= 3.0.0), knitr, rmarkdown, webfakes, promises, future, withr, later, sodium, shinytest2, xml2

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

NeedsCompilation no

Author Luka Koning [aut, cre, cph]

Maintainer Luka Koning <koningluka@gmail.com>

Repository CRAN

Date/Publication 2025-11-10 13:20:02 UTC

Contents

client_bearer_req 2
custom_cache 3
error_on_softened 5
get_userinfo 6
handle_callback 7
introspect_token 9
is_ok_host 11
OAuthClient 12
OAuthProvider 17
OAuthToken 23
oauth_client 24
oauth_module_server 29
oauth_provider 37
oauth_provider_auth0 42
oauth_provider_github 44
oauth_provider_google 46
oauth_provider_keycloak 47
oauth_provider_microsoft 49
oauth_provider_oidc 52
oauth_provider_oidc_discover 54
oauth_provider_okta 58
oauth_provider_slack 59
oauth_provider_spotify 61
prepare_call 63
refresh_token 64
use_shinyOAuth 66

Index **67**

`client_bearer_req` *Build an authorized httr2 request with Bearer token*

Description

Convenience helper to reduce boilerplate when calling downstream APIs. It creates an `httr2::request()` for the given URL, attaches the `Authorization: Bearer <token>` header, and applies the package's standard HTTP defaults (timeout and User-Agent).

Accepts either a raw access token string or an OAuthToken object.

Usage

```
client_bearer_req(token, url, method = "GET", headers = NULL, query = 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".
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` header is ignored to ensure the Bearer token set by this function is not overridden.
query	Optional named list of query parameters to append to the URL.

Value

An httr2 request object, ready to be further customized or performed with `httr2::req_perform()`.

Examples

```
# Make request using OAuthToken object
# (code is not run because it requires a real token from user interaction)
## Not run:
# Get an OAuthToken
# (typically provided as reactive return value by `oauth_module_server()`)
token <- OAuthToken()

# Build request
request <- client_bearer_req(
  token,
  "https://api.example.com/resource",
  query = list(limit = 5)
)

# Perform request
response <- httr2::req_perform(request)

## End(Not run)
```

---

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

Description

Builds a minimal cachem-like cache backend object that exposes cachem-compatible methods: `$get(key, missing)`, `$set(key, value)`, `$remove(key)`, and `$info()`.

Use this helper when you want to plug a custom state store or JWKS cache into 'shinyOAuth', when `cachem::cache_mem()` or `cachem::cache_disk()` are not suitable. This may be useful specifically when you deploy a Shiny app to a multi-process environment 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`)

The `$info()` method is optional, but if provided and it returns a list with `max_age` (seconds), shinyOAuth will align cookie/issued_at TTLs to that value.

Usage

```
custom_cache(get, set, remove, 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 mandatory because both `OAuthClient` and `OAuthProvider` validators will pass it explicitly.
`set`	A function(key, value) -> invisible(NULL). Required. Should store the value under the given key
`remove`	A function(key) -> logical or sentinel. Required. For state stores, this enforces single-use eviction. If your backend performs an atomic "get-and-delete" (e.g., SQL DELETE .. RETURNING), you may supply a function which does nothing here but returns `TRUE`. (The login flow will always attempt to call `$remove()` after `$get()` as a best-effort cleanup.) Recommended contract for interoperability and strong replay protection: • Return `TRUE` when a key was actually deleted or if it already did not exist • Return `FALSE` when they key could not be deleted or when it is unknown if they key was deleted When the return value is not `TRUE`, 'shinyOAuth' will attempt to retrieve the value from the state store to check if it may still be present; if that fails (i.e., key is not present), it will treat the removal as succesful. If it does find the key, it will produce an error indicating that removal did not succeed.
`info`	Function() -> list(max_age = seconds, ...). Optional. This may be provided to because 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)
      return(TRUE) # signal successful deletion
    }
    return(TRUE) # key did not exist
  },

  info = function() list(max_age = 600)
)
```

error_on_softened *Throw an error if any safety checks have been disabled*

Description

This function checks if any safety checks have been disabled via options intended for local development use only. If any such options are detected, an error is thrown to prevent accidental use in production environments.

Usage

```
error_on_softened()
```

Details

It checks for the following options:

- `shinyOAuth.skip_browser_token`: Skips browser cookie presence check
- `shinyOAuth.skip_id_sig`: Skips ID token signature verification
- `shinyOAuth.print_errors`: Enables printing of error messages
- `shinyOAuth.print_traceback`: Enables printing of tracebacks (opt-in only; default FALSE)
- `shinyOAuth.expose_error_body`: Exposes HTTP response bodies

Note: Tracebacks are only treated as a "softened" behavior when the `shinyOAuth.print_traceback` option is explicitly set to `TRUE`. The default is `FALSE`, even in interactive or test sessions.

Value

Invisible TRUE if no safety checks are disabled; otherwise, an error is thrown.

Examples

```
# Throw an error if any developer-only softening options are enabled
# Below call does not error if run with default options:
error_on_softened()

# Below call would error (is therefore not run):
## Not run:
options(shinyOAuth.skip_id_sig = TRUE)
error_on_softened()

## End(Not run)
```

get_userinfo *Get user info from OAuth 2.0 provider*

Description

Fetches user information from the provider's userinfo endpoint using the provided access token. Emits an audit event with redacted details.

Usage

```
get_userinfo(oauth_client, token)
```

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.

Value

A list containing the user information as 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)
## Not run:
# 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)

## End(Not run)
```

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

Description

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

Usage

```
handle_callback(
  oauth_client,
  code,
  payload,
  browser_token,
  decrypted_payload = NULL,
  state_store_values = NULL
)
```

Arguments

`oauth_client`	An OAuthClient object representing the OAuth client configuration.
`code`	The authorization code received from the OAuth provider during the callback.
`payload`	The encrypted state payload received from the OAuth provider during the callback (this should be the same value that was generated and sent in `prepare_call()`).
`browser_token`	Browser token present in the user's session (this is managed by `oauth_module_server()` and should match the one used in `prepare_call()`).

`decrypted_payload`
: Optional pre-decrypted and validated payload list (as returned by `state_decrypt_gcm()` followed by internal validation). Supplying this allows callers to validate and bind the state on the main thread before dispatching to a background worker for async flows.

`state_store_values`
: Optional pre-fetched state store entry (a list with `browser_token`, `pkce_code_verifier`, and nonce). When supplied, the function will skip reading/removing from `oauth_client@state_store` and use the provided values instead. This supports async flows that prefetch and remove the single-use state entry on the main thread to avoid cross-process cache visibility issues.

Value

An OAuthToken` object containing the access token, refresh token, expiration time, user information (if requested), and ID token (if applicable). If any step of the process fails (e.g., state verification, token exchange, token validation), an error is thrown indicating the failure reason.

Examples

```
# Please note: `prepare_callback()` & `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_callback()` and `handle_callback()`
# (code is not run because it would require user interaction)
## Not run:
# 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_callback(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)

## End(Not run)
```

`introspect_token` *Introspect an OAuth 2.0 token*

Description

Introspects an access or refresh token using RFC 7662 when the provider exposes an introspection endpoint. Returns a list including at least `supported` (logical) and `active` (logical|NA) and the parsed response (if any) under `raw`.

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`.
- "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` set to the provider's `introspection_url`.

Usage

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

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 promises is available, run in background and return a promise resolving to the result list

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"`). 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, active, raw, status

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)
## Not run:
# 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)

## End(Not run)
```

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

Description

Returns TRUE if every input URL is 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 thus an allowlist of hosts/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 scheme rules), but HTTPS is still required unless the host is in `allowed_non_https_hosts`.

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 be used elsewhere to test if URLs would be allowed. Internally, it will always determine the default values for `allowed_non_https_hosts` and `allowed_hosts` from the options `shinyOAuth.allowed_non_https_hosts` and `shinyOAuth.allowed_hosts`, respectively.

Value

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

Examples

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

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

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

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

`OAuthClient` *OAuthClient S7 class*

Description

S7 class representing an OAuth 2.0 client configuration, including a provider, client credentials, redirect URI, requested scopes, and state management.

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),
  client_private_key = NULL,
  client_private_key_kid = NA_character_,
  client_assertion_alg = NA_character_,
  redirect_uri = character(0),
  scopes = character(0),
  state_store = cachem::cache_mem(max_age = 300),
  state_entropy = 64,
  state_key = random_urlsafe(n = 128)
)
```

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 for public PKCE-only clients when the provider is configured with `use_pkce = TRUE` and uses form-body client authentication at the token endpoint (`token_auth_style = "body"`, also known as `client_secret_post`). In this case, the secret is omitted from token requests. Note: If your provider issues HS256 ID tokens and `id_token_validation` is enabled, a non-empty `client_secret` is required for signature validation.
`client_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.
`client_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, `ES256`/`ES384`/`ES512` for EC P-256/384/521, or `EdDSA` for Ed25519/Ed448). If an explicit value is provided but incompatible with the key, validation fails early with a configuration error. Supported values are `HS256`, `HS384`, `HS512` for client_secret_jwt and asymmetric algorithms supported by `jose::jwt_encode_sig` (e.g., `RS256`, `PS256`, `ES256`, `EdDSA`) for private keys.
`redirect_uri`	Redirect URI registered with provider
`scopes`	Vector of scopes to request
`state_store`	State storage backend. Defaults to `cachem::cache_mem(max_age = 300)`. Alternative backends could include `cachem::cache_disk()` or a custom implementation (which you can create with `custom_cache()`. The backend must implement cachem-like methods `$get(key, missing)`, `$set(key, value)`, and `$remove(key)`; `$info()` is optional. Trade-offs: `cache_mem` is in-memory and thus scoped to a single R process (good default for a single Shiny process). `cache_disk` persists to disk and can be shared across multiple R processes (useful for multi-process deployments

or when Shiny workers aren't sticky). A `custom_cache()` backend could use a database or external store (e.g., Redis, Memcached). See also `vignette("usage", package = "shinyOAuth")`.

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_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, which provides approximately 384 bits of entropy
`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, you must configure a shared `state_store` and the same `state_key` across all workers. Otherwise callbacks that land on a different worker will be unable to decrypt/validate the state envelope and authentication will fail. In such environments, do not rely on the random per-process default: provide an explicit, high-entropy key (for example via a secret store or environment variable). Prefer values with substantial entropy (e.g., 64–128 base64url characters or a raw 32+ byte key). Avoid human-memorable passphrases. See also `vignette("usage", package = "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... "
    )
  )
}


# 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)
}


# 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)
}


# 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)
          req <- client_bearer_req(auth$token, "https://api.github.com/user/repos")
          resp <- httr2::req_perform(req)

          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)
      }
    }
```

`OAuthProvider` *OAuthProvider S7 class*

Description

S7 class representing an OAuth 2.0 provider configuration. Includes endpoints, OIDC settings, and various security options which govern the OAuth and OIDC flows.

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),
  userinfo_url = NA_character_,
  introspection_url = NA_character_,
  issuer = NA_character_,
  use_nonce = FALSE,
  use_pkce = TRUE,
  pkce_method = "S256",
  userinfo_required = FALSE,
  userinfo_id_selector = function(userinfo) userinfo$sub,
  userinfo_id_token_match = FALSE,
  id_token_required = FALSE,
  id_token_validation = FALSE,
  extra_auth_params = list(),
  extra_token_params = list(),
  extra_token_headers = character(0),
  token_auth_style = "header",
  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", "PS256", "PS384", "PS512", "ES256",
    "ES384", "ES512", "EdDSA"),
  allowed_token_types = character(0),
  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
`userinfo_url`	User info endpoint URL (optional)

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

`issuer`
: OIDC issuer URL (optional; required for ID token validation). This is the base URL that identifies the OpenID Provider (OP). It is used during ID token validation to verify the `iss` claim in the ID token matches the expected issuer. It is also used to fetch the provider's JSON Web Key Set (JWKS) for verifying ID token signatures (typically via the OIDC discovery document located at `/.well-known/openid-configuration` relative to the issuer URL)

`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. This is recommended for OIDC flows to mitigate replay attacks

`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 is prevents authorization code interception attacks

`pkce_method`
: PKCE code challenge method ("S256" or "plain"). "S256" is recommended. "plain" should only be used for non-compliant providers that do not support "S256"

`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 the userinfo fails, the token exchange will fail.

  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 when `userinfo_id_token_match` is TRUE. Optional otherwise; when not supplied, some features (like subject matching) will be unavailable. Helper constructors like `oauth_provider()` and `oauth_provider_oidc()` provide a default selector that extracts the `sub` field.

`userinfo_id_token_match`
: Whether to verify that the user ID ("sub") from the ID token matches the user ID extracted from the userinfo response. This requires both `userinfo_required` and `id_token_validation` to be TRUE (and thus a valid `userinfo_url` and `issuer` to be set, plus potentially setting the client's scope to include "openid", so that an ID token is returned). Furthermore, the provider's `userinfo_id_selector` must be configured to extract the user ID from the userinfo response. This check helps ensure the integrity of the user information by confirming that both sources agree on the user's identity.

  For `oauth_provider()`, when not explicitly supplied, this is inferred as `TRUE` only if both `userinfo_required` and `id_token_validation` are `TRUE`; otherwise it defaults to `FALSE`.

`id_token_required`
: Whether to require an ID token to be returned during token exchange. If no ID token is returned, the token exchange will fail. This requires the provider to be a valid OpenID Connect provider and may require setting 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.

`extra_auth_params`
: Extra parameters for authorization URL

`extra_token_params`
: Extra parameters for token exchange

`extra_token_headers`
: Extra headers for token exchange requests (named character vector)

`token_auth_style`
: How to authenticate when exchanging tokens. One of:

  - "header": HTTP Basic (client_secret_basic)
  - "body": Form body (client_secret_post)
  - "client_secret_jwt": JWT client assertion signed with HMAC using client_secret (RFC 7523)
  - "private_key_jwt": JWT client assertion signed with an asymmetric key (RFC 7523)

`jwks_cache`
: JWKS cache backend. If not provided, a `cachem::cache_mem(max_age = 3600)` (1 hour) cache will be created. May be any cachem-compatible backend, including `cachem::cache_disk()` for a filesystem cache shared across workers, or a custom implementation created via `custom_cache()` (e.g., database/Redis backed).

  TTL guidance: Choose `max_age` in line with your identity platform's JWKS rotation and cache-control cadence. A range of 15 minutes to 2 hours is typically sensible; the default is 1 hour. Shorter TTLs adopt new keys faster at the cost of more JWKS traffic; longer TTLs reduce traffic but may delay new keys slightly. Signature verification will automatically perform a one-time JWKS refresh when a new `kid` appears in an ID token.

  Cache keys are internal, hashed by issuer and pinning configuration. Cache values are lists with elements `jwks` and `fetched_at` (numeric epoch seconds)

`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`. Use to reduce key substitution risks by pre-authorizing 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 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 (or a subdomain). 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 (e.g., 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. 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`, `PS256`, `PS384`, `PS512`, `ES256`, `ES384`, `ES512`, and `EdDSA` (Ed25519/Ed448 via OKP). 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","PS256","PS384","PS512", "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. Prefer rotating secrets regularly when enabling this.

`allowed_token_types`
: Character vector of acceptable OAuth token types returned by the token endpoint (case-insensitive). When non-empty, the token response MUST include `token_type` and it must be one of the allowed values; otherwise the flow fails fast with a `shinyOAuth_token_error`. When empty, no check is performed and `token_type` may be omitted by the provider. Helper constructors default this more strictly: for `oauth_provider()` when an `issuer` is supplied or OIDC flags are enabled, `allowed_token_types` defaults to `c("Bearer")` to enforce Bearer by default; otherwise it remains empty. You can override to widen or disable enforcement by setting it explicitly

`leeway`
: Clock skew leeway (seconds) applied to ID token `exp`/`iat` checks. 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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

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

```
## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

`OAuthToken` *OAuthToken S7 class*

Description

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

Usage

```
OAuthToken(
  access_token = character(0),
  refresh_token = NA_character_,
  id_token = NA_character_,
  expires_at = Inf,
  userinfo = list()
)
```

Arguments

`access_token`	Access token
`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)

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)
## Not run:
```

```
# 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)

## End(Not run)
```

oauth_client — *Create generic OAuthClient*

Description

Create generic OAuthClient

Usage

```
oauth_client(
  provider,
  client_id = Sys.getenv("OAUTH_CLIENT_ID"),
  client_secret = Sys.getenv("OAUTH_CLIENT_SECRET"),
  redirect_uri,
  scopes = character(0),
  state_store = cachem::cache_mem(max_age = 300),
  state_entropy = 64,
  state_key = random_urlsafe(128),
  client_private_key = NULL,
  client_private_key_kid = NULL,
  client_assertion_alg = NULL
)
```

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 for public PKCE-only clients when the provider is configured with `use_pkce = TRUE` and uses form-body client authentication at the token endpoint (`token_auth_style = "body"`, also known as `client_secret_post`). In this case, the secret is omitted from token requests.

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

`state_store` State storage backend. Defaults to `cachem::cache_mem(max_age = 300)`. Alternative backends could include `cachem::cache_disk()` or a custom implementation (which you can create with `custom_cache()`. The backend must implement cachem-like methods `$get(key, missing)`, `$set(key, value)`, and `$remove(key)`; `$info()` is optional.

Trade-offs: `cache_mem` is in-memory and thus scoped to a single R process (good default for a single Shiny process). `cache_disk` persists to disk and can be shared across multiple R processes (useful for multi-process deployments or when Shiny workers aren't sticky). A `custom_cache()` backend could use a database or external store (e.g., Redis, Memcached). See also `vignette("usage", package = "shinyOAuth")`.

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_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, which provides approximately 384 bits of entropy

`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, you must configure a shared `state_store` and the same `state_key` across all workers. Otherwise callbacks that land on a different worker will be unable to decrypt/validate the state envelope and authentication will fail. In such environments, do not rely on the random per-process default: provide an explicit, high-entropy key (for example via a secret store or environment variable). Prefer values with substantial entropy (e.g., 64–128 base64url characters or a raw 32+ byte key). Avoid human-memorable passphrases. See also `vignette("usage", package = "shinyOAuth")`.

`client_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.

`client_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, `ES256`/`ES384`/`ES512` for EC P-256/384/521, or `EdDSA` for Ed25519/Ed448). If an explicit value is provided but incompatible with the key, validation fails early with a configuration error. Supported values are `HS256`, `HS384`, `HS512` for client_secret_jwt and asymmetric algorithms supported by `jose::jwt_encode_sig` (e.g., `RS256`, `PS256`, `ES256`, `EdDSA`) for private keys.

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... "
      )
    )
  }


  # 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)
  }


  # 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)
}


# 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)
      req <- client_bearer_req(auth$token, "https://api.github.com/user/repos")
      resp <- httr2::req_perform(req)

      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)
  }
}
```

`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,
  tab_title_cleaning = TRUE,
  tab_title_replacement = 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, performs token exchange and refresh in the background using the promises package (future_promise), and updates values when the promise resolves. Requires the promises::promises package and a suitable backend to be configured with `future::plan()`. If FALSE (default), token exchange and refresh are performed synchronously (which may block the Shiny event loop; it is thus strongly recommended to set `async = TRUE` in production apps)
`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/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

`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`

`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. 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

- 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` so network calls run in a background future via the promises package (configure a multisession/multicore backend), 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 and the state payload's `issued_at` validation window.
- Watchdog for missing browser token: to catch misconfiguration early during development, the module includes a short watchdog. If the browser token cookie is not set within 1500ms of module initialization, a warning is emitted to the R console. This likely means you forgot to include `use_shinyOAuth()` in your UI, but it may also indicate that a user of your app is using a browser with JavaScript disabled. The watchdog prints a warning only once per R session, but if you want to suppress it permanently, you can set `options(shinyOAuth.disable_watchdog_warning = TRUE)`.

Value

A reactiveValues object with `token`, `error`, `error_description`, 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.
- `token`: OAuthToken object, or NULL if not yet authenticated. This contains the access token, refresh token (if any), ID token (if any), and userinfo (if fetched). See OAuthToken for details. Note that since OAuthToken is a S7 object, you access its fields with `@`, e.g., `token@userinfo`.
- `error`: error code string when the OAuth flow fails. Be careful with exposing this directly to users, as it may contain sensitive information which could aid an attacker.
- `error_description`: human-readable error detail when available. Be extra careful with exposing this directly to users, as it may contain even more sensitive information which could aid an attacker.

- `browser_token`: internal opaque browser cookie value; used for state double-submit protection; NULL if not yet set
- `pending_callback`: internal list(code, state); used to defer token exchange 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 implement a manual login flow (e.g., with your own 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. This is the main entry point for login when `auto_redirect = FALSE` and you want to trigger login from your own UI
- `logout()`: clears the current token setting `authenticated` to FALSE, and clears 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 (verify with `has_browser_token()` first). 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... "
      )
    )
  }


  # 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)
  }


  # 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)
}


# 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)
    req <- client_bearer_req(auth$token, "https://api.github.com/user/repos")
    resp <- httr2::req_perform(req)

    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)
}
}
```

`oauth_provider` *Create generic OAuthProvider*

Description

Helper function to create an OAuthProvider object. This function provides sensible defaults and infers some settings based on the provided parameters.

Usage

```
oauth_provider(
  name,
  auth_url,
  token_url,
  userinfo_url = NA_character_,
  introspection_url = NA_character_,
  issuer = NA_character_,
  use_nonce = NULL,
  use_pkce = TRUE,
  pkce_method = "S256",
  userinfo_required = NULL,
  userinfo_id_token_match = NULL,
  userinfo_id_selector = function(userinfo) userinfo$sub,
  id_token_required = NULL,
  id_token_validation = NULL,
  extra_auth_params = list(),
  extra_token_params = list(),
  extra_token_headers = character(),
  token_auth_style = "header",
  jwks_cache = NULL,
  jwks_pins = character(),
  jwks_pin_mode = "any",
  jwks_host_issuer_match = NULL,
  jwks_host_allow_only = NULL,
 allowed_algs = c("RS256", "RS384", "RS512", "PS256", "PS384", "PS512", "ES256",
    "ES384", "ES512", "EdDSA"),
  allowed_token_types = NULL,
  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
`userinfo_url`	User info endpoint URL (optional)
`introspection_url`	Token introspection endpoint URL (optional; RFC 7662)
`issuer`	OIDC issuer URL (optional; required for ID token validation). This is the base URL that identifies the OpenID Provider (OP). It is used during ID token validation to verify the `iss` claim in the ID token matches the expected issuer. It is also used to fetch the provider's JSON Web Key Set (JWKS) for verifying ID token signatures (typically via the OIDC discovery document located at `/.well-known/openid-configuration` relative to the issuer URL)
`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. This is recommended for OIDC flows to mitigate replay attacks
`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 is prevents authorization code interception attacks
`pkce_method`	PKCE code challenge method ("S256" or "plain"). "S256" is recommended. "plain" should only be used for non-compliant providers that do not support "S256"
`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 the userinfo fails, the token exchange will fail. 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_token_match`
: Whether to verify that the user ID ("sub") from the ID token matches the user ID extracted from the userinfo response. This requires both `userinfo_required` and `id_token_validation` to be TRUE (and thus a valid `userinfo_url` and `issuer` to be set, plus potentially setting the client's scope to include "openid", so that an ID token is returned). Furthermore, the provider's `userinfo_id_selector` must be configured to extract the user ID from the userinfo response. This check helps ensure the integrity of the user information by confirming that both sources agree on the user's identity.

  For `oauth_provider()`, when not explicitly supplied, this is inferred as `TRUE` only if both `userinfo_required` and `id_token_validation` are `TRUE`; otherwise it defaults to `FALSE`.

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

  This is used when `userinfo_id_token_match` is TRUE. Optional otherwise; when not supplied, some features (like subject matching) will be unavailable. Helper constructors like `oauth_provider()` and `oauth_provider_oidc()` provide a default selector that extracts the `sub` field.

`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 requires the provider to be a valid OpenID Connect provider and may require setting 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.

`extra_auth_params`
: Extra parameters for authorization URL

`extra_token_params`
: Extra parameters for token exchange

`extra_token_headers`
: Extra headers for token exchange requests (named character vector)

`token_auth_style`
: How to authenticate when exchanging tokens. One of:

  - "header": HTTP Basic (client_secret_basic)
  - "body": Form body (client_secret_post)

- "client_secret_jwt": JWT client assertion signed with HMAC using client_secret (RFC 7523)
- "private_key_jwt": JWT client assertion signed with an asymmetric key (RFC 7523)

`jwks_cache` JWKS cache backend. If not provided, a `cachem::cache_mem(max_age = 3600)` (1 hour) cache will be created. May be any cachem-compatible backend, including `cachem::cache_disk()` for a filesystem cache shared across workers, or a custom implementation created via `custom_cache()` (e.g., database/Redis backed).

TTL guidance: Choose `max_age` in line with your identity platform's JWKS rotation and cache-control cadence. A range of 15 minutes to 2 hours is typically sensible; the default is 1 hour. Shorter TTLs adopt new keys faster at the cost of more JWKS traffic; longer TTLs reduce traffic but may delay new keys slightly. Signature verification will automatically perform a one-time JWKS refresh when a new `kid` appears in an ID token.

Cache keys are internal, hashed by issuer and pinning configuration. Cache values are lists with elements `jwks` and `fetched_at` (numeric epoch seconds)

`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`. Use to reduce key substitution risks by pre-authorizing 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 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 (or a subdomain). 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 (e.g., 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. 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`, `PS256`, `PS384`, `PS512`, `ES256`, `ES384`, `ES512`, and `EdDSA` (Ed25519/Ed448 via OKP). 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","PS256","PS384","PS512", "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. Prefer rotating secrets regularly when enabling this.

`allowed_token_types`
: Character vector of acceptable OAuth token types returned by the token endpoint (case-insensitive). When non-empty, the token response MUST include `token_type` and it must be one of the allowed values; otherwise the flow fails fast with a `shinyOAuth_token_error`. When empty, no check is performed and `token_type` may be omitted by the provider. Helper constructors default this more strictly: for `oauth_provider()` when an `issuer` is supplied or OIDC flags are enabled, `allowed_token_types` defaults to `c("Bearer")` to enforce Bearer by default; otherwise it remains empty. You can override to widen or disable enforcement by setting it explicitly

`leeway`
: Clock skew leeway (seconds) applied to ID token `exp`/`iat` checks. Default 30. Can be globally overridden via option `shinyOAuth.leeway`

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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

`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 to request in auth flows

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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")
```

```
## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

`oauth_provider_github` *Create a GitHub OAuthProvider*

Description

Pre-configured OAuth 2.0 provider 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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

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

```
## End(Not run)
```

oauth_provider_google *Create a Google OAuthProvider*

Description

Pre-configured OAuthProvider 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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

`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"
)
```

Arguments

`base_url`	Base URL of the Keycloak server, e.g., "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), "private_key_jwt", or "client_secret_jwt". Defaults to "body" for Keycloak, which works for both confidential clients and public PKCE clients (secretless). If you pass `NULL`, discovery will infer the method from the provider's `token_endpoint_auth_methods_supported` metadata.

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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

oauth_provider_microsoft

Create a Microsoft (Entra ID) OAuthProvider

Description

Pre-configured OAuthProvider 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 (no discovery).

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, otherwise disabled

Details

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 will enable strict ID token validation (issuer match). When using the multi-tenant aliases ("common", "organizations", "consumers"), the exact issuer depends on the account that signs in and therefore ID token validation is disabled by default to avoid false negatives. You can override this via `id_token_validation` if you know the environment guarantees a fixed issuer.

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` '9391afd1-7129-4938-9e4d-633c688f93c0' (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",
    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 (str):\n",
        paste(capture.output(str(tok)), collapse = "\n")
      )
    })

    output$user_info <- renderPrint({
      req(auth$token)
      auth$token@userinfo
    })

    output$oauth_error <- renderUI({
      if (!is.null(auth$error)) {
        msg <- auth$error
        if (!is.null(auth$error_description)) {
          msg <- paste0(msg, ": ", auth$error_description)
        }
        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

Preconfigured OAuthProvider for OpenID Connect (OIDC) compliant providers.

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 (or a subdomain). 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)
# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

oauth_provider_oidc_discover

Discover and create an OpenID Connect (OIDC) OAuthProvider

Description

Uses the OpenID Connect discovery document at `/.well-known/openid-configuration` to auto-configure an OAuthProvider. When present, `introspection_endpoint` is wired into the resulting provider for RFC 7662 support.

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", "PS256", "PS384", "PS512", "ES256",
    "ES384", "ES512", "EdDSA"),
  allowed_token_types = c("Bearer"),
  jwks_host_issuer_match = TRUE,
  issuer_match = TRUE,
  ...
)
```

Arguments

`issuer`	The OIDC issuer base URL (including scheme), e.g., "https://login.example.com"
`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) or "body" (client_secret_post). If NULL (default), it is inferred conservatively from discovery. When PKCE is enabled and the provider advertises support for public clients via `none`, a secretless flow is preferred (modeled as `"body"` without credentials). 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*), RSA-PSS (PS*), 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 (or a subdomain). 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`	Logical, default TRUE. When TRUE, requires the discovery issuer's scheme/host to match the input `issuer`. When FALSE, host mismatch is allowed. Prefer tightening hosts via `options(shinyOAuth.allowed_hosts)` when feasible
`...`	Additional fields passed to `oauth_provider()`

Details

- ID token algorithms: by default this helper accepts common asymmetric algorithms RSA (RS*), RSA-PSS (PS*), 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 (with PKCE), as well as JWT-based methods `private_key_jwt` and `client_secret_jwt` per RFC 7523.

  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 only uses public none 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 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. 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_ho` (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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

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

```
## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

`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 (default "default")
`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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

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)

# 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)

slack_provider <- oauth_provider_slack()


# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

---

oauth_provider_spotify

Create a Spotify OAuthProvider

Description

Pre-configured OAuth 2.0 provider for Spotify. Uses /v1/me as "userinfo". No ID token (not OIDC).

Usage

```
oauth_provider_spotify(
  name = "spotify",
  scope = "user-read-email user-read-private"
)
```

Arguments

name	Optional provider name (default "spotify")
scope	Optional space-separated scope string (default "user-read-email user-read-private")

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)

# 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)

slack_provider <- oauth_provider_slack()
```

```
# Keycloak
# (requires configured Keycloak realm; example below is therefore not run)
## Not run:
oauth_provider_keycloak(base_url = "http://localhost:8080", realm = "myrealm")

## End(Not run)

# Auth0
# (requires configured Auth0 domain; example below is therefore not run)
## Not run:
oauth_provider_auth0(domain = "your-tenant.auth0.com")

## End(Not run)

# Okta
# (requires configured Okta domain; example below is therefore not run)
## Not run:
oauth_provider_okta(domain = "dev-123456.okta.com")

## End(Not run)
```

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

Description

This function prepares an OAuth 2.0 authorization call by generating necessary state, PKCE, and nonce values, storing them securely, and constructing the authorization URL to redirect the user to. The state and accompanying values are stored in the client's state store for later verification during the callback phase of the OAuth 2.0 flow.

Usage

```
prepare_call(oauth_client, browser_token)
```

Arguments

`oauth_client`	An OAuthClient object representing the OAuth client configuration.
`browser_token`	A string token (e.g., from a browser cookie) to identify the user/session.

Value

A string containing the constructed authorization URL. This URL should be used to redirect the user to the OAuth provider's authorization endpoint.

Examples

```
# Please note: `prepare_callback()` & `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_callback()` and `handle_callback()`
# (code is not run because it would require user interaction)
## Not run:
# 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_callback(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)

## End(Not run)
```

---

refresh_token *Refresh an OAuth 2.0 token*

Description

Refreshes an OAuth 2.0 access token using a refresh token.

Usage

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

Arguments

`oauth_client`	OAuthClient object
`token`	OAuthToken object containing the refresh token
`async`	Logical, default FALSE. If TRUE and the `promises` package is available, the refresh is performed off the main R session using `promises::future_promise()` and this function returns a promise that resolves to an updated `OAuthToken`. If `promises` is not available, falls back to synchronous behavior
`introspect`	Logical, default FALSE. After a successful refresh, if the provider exposes an introspection endpoint, perform a best-effort introspection of the new access token for audit/diagnostics. The result is not stored on the token object.

Value

An updated OAuthToken object with a new access token. If the provider issues a new refresh token, that replaces the old one. When the provider returns an ID token and `id_token_validation = TRUE`, it is validated. When `userinfo_required = TRUE`, fresh userinfo is fetched and stored on the token. `expires_at` is computed from `expires_in` when provided; otherwise set to `Inf`.

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)
## Not run:
# 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)
```

```
## End(Not run)
```

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

Description

Adds the package's client-side JavaScript helpers as an htmlDependency to your Shiny UI. This enables features such as redirection and setting the browser cookie token.

Without adding this to the UI of your app, the `oauth_module_server()` will not function.

Usage

```
use_shinyOAuth()
```

Details

Place this near the top-level of your UI (e.g., inside `fluidPage()` or `tagList()`), similar to how you would use `shinyjs::useShinyjs()`.

Value

A tagList containing a singleton dependency tag that ensures the JS file `inst/www/shinyOAuth.js` is loaded

See Also

`oauth_module_server()`

Examples

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

Index

cachem::cache_disk(), *3*, *20*, *40*
cachem::cache_mem(), *3*
client_bearer_req, 2
custom_cache, 3
custom_cache(), *13, 14*, *20*, *25*, *40*

error_on_softened, 5

future::plan(), *30*

get_userinfo, 6

handle_callback, 7
httr2::req_perform(), *3*
httr2::request(), *2*

introspect_token, 9
is_ok_host, 11

oauth_client, 24
oauth_client(), *12*
oauth_module_server, 29
oauth_module_server(), *66*
oauth_provider, 37
oauth_provider(), *18–21*, *38–41*, *53*, *56*
oauth_provider_auth0, 42
oauth_provider_github, 44
oauth_provider_google, 46
oauth_provider_keycloak, 47
oauth_provider_microsoft, 49
oauth_provider_oidc, 52
oauth_provider_oidc(), *18–21*, *39, 40*
oauth_provider_oidc_discover, 54
oauth_provider_oidc_discover(), *18*, *21*, *40*
oauth_provider_okta, 58
oauth_provider_slack, 59
oauth_provider_spotify, 61
OAuthClient, *6, 7*, *9*, 12, *24*, *26*, *30*, *46*, *50*, *56*, *63*, *65*
OAuthProvider, *6*, *13*, 17, *25*, *37*, *41–44*, *46–50*, *52–56*, *58–62*
OAuthToken, *2, 3*, *6*, *8, 9*, 23, *32*, *65*

prepare_call, 63
promises::promises, *30*

refresh_token, 64

use_shinyOAuth, 66
use_shinyOAuth(), *34*