Package 'fireproof'

May 8, 2026

Title Authentication and Authorization for 'fiery' Servers

Version 0.1.0

Description Provides a plugin for 'fiery' that supports various forms of authorization and authentication schemes. Schemes can be required in various combinations or by themselves and can be combined with scopes to provide fine-grained access control to the server.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

URL https://github.com/thomasp85/fireproof, https://thomasp85.github.io/fireproof/

BugReports https://github.com/thomasp85/fireproof/issues

Imports base64enc, cli, curl, fiery, firesale (>= 0.1.1), jose, jsonlite, R6, reqres (>= 1.1.0), rlang, routr (>= 1.0.0), sodium, urltools

Suggests callr, dplyr, quarto, rmarkdown, storr, testthat (>= 3.0.0), webfakes, xml2

Config/testthat/edition 3

VignetteBuilder quarto

NeedsCompilation no

Author Thomas Lin Pedersen [aut, cre] (ORCID: <https://orcid.org/0000-0002-5147-4711>), Posit, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Thomas Lin Pedersen <thomas.pedersen@posit.co>

Repository CRAN

Date/Publication 2025-12-17 10:10:26 UTC

Contents

`Fireproof` *A plugin that handles authentication and/or authorization*

Description

This plugin orchestrates all guards for your fiery app. It is a special Route that manages all the different guards you have defined as well as testing all the endpoints that have auth requirements.

Details

A guard is an object deriving from the Guard class which is usually created with one of the `guard_*()` constructors. You can provide it with a name as you register it and can thus have multiple instances of the same scheme (e.g. two `guard_basic()` with different user lists)

An auth handler is a handler that consists of a method, path, and flow. The flow is a logical expression of the various guards the request must pass to get access to that endpoint. For example, if you have two guards named `auth1` and `auth2`, a flow could be `auth1 || auth2` to allow a request if it passes either of the guards. Given an additional guard, `auth3`, it could also be something like `auth1 || (auth2 && auth3)`. The flow is given as a bare expression, not as a string. In addition to the three required arguments you can also supply a character vector of scopes that are required to have access to the endpoint. If your guard has scope support then the request will be tested against these to see if the (otherwise valid) user has permission to the resource.

Super class

`routr::Route` -> `Fireproof`

Active bindings

`name` The name of the plugin: `"fireproof"`

`require` Required plugins for Fireproof

`guards` The name of all the guards currently added to the plugin

Methods

Public methods:

- `Fireproof$print()`
- `Fireproof$add_auth()`
- `Fireproof$add_guard()`
- `Fireproof$add_handler()`
- `Fireproof$flow_to_openapi()`
- `Fireproof$on_attach()`
- `Fireproof$clone()`

Method `print()`**:** Print method for the class

Usage:

```
Fireproof$print(...)
```

Arguments:

`...` Ignored

Method `add_auth()`**:** Add a new authentication handler. It invisibly returns the parsed flow so it can be used for generating OpenAPI specs with.

Usage:

```
Fireproof$add_auth(method, path, flow, scope = NULL)
```

Arguments:

`method` The http method to match the handler to

`path` The URL path to match to

`flow` The authentication flow the request must pass to be valid. See *Details*. If `NULL` then authentication is turned off for the endpoint.

`scope` An optional character vector of scopes that the request must have permission for to access the resource

Method `add_guard()`**:** Add a guard to the plugin

Usage:

```
Fireproof$add_guard(guard, name = NULL)
```

Arguments:

`guard` Either a Guard object defining the guard (preferred) or a function taking the standard route handler arguments (`request`, `response`, `keys`, and `...`) and returns `TRUE` if the request is valid and `FALSE` if not.

`name` The name of the guard to be used when defining flow for endpoint auth.

Method `add_handler()`**:** Defunct overwrite of the `add_handler()` method to prevent this route to be used for anything other than auth. Will throw an error if called.

Usage:

```
Fireproof$add_handler(method, path, handler, reject_missing_methods = FALSE)
```

Arguments:

`method` ignored

`path` ignored

`handler` ignored

`reject_missing_methods` ignored

Method `flow_to_openapi()`**:** Turns a parsed flow (as returned by `add_auth()`) into an OpenAPI Security Requirement compliant list. Not all flows can be represented by the OpenAPI spec and the method will return `NULL` with a warning if so. Scope is added to all schemes, even if not applicable, so the final OpenAPI doc should be run through `prune_openapi()` before serving it.

Usage:

```
Fireproof$flow_to_openapi(flow, scope)
```

Arguments:

`flow` A parsed flow as returned by `add_auth()`

`scope` A character vector of scopes required for this particular flow

Method `on_attach()`**:** Method for use by fiery when attached as a plugin. Should not be called directly. This method looks for a header route stack in the app and if it doesn't exist it creates one. It then attaches the plugin as the first route to it.

Usage:

```
Fireproof$on_attach(app, ...)
```

Arguments:

`app` The Fire object to attach the router to

`...` Ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Fireproof$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a fireproof plugin
fp <- Fireproof$new()

# Create some authentication schemes and add them
basic <- guard_basic(
  validate = function(user, password) {
    user == "thomas" && password == "pedersen"
  },
```

```
  user_info = function(user) {
    new_user_info(
      name_given = "Thomas",
      name_middle = "Lin",
      name_family = "Pedersen"
    )
  }
)
fp$add_guard(basic, "basic_auth")

key <- guard_key(
  key_name = "my-key-location",
  validate = "SHHH!!DONT_TELL_ANYONE"
)
fp$add_guard(key, "key_auth")

google <- guard_google(
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET",
)
fp$add_guard(google, "google_auth")

# Add authentication to different paths
fp$add_auth("get", "/require_basic", basic_auth)

fp$add_auth("get", "/require_basic_and_key", basic_auth && key_auth)

fp$add_auth(
  "get",
  "/require_google_or_the_others",
  google_auth || (basic_auth && key_auth)
)


# Add plugin to fiery app
app <- fiery::Fire$new()

# First add the firesale plugin as it is required
fs <- firesale::FireSale$new(storr::driver_environment(new.env()))
app$attach(fs)

# Then add the fireproof plugin
app$attach(fp)
```

Guard *R6 base class for guards*

Description

All guards inherit from this base class and adapts it for the particular scheme it implements. Additional schemes can be implemented as subclasses of this and will work transparently with fireproof.

Usage

```
is_guard(x)
```

Arguments

x	An object

Active bindings

`name` The name of the instance

`open_api` An OpenID compliant security scheme description

Methods

Public methods:

- `Guard$new()`
- `Guard$check_request()`
- `Guard$reject_response()`
- `Guard$forbid_user()`
- `Guard$register_handler()`
- `Guard$clone()`

Method `new()`**:** Constructor for the class

Usage:

```
Guard$new(name = NULL)
```

Arguments:

`name` The name of the scheme instance

Method `check_request()`**:** A function that validates an incoming request, returning `TRUE` if it is valid and `FALSE` if not. The base class simply returns `TRUE` for all requests

Usage:

```
Guard$check_request(request, response, keys, ..., .datastore)
```

Arguments:

`request` The request to validate as a Request object

`response` The corresponding response to the request as a Response object

`keys` A named list of path parameters from the path matching

`...` Ignored

`.datastore` The data storage from firesale

Method `reject_response()`**:** Action to perform on the response in case the request fails to get validated by any instance in the flow. All failing instances will have this method called one by one so be mindful if you are overwriting information set by another instance

Usage:

```
Guard$reject_response(response, scope, ..., .datastore)
```

Arguments:

`response` The response object

`scope` The scope of the endpoint

`...` Ignored

`.datastore` The data storage from firesale

Method `forbid_user()`**:** Action to perform on the response in case the request does not have the necessary permissions for the endpoint. All succeeding instances will have this method called one by one if permissions are insufficient so be mindful if you are overwriting information set by another instance

Usage:

```
Guard$forbid_user(response, scope, ..., .datastore)
```

Arguments:

`response` The response object

`scope` The scope of the endpoint

`...` Ignored

`.datastore` The data storage from firesale

Method `register_handler()`**:** Hook for registering endpoint handlers needed for this auth method

Usage:

```
Guard$register_handler(add_handler)
```

Arguments:

`add_handler` The `add_handler` method from Fireproof to be called for adding additional handlers

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Guard$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# You'd never actually do this, rather you'd use the subclasses
guard <- Guard$new(name = "base_auth")
```

GuardBasic *R6 class for the Basic authentication guard*

Description

This class encapsulates the logic of the Basic authentication scheme. See `guard_basic()` for more information.

Super class

`fireproof::Guard` -> `GuardBasic`

Active bindings

`open_api` An OpenID compliant security scheme description

Methods

Public methods:

- `GuardBasic$new()`
- `GuardBasic$check_request()`
- `GuardBasic$reject_response()`
- `GuardBasic$clone()`

Method `new()`**:** Constructor for the class

Usage:

```
GuardBasic$new(validate, user_info = NULL, realm = "private", name = NULL)
```

Arguments:

`validate` A function that will be called with the arguments `username`, `password`, `realm`, `request`, and `response` and returns `TRUE` if the user is valid, and `FALSE` otherwise. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.

`user_info` A function to extract user information from the username. It is called with a single argument: `user` which is the username used for the successful authentication. The function should return a new user_info list.

`realm` The realm this authentication corresponds to. Will be returned to the client on a failed authentication attempt to inform them of the credentials required, though most often these days it is kept from the user.

`name` The name of the authentication

Method `check_request()`**:** A function that validates an incoming request, returning `TRUE` if it is valid and `FALSE` if not. It decodes the credentials in the `Authorization` header, splits it into username and password and then calls the `validate` function provided at construction.

Usage:

```
GuardBasic$check_request(request, response, keys, ..., .datastore)
```

Arguments:

`request` The request to validate as a Request object

`response` The corresponding response to the request as a Response object

`keys` A named list of path parameters from the path matching

`...` Ignored

`.datastore` The data storage from firesale

Method `reject_response()`**:** Upon rejection this scheme sets the response status to `401` and sets the `WWW-Authenticate` header to `Basic realm="<realm>", charset=UTF-8`

Usage:

```
GuardBasic$reject_response(response, scope, ..., .datastore)
```

Arguments:

`response` The response object

`scope` The scope of the endpoint

`...` Ignored

`.datastore` The data storage from firesale

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GuardBasic$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a guard of dubious quality
basic <- GuardBasic$new(
  validate = function(user, password) {
    user == "thomas" && password == "pedersen"
  },
  user_info = function(user) {
    new_user_info(
      name_given = "Thomas",
      name_middle = "Lin",
      name_family = "Pedersen"
    )
  }
)
```

GuardBearer *R6 class for the Bearer authentication guard*

Description

This class encapsulates the logic of the Bearer authentication scheme. See `guard_bearer()` for more information.

Super class

`fireproof::Guard` -> `GuardBearer`

Active bindings

`open_api` An OpenID compliant security scheme description

Methods

Public methods:

- `GuardBearer$new()`
- `GuardBearer$check_request()`
- `GuardBearer$reject_response()`
- `GuardBearer$clone()`

Method `new()`**:** Constructor for the class

Usage:

```
GuardBearer$new(
  validate,
  user_info = NULL,
  realm = "private",
  allow_body_token = TRUE,
  allow_query_token = FALSE,
  name = NULL
)
```

Arguments:

`validate` A function that will be called with the arguments `token`, `realm`, `request`, and `response` and returns `TRUE` if the token is valid, and `FALSE` otherwise. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.

`user_info` A function to extract user information from the token. It is called with a single argument: `token` which is the token used for the successful authentication. The function should return a new user_info list.

`realm` The realm this authentication corresponds to. Will be returned to the client on a failed authentication attempt to inform them of the credentials required, though most often these days it is kept from the user.

`allow_body_token` Should it be allowed to pass the token in the request body as a query form type with the `access_token` name. Defaults to `TRUE` but you can turn it off to force the client to use the `Authorization` header.

`allow_query_token` Should it be allowed to pass the token in the query string of the url with the `access_token` name. Default to `FALSE` due to severe security implications but can be turned on if you have very well-thought-out reasons to do so.

`name` The name of the authentication

Method `check_request()`**:** A function that validates an incoming request, returning `TRUE` if it is valid and `FALSE` if not. It fetches the token from the request according to the `allow_body_token` and `allow_query_token` settings and validates it according to the provided function. If the token is present multiple times it will fail with `400` as this is not allowed.

Usage:

```
GuardBearer$check_request(request, response, keys, ..., .datastore)
```

Arguments:

`request` The request to validate as a Request object

`response` The corresponding response to the request as a Response object

`keys` A named list of path parameters from the path matching

`...` Ignored

`.datastore` The data storage from firesale

Method `reject_response()`**:** Upon rejection this scheme sets the response status to `401` and sets the `WWW-Authenticate` header to `Bearer realm="<realm>"`. If any scope is provided by the endpoint it will be appended as `, scope="<scope>"` and if the token is present but invalid, it will append `, error="invalid_token"`

Usage:

```
GuardBearer$reject_response(response, scope, ..., .datastore)
```

Arguments:

`response` The response object

`scope` The scope of the endpoint

`...` Ignored

`.datastore` The data storage from firesale

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GuardBearer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a guard of dubious quality
bearer <- GuardBearer$new(
  validate = function(token) {
    token == "abcd1234"
```

```
  },
  user_info = function(user) {
    new_user_info(
      name_given = "Thomas",
      name_middle = "Lin",
      name_family = "Pedersen"
    )
  }
)
```

GuardKey *R6 class for the Key guard*

Description

This class encapsulates the logic of the key based authentication scheme. See `guard_key()` for more information

Super class

`fireproof::Guard` -> `GuardKey`

Active bindings

`location` The location of the secret in the request, either `"cookie"` or `"header"`

`open_api` An OpenID compliant security scheme description

Methods

Public methods:

- `GuardKey$new()`
- `GuardKey$check_request()`
- `GuardKey$reject_response()`
- `GuardKey$clone()`

Method `new()`**:** Constructor for the class

Usage:

```
GuardKey$new(key_name, validate, user_info = NULL, cookie = TRUE, name = NULL)
```

Arguments:

`key_name` The name of the header or cookie to store the secret under

`validate` Either a single string with the secret or a function that will be called with the arguments `key`, `request`, and `response` and returns `TRUE` if its a valid secret (useful if you have multiple or rotating secrets). If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted. Make sure never to store secrets in plain text and avoid checking them into version control.

`user_info` A function to extract user information from the key. It is called with a single argument: `key` which is the key used for the successful authentication. The function should return a new user_info list.

`cookie` Boolean. Should the secret be transmitted as a cookie. If `FALSE` it is expected to be transmitted as a header.

`name` The name of the guard

Method `check_request()`**:** A function that validates an incoming request, returning `TRUE` if it is valid and `FALSE` if not. It extracts the secret from either the cookie or header based on the provided `key_name` and test it using the provided `validate` function.

Usage:

```
GuardKey$check_request(request, response, keys, ..., .datastore)
```

Arguments:

`request` The request to validate as a Request object

`response` The corresponding response to the request as a Response object

`keys` A named list of path parameters from the path matching

`...` Ignored

`.datastore` The data storage from firesale

Method `reject_response()`**:** Upon rejection this guard sets the response status to `400` if it has not already been set by others. In contrast to some of the other guards which implements proper HTTP schemes, this one doesn't set a `WWW-Authenticate` header.

Usage:

```
GuardKey$reject_response(response, scope, ..., .datastore)
```

Arguments:

`response` The response object

`scope` The scope of the endpoint

`...` Ignored

`.datastore` The data storage from firesale

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GuardKey$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a guard of dubious quality
key <- GuardKey$new(
  key = "my-key-location",
  validate = "SHHH!!DONT_TELL_ANYONE"
)
```

GuardOAuth2 *R6 class for the OAuth 2.0 Guard*

Description

This class encapsulates the logic of the oauth 2.0 based authentication scheme. See `guard_oauth2()` for more information

Super class

`fireproof::Guard` -> `GuardOAuth2`

Active bindings

`open_api` An OpenID compliant security scheme description

Methods

Public methods:

- `GuardOAuth2$new()`
- `GuardOAuth2$check_request()`
- `GuardOAuth2$reject_response()`
- `GuardOAuth2$register_handler()`
- `GuardOAuth2$refresh_token()`
- `GuardOAuth2$clone()`

Method `new()`**:** Constructor for the class

Usage:

```
GuardOAuth2$new(
  token_url,
  redirect_url,
  client_id,
  client_secret,
  auth_url = NULL,
  grant_type = c("authorization_code", "password"),
  oauth_scopes = NULL,
  validate = function(info) TRUE,
  redirect_path = get_path(redirect_url),
  on_auth = replay_request,
  user_info = NULL,
  service_params = list(),
  scopes_delim = " ",
  name = NULL
)
```

Arguments:

`token_url` The URL to the authorization servers token endpoint

`redirect_url` The URL the authorization server should redirect to following a successful authorization. Must be equivalent to one provided when registering your application

`client_id` The ID issued by the authorization server when registering your application

`client_secret` The secret issued by the authorization server when registering your application. Do NOT store this in plain text

`auth_url` The URL to redirect the user to when requesting authorization (only needed for `grant_type = "authorization_code"`)

`grant_type` The type of authorization scheme to use, either `"authorization_code"` or `"password"`

`oauth_scopes` Optional character vector of scopes to request the user to grant you during authorization. These will *not* influence the scopes granted by the `validate` function and fireproof scoping. If named, the names are taken as scopes and the elements as descriptions of the scopes, e.g. given a scope, `read`, it can either be provided as `c("read")` or `c(read = "Grant read access")`

`validate` Function to validate the user once logged in. It will be called with a single argument `info`, which gets the information of the user as provided by the `user_info` function. By default it returns `TRUE` on everything meaning that anyone who can log in with the provider will be accepted, but you can provide a different function to e.g. restrict access to certain user names etc. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.

`redirect_path` The path that should capture redirects after successful authorization. By default this is derived from `redirect_url` by removing the domain part of the url, but if for some reason this doesn't yields the correct result for your server setup you can overwrite it here.

`on_auth` A function which will handle the result of a successful authorization. It will be called with four arguments: `request`, `response`, `session_state`, and `server`. The first contains the current request being responded to, the second is the response being send back, the third is a list recording the state of the original request which initiated the authorization (containing `method`, `url`, `headers`, and `body` fields with information from the original request). By default it will use replay_request to internally replay the original request and send back the response.

`user_info` A function to extract user information from the access token. It is called with a single argument: `token_info` which is the access token information returned by the OAuth 2 server after a successful authentication. The function should return a new user_info list.

`service_params` A named list of additional query params to add to the url when constructing the authorization url in the `"authorization_code"` grant type

`scopes_delim` The separator of the scopes as returned by the service. The default `" "` is the spec recommendation but some services *cough* github *cough* are non-compliant

`name` The name of the guard.

Method `check_request()`**:** A function that validates an incoming request, returning `TRUE` if it is valid and `FALSE` if not.

Usage:

```
GuardOAuth2$check_request(request, response, keys, ..., .datastore)
```

Arguments:

`request` The request to validate as a Request object

`response` The corresponding response to the request as a Response object

`keys` A named list of path parameters from the path matching

`...` Ignored

`.datastore` The data storage from firesale

Method `reject_response()`**:** Upon rejection this guard initiates the grant flow to obtain authorization. This can sound a bit backwards, but we don't want to initiate authorization if the authorization flow doesn't need it

Usage:

```
GuardOAuth2$reject_response(response, scope, ..., .datastore)
```

Arguments:

`response` The response object

`scope` The scope of the endpoint

`...` Ignored

`.datastore` The data storage from firesale

Method `register_handler()`**:** Hook for registering endpoint handlers needed for this authentication method

Usage:

```
GuardOAuth2$register_handler(add_handler)
```

Arguments:

`add_handler` The `add_handler` method from Fireproof to be called for adding additional handlers

Method `refresh_token()`**:** Refresh the access token of the session. Will return `TRUE` upon success and `FALSE` upon failure. Failure can either be issues with the token provider, but also lack of a refresh token.

Usage:

```
GuardOAuth2$refresh_token(session, force = FALSE)
```

Arguments:

`session` The session data store

`force` Boolean. Should the token be refreshed even if it hasn't expired yet

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GuardOAuth2$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Example using GitHub endpoints (use `guard_github()` in real code)
github <- GuardOAuth2$new(
  token_url = "https://github.com/login/oauth/access_token",
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET",
  auth_url = "https://github.com/login/oauth/authorize",
  grant_type = "authorization_code"
)
```

GuardOIDC *R6 class for the OpenID Connect guard*

Description

This class encapsulates the logic of the OpenID Connect based authentication scheme. See guard_oidc() for more information

Super classes

fireproof::Guard -> fireproof::GuardOAuth2 -> GuardOIDC

Active bindings

open_api An OpenID compliant security scheme description

Methods

Public methods:

- GuardOIDC$new()
- GuardOIDC$clone()

Method new(): Constructor for the class

Usage:

```
GuardOIDC$new(
  service_url,
  redirect_url,
  client_id,
  client_secret,
  grant_type = c("authorization_code", "password"),
  oauth_scopes = c("profile"),
  request_user_info = FALSE,
  validate = function(info) TRUE,
  redirect_path = get_path(redirect_url),
  on_auth = replay_request,
```

```
  service_name = service_url,
  service_params = list(),
  name = NULL
)
```

Arguments:

`service_url` The url to the authentication service

`redirect_url` The URL the authorization server should redirect to following a successful authorization. Must be equivalent to one provided when registering your application

`client_id` The ID issued by the authorization server when registering your application

`client_secret` The secret issued by the authorization server when registering your application. Do NOT store this in plain text

`grant_type` The type of authorization scheme to use, either `"authorization_code"` or `"password"`

`oauth_scopes` Optional character vector of scopes to request the user to grant you during authorization. These will *not* influence the scopes granted by the `validate` function and fireproof scoping. If named, the names are taken as scopes and the elements as descriptions of the scopes, e.g. given a scope, `read`, it can either be provided as `c("read")` or `c(read = "Grant read access")`

`request_user_info` Logical. Should the userinfo endpoint be followed to add information about the user not present in the JWT token. Setting this to `TRUE` will add an additional API call to your authentication flow but potentially provide richer information about the user.

`validate` Function to validate the user once logged in. It must have a single argument `info`, which gets the information of the user as provided by the `user_info` function in the. By default it returns `TRUE` on everything meaning that anyone who can log in with the provider will be accepted, but you can provide a different function to e.g. restrict access to certain user names etc.

`redirect_path` The path that should capture redirects after successful authorization. By default this is derived from `redirect_url` by removing the domain part of the url, but if for some reason this doesn't yields the correct result for your server setup you can overwrite it here.

`on_auth` A function which will handle the result of a successful authorization. It must have four arguments: `request`, `response`, `session_state`, and `server`. The first contains the current request being responded to, the second is the response being send back, the third is a list recording the state of the original request which initiated the authorization (containing `method`, `url`, `headers`, and `body` fields with information from the original request). By default it will use replay_request to internally replay the original request and send back the response.

`service_name` The name of the service provider. Will be passed on to the `provider` slot in the user info list

`service_params` A named list of additional query params to add to the url when constructing the authorization url in the `"authorization_code"` grant type

`name` The name of the scheme instance. This will also be the name under which token info and user info is saved in the session store

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GuardOIDC$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Example using Google endpoint (use `guard_google()` in real code)
google <- GuardOIDC$new(
  service_url = "https://accounts.google.com/",
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET"
)
```

---

guard_basic *Basic authentication guard*

Description

Basic authentication is a HTTP scheme that sends username and password as a : separated, base64 encoded string in the authorization header. Because it is effectively send in plain text (base64 encoding can easily be decoded) this should only ever be used along with other security measures such as https/ssl to avoid username and passwords being snooped from the request.

Usage

```
guard_basic(validate, user_info = NULL, realm = "private", name = "BasicAuth")
```

Arguments

validate	A function that will be called with the arguments username, password, realm, request, and response and returns TRUE if the user is valid, and FALSE otherwise. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.
user_info	A function to extract user information from the username. It is called with a single argument: user which is the username used for the successful authentication. The function should return a new user_info list.
realm	The realm this authentication corresponds to. Will be returned to the client on a failed authentication attempt to inform them of the credentials required, though most often these days it is kept from the user.
name	The name of the guard

Details

This guard will use a user-provided function to test a username/password pair. It is up to the server implementation to handle the storage and testing of the passwords in a sensible and responsible way. See sodium::password_store() for a good first step towards responsible design.

Value

A GuardBasic R6 object

User information

`guard_basic()` automatically adds user information after authentication. By default it will set the `provider` field to `"local"` and the `id` field to the username used for logging in. Further, it will set the `scopes` field to any scopes returned by the `authenticator` function.

References

Basic authentication RFC

Examples

```
# Create a guard of dubious quality
basic <- guard_basic(
  validate = function(user, password) {
    user == "thomas" && password == "pedersen"
  },
  user_info = function(user) {
    new_user_info(
      name_given = "Thomas",
      name_middle = "Lin",
      name_family = "Pedersen"
    )
  }
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(basic, "basic_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", basic_auth)
```

---

guard_bearer *Bearer authentication guard*

Description

Bearer authentication is a HTTP scheme based on tokens. It is used in a lot of places as it is often used for transmitting the tokens issued as part of OAuth 2.0 and OpenID Connect authentication. It is a quite simple scheme that is based on the concept of time- and scope-limited bearer tokens. Whoever has a valid token gains access to the resources the token unlocks. This prevents the leaking of passwords as well as makes it easy to rotate tokens etc. While the time-limited aspect of tokens means that an attacker may only gain temporary access to a resource if they intercept a token during transmission, it is still highly recommended to only transmit tokens over HTTPS

Usage

```
guard_bearer(
  validate,
  user_info = NULL,
  realm = "private",
  allow_body_token = TRUE,
  allow_query_token = FALSE,
  name = "BearerAuth"
)
```

Arguments

validate	A function that will be called with the arguments `token`, `realm`, `request`, and `response` and returns `TRUE` if the token is valid, and `FALSE` otherwise. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.
user_info	A function to extract user information from the token. It is called with a single argument: `token` which is the token used for the successful authentication. The function should return a new user_info list.
realm	The realm this authentication corresponds to. Will be returned to the client on a failed authentication attempt to inform them of the credentials required, though most often these days it is kept from the user.
allow_body_token	Should it be allowed to pass the token in the request body as a query form type with the `access_token` name. Defaults to `TRUE` but you can turn it off to force the client to use the `Authorization` header.
allow_query_token	Should it be allowed to pass the token in the query string of the url with the `access_token` name. Default to `FALSE` due to severe security implications but can be turned on if you have very well-thought-out reasons to do so.
name	The name of the guard

Details

This `validate` function is provided by the user and is used to test the provided token. The complexity of the test fully depends on the issuer of the token. At it's simplest the token is opaque and the function test it against a database. However, it is more common to use a JSON web token to encode various information into the token itself that can help in determining scoped access etc.

The `validate` function should not test the scope of the token, but rather return a vector of scopes (which implicitly means that the token is valid). The scope requirement of the exact endpoint will then be tested automatically.

Value

A GuardBearer R6 object

User information

`guard_bearer()` automatically adds user information after authentication. By default it will set the `provider` field to `"local"`. Further, it will set the `scopes` field to any scopes returned by the `validate` function and the `token` field to a list with the following elements:

- `access_token`: The provided token
- `token_type`: `"bearer"`
- `scope` The scopes concatenated into a space separated string

This structure mimics the structure of the token information returned by OAuth 2.0 and OpenID Connect services.

References

Bearer authentication RFC

Examples

```
# Create a guard of dubious quality
bearer <- guard_bearer(
  validate = function(token) {
    token == "abcd1234"
  },
  user_info = function(user) {
    new_user_info(
      name_given = "Thomas",
      name_middle = "Lin",
      name_family = "Pedersen"
    )
  },
  allow_body_token = FALSE
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(bearer, "bearer_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", bearer_auth)
```

---

guard_beeceptor *Guard using the mock OAuth servers provided by Beeceptor*

Description

These two functions sets up mock OAuth 2.0 guards based on tools provided by Beeceptor. They should obviously not be used for production because they allow anyone to be authenticated, but they can be used while testing your authentication setup.

Usage

```
guard_beeceptor_github(
  redirect_url,
  client_id = "MOCK_CLIENT",
  ...,
  name = "beeceptor_github"
)

guard_beeceptor_google(
  redirect_url,
  client_id = "MOCK_CLIENT",
  ...,
  name = "beeceptor_google"
)
```

Arguments

`redirect_url`	The URL the authorization server should redirect to following a successful authorization. Must be equivalent to one provided when registering your application
`client_id`	The ID issued by the authorization server when registering your application

`...`	Arguments passed on to `guard_oauth2`
	`token_url` The URL to the authorization servers token endpoint
	`client_secret` The secret issued by the authorization server when registering your application. Do NOT store this in plain text
	`auth_url` The URL to redirect the user to when requesting authorization (only needed for `grant_type = "authorization_code"`)
	`grant_type` The type of authorization scheme to use, either `"authorization_code"` or `"password"`
	`oauth_scopes` Optional character vector of scopes to request the user to grant you during authorization. These will *not* influence the scopes granted by the `validate` function and fireproof scoping. If named, the names are taken as scopes and the elements as descriptions of the scopes, e.g. given a scope, `read`, it can either be provided as `c("read")` or `c(read = "Grant read access")`
	`validate` Function to validate the user once logged in. It will be called with a single argument `info`, which gets the information of the user as provided by the `user_info` function in the. By default it returns `TRUE` on everything meaning that anyone who can log in with the provider will be accepted, but you can provide a different function to e.g. restrict access to certain user names etc. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.
	`redirect_path` The path that should capture redirects after successful authorization. By default this is derived from `redirect_url` by removing the domain part of the url, but if for some reason this doesn't yields the correct result for your server setup you can overwrite it here.

on_auth A function which will handle the result of a successful authorization. It will be called with four arguments: request, response, session_state, and server. The first contains the current request being responded to, the second is the response being send back, the third is a list recording the state of the original request which initiated the authorization (containing method, url, headers, and body fields with information from the original request). By default it will use replay_request to internally replay the original request and send back the response.

user_info A function to extract user information from the access token. It is called with a single argument: token_info which is the access token information returned by the OAuth 2 server after a successful authentication. The function should return a new user_info list.

service_params A named list of additional query params to add to the url when constructing the authorization url in the "authorization_code" grant type

scopes_delim The separator of the scopes as returned by the service. The default " " is the spec recommendation but some services *cough* github *cough* are non-compliant

name The name of the guard

Value

A GuardOAuth2 object

Examples

```
beeceptor <- guard_beeceptor_github(
  redirect_url = "https://example.com/auth"
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(beeceptor, "beeceptor_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", beeceptor_auth)
```

---

guard_github *Guard for authenticating with the GitHub OAuth 2.0 server*

Description

This guard requests you to log in with GitHub and authenticates yourself through their service. Your server must be registered and have a valid client ID and client secret for this to work. Register an application at https://github.com/settings/applications/new. If you want to limit access to only select users you should make sure to provide a validate function that checks the userinfo against a whitelist.

Usage

```
guard_github(redirect_url, client_id, client_secret, ..., name = "github")
```

Arguments

redirect_url	The URL the authorization server should redirect to following a successful authorization. Must be equivalent to one provided when registering your application
client_id	The ID issued by the authorization server when registering your application
client_secret	The secret issued by the authorization server when registering your application. Do NOT store this in plain text
...	Arguments passed on to guard_oauth2
	grant_type The type of authorization scheme to use, either "authorization_code" or "password"
	oauth_scopes Optional character vector of scopes to request the user to grant you during authorization. These will *not* influence the scopes granted by the validate function and fireproof scoping. If named, the names are taken as scopes and the elements as descriptions of the scopes, e.g. given a scope, read, it can either be provided as c("read") or c(read = "Grant read access")
	validate Function to validate the user once logged in. It will be called with a single argument info, which gets the information of the user as provided by the user_info function in the. By default it returns TRUE on everything meaning that anyone who can log in with the provider will be accepted, but you can provide a different function to e.g. restrict access to certain user names etc. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.
	redirect_path The path that should capture redirects after successful authorization. By default this is derived from redirect_url by removing the domain part of the url, but if for some reason this doesn't yields the correct result for your server setup you can overwrite it here.
	on_auth A function which will handle the result of a successful authorization. It will be called with four arguments: request, response, session_state, and server. The first contains the current request being responded to, the second is the response being send back, the third is a list recording the state of the original request which initiated the authorization (containing method, url, headers, and body fields with information from the original request). By default it will use replay_request to internally replay the original request and send back the response.
	service_params A named list of additional query params to add to the url when constructing the authorization url in the "authorization_code" grant type
	scopes_delim The separator of the scopes as returned by the service. The default " " is the spec recommendation but some services *cough* github *cough* are non-compliant
name	The name of the guard

Value

A GuardOAuth2 object

User information

`guard_github()` automatically adds user information according to the description in `guard_oauth2()`.
It sets the `provider` field to `"github"`. Further, extracts information from the `https://api.github.com/user` endpoint and maps the information accordingly:

- `id` -> `id`
- `name` -> `name_display`
- `login` -> `name_user`
- `email` -> `emails`
- `avatar_url` -> `photos`

It also sets the `.raw` field to the full list of information returned from github.

References

Documentation for GitHub's OAuth 2 flow

Examples

```
github <- guard_github(
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET"
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(github, "github_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", github_auth)
```

---

guard_google *Guard for Authenticating with the Google OpenID Connect server*

Description

This guard requests you to log in with google and authenticates you through their service. Your server must be registered and have a valid client ID and client secret for this to work. Read more about registering an application at `https://developers.google.com/identity/protocols/oauth2`.
If you want to limit access to only select users you should make sure to provide a `validate` function that checks the userinfo against a whitelist.

Usage

```
guard_google(
  redirect_url,
  client_id,
  client_secret,
  oauth_scopes = "profile",
 service_params = list(access_type = "offline", include_granted_scopes = "true"),
  ...,
  name = "google"
)
```

Arguments

redirect_url	The URL the authorization server should redirect to following a successful authorization. Must be equivalent to one provided when registering your application
client_id	The ID issued by the authorization server when registering your application
client_secret	The secret issued by the authorization server when registering your application. Do NOT store this in plain text
oauth_scopes	Optional character vector of scopes to request the user to grant you during authorization. These will *not* influence the scopes granted by the `validate` function and fireproof scoping. If named, the names are taken as scopes and the elements as descriptions of the scopes, e.g. given a scope, `read`, it can either be provided as `c("read")` or `c(read = "Grant read access")`
service_params	A named list of additional query params to add to the url when constructing the authorization url in the `"authorization_code"` grant type
...	Arguments passed on to guard_oidc `request_user_info` Logical. Should the userinfo endpoint be followed to add information about the user not present in the JWT token. Setting this to `TRUE` will add an additional API call to your authentication flow but potentially provide richer information about the user. `grant_type` The type of authorization scheme to use, either `"authorization_code"` or `"password"` `validate` Function to validate the user once logged in. It will be called with a single argument `info`, which gets the information of the user as provided by the `user_info` function in the. By default it returns `TRUE` on everything meaning that anyone who can log in with the provider will be accepted, but you can provide a different function to e.g. restrict access to certain user names etc. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted. `redirect_path` The path that should capture redirects after successful authorization. By default this is derived from `redirect_url` by removing the domain part of the url, but if for some reason this doesn't yields the correct result for your server setup you can overwrite it here.

on_auth A function which will handle the result of a successful authorization. It will be called with four arguments: request, response, session_state, and server. The first contains the current request being responded to, the second is the response being send back, the third is a list recording the state of the original request which initiated the authorization (containing method, url, headers, and body fields with information from the original request). By default it will use replay_request to internally replay the original request and send back the response.

name The name of the guard

Value

A GuardOIDC object

User information

guard_google() automatically adds user information according to the description in guard_oidc(). It sets the provider field to "google".

References

Documentation for Googles OpenID Connect flow

Examples

```
google <- guard_google(
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET"
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(google, "google_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", google_auth)
```

---

guard_key *Shared secret guard*

Description

This guard is based on a mutually shared secret between the server and the client. The client provides the secret either as a header or in a cookie, and the server verifies the authenticity of the secret. Like with basic authentication, this scheme relies on additional technology like HTTPS/SSL to make it secure since the secret can otherwise easily be extracted from the request by man-in-the-middle attack.

Usage

```
guard_key(
  key_name,
  validate,
  user_info = NULL,
  cookie = TRUE,
  name = "KeyAuth"
)
```

Arguments

key_name	The name of the header or cookie to store the secret under
validate	Either a single string with the secret or a function that will be called with the arguments key, request, and response and returns TRUE if its a valid secret (useful if you have multiple or rotating secrets). If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted. Make sure never to store secrets in plain text and avoid checking them into version control.
user_info	A function to extract user information from the key. It is called with a single argument: key which is the key used for the successful authentication. The function should return a new user_info list.
cookie	Boolean. Should the secret be transmitted as a cookie. If FALSE it is expected to be transmitted as a header.
name	The name of the guard

Details

This authentication is not a classic HTTP authentication scheme and thus doesn't return a 401 response with a WWW-Authenticate header. Instead it returns a 400 response unless another guard has already set the response status to something else.

Value

A GuardKey object

User information

guard_key() automatically adds user information after authentication. By default it will set the provider field to "local". Further, it will set the scopes field to any scopes returned by the validate function (provided validate is passed a function).

Since key-based authentication is seldom used with user specific keys it is unlikely that it makes sense to populate the information any further.

Examples

```
# Create a guard of dubious quality
key <- guard_key(
  key_name = "my-key-location",
```

```
  validate = "SHHH!!DONT_TELL_ANYONE"
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(key, "key_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", key_auth)
```

---

guard_oauth2 *Guard based on OAuth 2.0*

Description

OAuth 2.0 is an authorization scheme that is powering much of the modern internet and is behind things like "log in with GitHub" etc. It separates the responsibility of authentication away from the server, and allows a user to grant limited access to a service on the users behalf. While OAuth also allows a server to make request on the users behalf the main purpose in the context of `fireproof` is to validate that the user can perform a successful login and potentially extract basic information about the user. The `guard_oauth2()` function is the base constructor which can be used to create guards with any provider. For ease of use `fireproof` comes with a range of predefined constructors for popular services such as GitHub etc. Central for all of these is the need for your server to register itself with the provider and get a client id and a client secret which must be used when logging users in.

Usage

```
guard_oauth2(
  token_url,
  redirect_url,
  client_id,
  client_secret,
  auth_url = NULL,
  grant_type = c("authorization_code", "password"),
  oauth_scopes = NULL,
  validate = function(info) TRUE,
  redirect_path = get_path(redirect_url),
  on_auth = replay_request,
  user_info = NULL,
  service_params = list(),
  scopes_delim = " ",
  name = "OAuth2Auth"
)
```

Arguments

token_url	The URL to the authorization servers token endpoint
redirect_url	The URL the authorization server should redirect to following a successful authorization. Must be equivalent to one provided when registering your application
client_id	The ID issued by the authorization server when registering your application
client_secret	The secret issued by the authorization server when registering your application. Do NOT store this in plain text
auth_url	The URL to redirect the user to when requesting authorization (only needed for `grant_type = "authorization_code"`)
grant_type	The type of authorization scheme to use, either `"authorization_code"` or `"password"`
oauth_scopes	Optional character vector of scopes to request the user to grant you during authorization. These will *not* influence the scopes granted by the `validate` function and fireproof scoping. If named, the names are taken as scopes and the elements as descriptions of the scopes, e.g. given a scope, `read`, it can either be provided as `c("read")` or `c(read = "Grant read access")`
validate	Function to validate the user once logged in. It will be called with a single argument `info`, which gets the information of the user as provided by the `user_info` function in the. By default it returns `TRUE` on everything meaning that anyone who can log in with the provider will be accepted, but you can provide a different function to e.g. restrict access to certain user names etc. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.
redirect_path	The path that should capture redirects after successful authorization. By default this is derived from `redirect_url` by removing the domain part of the url, but if for some reason this doesn't yields the correct result for your server setup you can overwrite it here.
on_auth	A function which will handle the result of a successful authorization. It will be called with four arguments: `request`, `response`, `session_state`, and `server`. The first contains the current request being responded to, the second is the response being send back, the third is a list recording the state of the original request which initiated the authorization (containing `method`, `url`, `headers`, and `body` fields with information from the original request). By default it will use replay_request to internally replay the original request and send back the response.
user_info	A function to extract user information from the access token. It is called with a single argument: `token_info` which is the access token information returned by the OAuth 2 server after a successful authentication. The function should return a new user_info list.
service_params	A named list of additional query params to add to the url when constructing the authorization url in the `"authorization_code"` grant type
scopes_delim	The separator of the scopes as returned by the service. The default `" "` is the spec recommendation but some services *cough* github *cough* are non-compliant
name	The name of the guard

Value

A GuardOAuth2 object

User information

`guard_oauth2()` automatically adds some user information after authentication, but it is advised to consult the service provider for more information (this is done automatically for the provider specific guards. See their documentation for details about what information is assigned to which field). The base constructor will set the `scopes` field to any scopes returned by the `validate` function. It will also set the `token` field to a list with the token data provided by the service during authorization. Some standard fields in the list are:

- `access_token`: The actual token value
- `token_type`: The type of token (usually `"bearer"`)
- `expires_in`: The lifetime of the token in seconds
- `refresh_token`: A long-lived token that can be used to issue a new access token if the current becomes stale
- `timestamp`: The time the token was received
- `scopes`: The scopes granted by the user for this token

But OAuth 2.0 providers may choose to supply others. Consult the documentation for the provider to learn of additional fields it may provide.

References

The OAuth 2.0 RFC

Examples

```
# Example using GitHub endpoints (use `guard_github()` in real code)
github <- guard_oauth2(
  token_url = "https://github.com/login/oauth/access_token",
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET",
  auth_url = "https://github.com/login/oauth/authorize",
  grant_type = "authorization_code"
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(github, "github_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", github_auth)
```

guard_oidc *Guard based on OpenID Connect*

Description

OpenID Connect is an authentication standard build on top of OAuth 2.0. OAuth 2.0 at its core is only about authorization and doesn't provide a standardized approach to extracting user information that can be used for authentication. OpenID Connect fills this gap in a number of ways. First, the token returned is a JSON Web Token (JWT) that contains claims about the user, signed by the issuer. Second, the authentication service provides means for discovery of all relevant end points making rotation of credentials etc easier. Third, the claims about users are standardized so authentication services are easily interchangable. Not all OAuth 2.0 authorization services provide an OpenID Connect layer, but if they do, it is generally preferable to use that. The `guard_oidc()` function is the base constructor which can be used to create authenticators with any provider. For ease of use `fireproof` comes with a range of predefined constructors for popular services such as Google etc. Central for all of these is the need for your server to register itself with the provider and get a client id and a client secret which must be used when logging users in.

Usage

```
guard_oidc(
  service_url,
  redirect_url,
  client_id,
  client_secret,
  grant_type = c("authorization_code", "password"),
  oauth_scopes = c("profile"),
  request_user_info = FALSE,
  validate = function(info) TRUE,
  redirect_path = get_path(redirect_url),
  on_auth = replay_request,
  service_name = service_url,
  service_params = list(),
  name = "OIDCAuth"
)
```

Arguments

`service_url`	The url to the authentication service
`redirect_url`	The URL the authorization server should redirect to following a successful authorization. Must be equivalent to one provided when registering your application
`client_id`	The ID issued by the authorization server when registering your application
`client_secret`	The secret issued by the authorization server when registering your application. Do NOT store this in plain text

`grant_type` The type of authorization scheme to use, either `"authorization_code"` or `"password"`

`oauth_scopes` Optional character vector of scopes to request the user to grant you during authorization. These will *not* influence the scopes granted by the `validate` function and fireproof scoping. If named, the names are taken as scopes and the elements as descriptions of the scopes, e.g. given a scope, `read`, it can either be provided as `c("read")` or `c(read = "Grant read access")`

`request_user_info` Logical. Should the userinfo endpoint be followed to add information about the user not present in the JWT token. Setting this to `TRUE` will add an additional API call to your authentication flow but potentially provide richer information about the user.

`validate` Function to validate the user once logged in. It will be called with a single argument `info`, which gets the information of the user as provided by the `user_info` function in the. By default it returns `TRUE` on everything meaning that anyone who can log in with the provider will be accepted, but you can provide a different function to e.g. restrict access to certain user names etc. If the function returns a character vector it is considered to be authenticated and the return value will be understood as scopes the user is granted.

`redirect_path` The path that should capture redirects after successful authorization. By default this is derived from `redirect_url` by removing the domain part of the url, but if for some reason this doesn't yields the correct result for your server setup you can overwrite it here.

`on_auth` A function which will handle the result of a successful authorization. It will be called with four arguments: `request`, `response`, `session_state`, and `server`. The first contains the current request being responded to, the second is the response being send back, the third is a list recording the state of the original request which initiated the authorization (containing `method`, `url`, `headers`, and `body` fields with information from the original request). By default it will use replay_request to internally replay the original request and send back the response.

`service_name` The name of the service provider. Will be passed on to the `provider` slot in the user info list

`service_params` A named list of additional query params to add to the url when constructing the authorization url in the `"authorization_code"` grant type

`name` The name of the guard

Value

An GuardOIDC object

User information

`guard_oidc()` automatically adds user information after authentication, based on the standardized user claims provided in the `id_token` as well as any additional user information provided at the `userinfo_endpoint` of the service if `request_user_info = TRUE`. You can see a list of standard

user information defined by OpenID Connect at the OpenID website. The mapping of these to `new_user_info()` is as follows:

- `sub` -> `id`
- `name` -> `name_display`
- `given_name` -> `name_given`
- `middle_name` -> `name_middle`
- `family_name` -> `name_family`
- `email` -> `emails`
- `picture` -> `photos`

Further, it will set the `scopes` field to any scopes returned by the `validate` function, the `provider` field to `service_name`, the `token` field to the token information as described in `guard_oauth2()`, and `.raw` to the full list of user information as provided unaltered by the service. Be aware that the information reported by the service depends on the `oauth_scopes` requested by fireproof and granted by the user. You can therefore never assume the existence of any information besides `id`, `provider` and `token`.

Examples

```
# Example using Google endpoint (use `guard_google()` in real code)
google <- guard_oidc(
  service_url = "https://accounts.google.com/",
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET"
)

# Add it to a fireproof plugin
fp <- Fireproof$new()
fp$add_guard(google, "google_auth")

# Use it in an endpoint
fp$add_auth("get", "/*", google_auth)
```

`new_user_info` *Well structured user information*

Description

Different services and authentication schemes may present user information in different ways. To ensure ease of interoperability, fireproof will attempt to standardize the information as it gets extracted by the service. This function is intended to be called to construct the output of `user_info` function.

Usage

```
new_user_info(
  provider = NULL,
  id = NULL,
  name_display = NULL,
  name_given = NULL,
  name_middle = NULL,
  name_family = NULL,
  name_user = NULL,
  emails = NULL,
  photos = NULL,
  ...
)
```

Arguments

provider	A string naming the provider of the user information
id	A unique identifier of this user
name_display, name_given, name_middle, name_family, name_user	The legal name of the user. Will be combined to a single name field in the output with the structure c(given = name_given, middle = name_middle, family = name_family, display = name_display, user = name_user)
emails	A character vector of emails to the user. The vector can be named in which case the names correspond to the category of email, e.g. "work", "home" etc.
photos	A character vector of urls pointing to profile pictures of the user.
...	Additional named arguments to be added to the user information

Value

A list of class fireproof_user_info. The fields of the list are as provided in the arguments except for the name_* arguments which will be combined to a single field. See the description of these arguments for more information.

Setting user information

Each authentication scheme will write to a field in the session data store named after its own name. What gets written can sometimes be influenced by the user by passing in a function to the user_info argument of the constructor. This output of this function will be combined with default information from the guard before being saved in the session storage (e.g. the scopes field is always created automatically).

Examples

```
new_user_info(
  provider = "local",
  id = 1234,
  name_display = "thomasp85",
  name_given = "Thomas",
```

```
  name_middle = "Lin",
  name_family = "Pedersen"
)
```

---

on_auth *Predefined functions for handling successful OAuth 2.0 authentication*

Description

When using the "authorization code" grant type in an OAuth 2.0 authorization flow, you have to decide what to do after an access token has been successfully retrieved. Since the flow goes through multiple redirection the original request is no longer available once the access token has been retrieved. The `replay_request()` function will use the saved session_state from the original request to construct a "fake" request and replay that on the server to obtain the response it would have given had the user already been authorized. The `redirect_back()` function will try to redirect the user back to the location they where at when they send the request that prompted authorization. You can also create your own function that e.g. presents a "Successfully logged on" webpage. See *Details* for information on the requirements for such a function.

Usage

```
replay_request(request, response, session_state, server)

redirect_back(request, response, session_state, server)

redirect_to(url)
```

Arguments

`request`	The current request being handled, as a reqres::Request object. The result of a redirection from the authorization server.
`response`	The response being returned to the user as a reqres::Response object.
`session_state`	A list of stored information from the original request. Contains the following fields: `state` (a random string identifying the authorization attempt), `time` (the timestamp of the original request), `method` (the http method of the original request), `url` (the full url of the original request, including any query string), `headers` (A named list of all the headers of the original request), `body` (a raw vector of the body of the original request), and `from` (The url the original request was sent from)
`server`	The fiery server handling the request
`url`	The URL to redirect to after successful authentication

Details

Creating your own success handler is easy and just requires that you conform to the input arguments of the functions described here. The main purpose of the function is to modify the response object that is being send back to the user to fit your needs. As with any routr handler it should return a boolean indicating if further processing should happen. For this situation it is usually sensible to return `FALSE`.

Value

`TRUE` if the request should continue processing in the server or `FALSE` if the response should be send straight away

Examples

```
# These functions are never called directly but passed on to the `on_auth`
# argument in OAuth 2.0 and OpenID Connect authentication flows

# Default
google <- guard_google(
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET",
  on_auth = replay_request
)

# Alternative
google <- guard_google(
  redirect_url = "https://example.com/auth",
  client_id = "MY_APP_ID",
  client_secret = "SUCHASECRET",
  on_auth = redirect_back
)
```

---

prune_openapi *Ensure consistency of OpenAPI auth description*

Description

Prune an OpenAPI doc so that security descriptions only contains references to the schemes defined in `securitySchemes` and only contains scopes for the schemes that are OAuth2.0 and OpenID. For OAuth2.0 specifically, scopes are removed if they are not explicitly named in `securitySchemes`.

Usage

```
prune_openapi(doc)
```

Arguments

doc	A list describing a full OpenAPI documentation

Value

The doc modified so the auth descriptions are internally consistent

Examples

```
# OpenAPI stub only containing relevant info for example
openapi <- list(
  components = list(
    securitySchemes = list(
      auth1 = list(
        type = "http",
        scheme = "basic"
      ),
      auth2 = list(
        type = "oauth2",
        flows = list(
          authorizationCode = list(
            scopes = list(
              read = "read data",
              write = "change data"
            )
          )
        )
      )
    )
  ),
  # Global auth settings
  security = list(
    list(auth1 = c("read", "write"))
  ),
  # Path specific auth settings
  paths = list(
    "/user/{username}" = list(
      get = list(
        security = list(
          list(auth2 = c("read", "write", "commit")),
          list(auth3 = c("read"))
        )
      )
    )
  )
)

prune_openapi(openapi)
```

Index