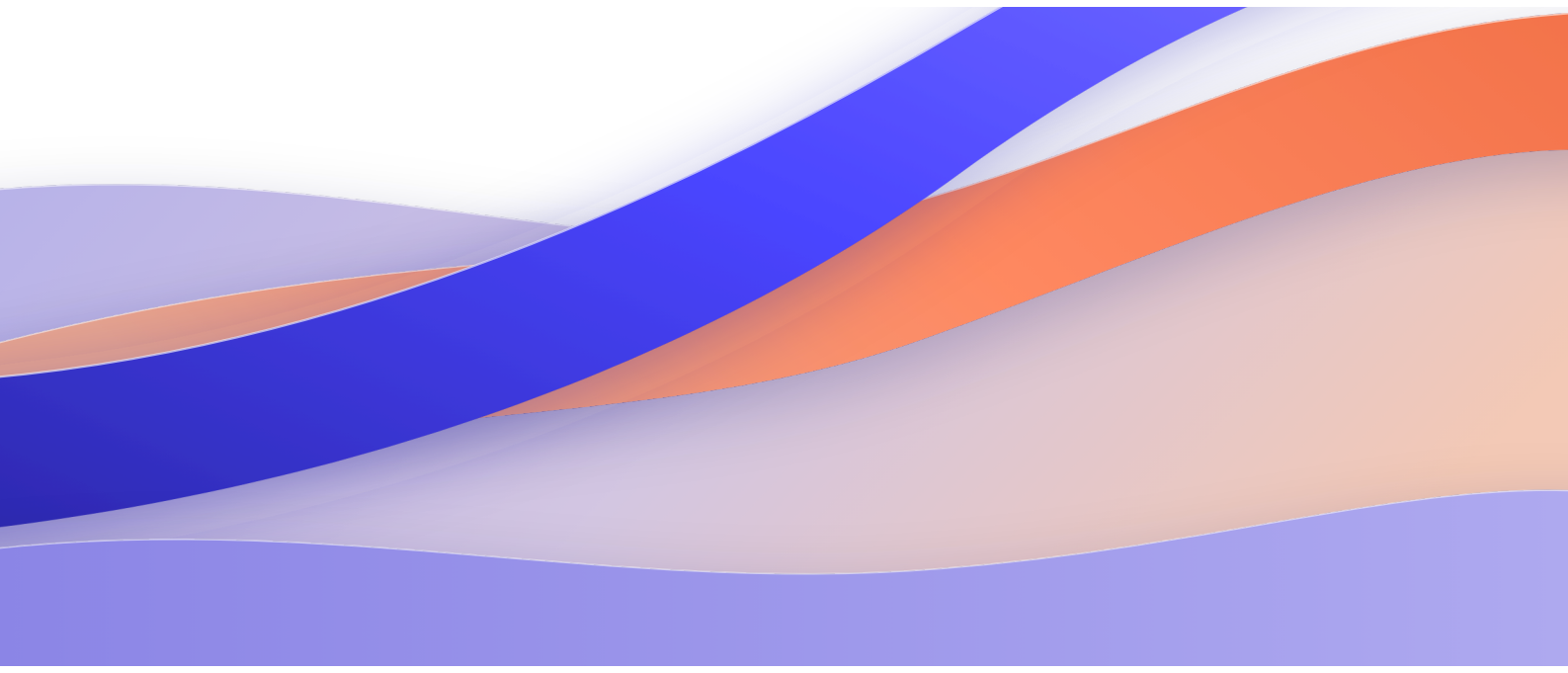
Foldr

Foldr API v1

Developer guide

by Foldr HQ – October 2026



Contents

1.0 Foldr API v1: developer guide
- 1.1 What the API is
- 1.2 Base URL
- 1.3 The OpenAPI document
- 1.4 Stability
- 1.5 What's in this guide
- 1.6 A first look

2.0 Getting access
- 2.1 What an administrator does
- 2.2 Scopes
- 2.3 The sign-in flow, step by step
- 2.4 Token lifetimes
- 2.5 Refreshing
- 2.6 Errors from the token endpoint
- 2.7 Revoking a token
- 2.8 What a person's Disconnect does
- 2.9 Things that stop an app seeing a share

3.0 Conventions
- 3.1 Requests and responses
- 3.2 Ids
- 3.3 Paths
- 3.4 Conditional requests: ETags
- 3.5 Idempotency keys
- 3.6 Pagination
- 3.7 Timestamps
- 3.8 Rate limits
- 3.9 HEAD
- 3.10 405 and 404

4.0 Errors
- 4.1 Every problem type
- 4.2 Which statuses can an operation give?

5.0 Reading
- 5.1 Who am I: `getMe`
- 5.2 Shares: `listShares`, `getShare`
- 5.3 Listing a folder: `listChildren`
- 5.4 One item: `getItem`
- 5.5 Downloading: `getContent`, `getDownloadUrl`
- 5.6 Search: `search`
- 5.7 Custom fields: `listFields`, `getFieldValues`
- 5.8 Versions: `listVersions`

6.0 Writing
- 6.1 Before you write
- 6.2 Create a folder: `createFolder`
- 6.3 Upload a file: `uploadFile`
- 6.4 Upload straight to the storage: upload sessions
- 6.5 Rename: `renameItem`
- 6.6 Copy: `copyItem`
- 6.7 Replace a file's contents: `overwriteFile`
- 6.8 Put an earlier version back: `restoreVersion`
- 6.9 Move: `moveItem`
- 6.10 Delete: `deleteItem`
- 6.11 Setting a custom field value: `setFieldValue`
- 6.12 Asking to share: `createShareRequest`

7.0 Operations and approval

- 7.1 When you get an operation
- 7.2 The `202`
- 7.3 Polling: `getOperation`
- 7.4 Finding an operation again: `listOperations`
- 7.5 Cancelling: `cancelOperation`
- 7.6 Moves and `source-remains`
- 7.7 Idempotent retries
- 7.8 What can stop an operation
- 7.9 Changes held for approval

8.0 Recoverability and audit

- 8.1 Why recoverability matters to a client
- 8.2 What a delete tells you
- 8.3 By storage
- 8.4 What an administrator can refuse
- 8.5 Telling the person afterwards
- 8.6 What administrators can see (audit)

9.0 A worked example

- 9.1 What to notice

10.0 Trying it in Postman

- 10.1 Before you start
- 10.2 Setting up
- 10.3 Working through the folders
- 10.4 Reading the results
- 10.5 Using a token from somewhere else

11.0 Troubleshooting and a checklist

- 11.1 Questions that come up
- 11.2 A checklist for a well-behaved client

1.0 Foldr API v1: developer guide

Developer guide, October 2026

This guide is for developers building an app or integration that works with files in Foldr, on a Foldr Cloud site or on a Foldr server (an appliance) run by an organisation. You don't need to know anything about how Foldr is built. With this guide and the OpenAPI document you can build a working client from nothing.

The guide says how the API behaves and why; the OpenAPI document is the exact contract (every field, header and status code). Where the two ever seem to differ, the OpenAPI document wins, and we'd like to hear about it.

1.1 What the API is

`/api/v1` is Foldr's public REST API. A few things are worth knowing before anything else.

- **An app acts as a person.** The person signs in to Foldr, sees what your app asks for, and agrees. From then on your app calls the API with an OAuth access token and sees exactly what that person can see in Foldr, and can do only what they could do themselves (and, for changes, only what their organisation has allowed your app to do). There are no service accounts and no app-wide access.
- **Files live in shares.** Foldr puts many kinds of storage (file servers, Microsoft 365, Google Drive, S3 and more) behind one set of shares. You address a share by its id and a file or folder by a path inside that share. The API looks the same whatever sits behind the share.
- **JSON in and out**, except file contents, which are raw bytes. Errors are `application/problem+json` (RFC 9457).
- **It's meant to be called from servers, scripts and automation tools**, not from web pages. It sends no CORS headers, and `OPTIONS` is answered 405.
- **Every call is recorded** against the person and your app, so an administrator can see what an app did.

1.2 Base URL

Every address is under `/api/v1` on the host you sign in to Foldr at.

Where Foldr runs	Base URL
Foldr Cloud	`https://your-company.foldr.cloud/api/v1`
Your own domain on Foldr Cloud	`https://files.your-company.example/api/v1`
A Foldr server (appliance)	`https://<the server's name>/api/v1`

Use the address people type into their browser to reach Foldr, with `/api/v1` after it. There is no central endpoint: each site is its own authorisation server and its own API, so an app that serves several organisations keeps one base URL, one set of tokens and one client registration per site.

In this guide the examples use `https://files.example.com` as the site. Replace it with yours.

1.3 The OpenAPI document

The API describes itself in an OpenAPI 3.1 document, served by the API itself and needing no token:

```
GET https://files.example.com/api/v1/openapi.json
```

Connector platforms (Power Automate, Logic Apps, n8n and the like) and code generators can import it from that address before they have a token. It is cached for five minutes and carries an `ETag`. Every operation has an `operationId` (`listShares`, `uploadFile` and so on); this guide uses those names, so you can match a section of the guide to an operation in the document.

The document also holds a few Foldr extensions:

- `x-foldr-rate-cost` on each operation says what one call takes from your app's rate budget (see Conventions).
- `info.version` is the document's own version number, and `info.x-changelog` lists what changed in each. Neither is the API's major version, which is the `v1` in the path.

1.4 Stability

The major version is in the path and nothing else selects a version: there is no version header and there are no dated versions. Within `v1` we may add things, and we won't break what's there.

What may change within v1 at any time:

- new endpoints, new optional parameters, new optional headers;
- new fields in a response;
- new values in a list the reference marks as open (marked `x-extensible-enum`, or described as "new values may appear"): Share `type`, Field `type`, Operation `kind` and `phase`, ShareRequest `kind` and `status`, and a few more;
- new problem types;
- a looser limit;
- the encoding of cursors, which are opaque.

So your client should **ignore fields it doesn't recognise**, **accept enum values it doesn't recognise** where the reference says a list is open, and **branch on a problem's `type` or `code`, never on its `title` or `detail`**.

What needs a new major version (`/api/v2`) or the deprecation route: removing or renaming an endpoint, field or parameter; changing a field's type or meaning; making an optional input required; validating an existing input more strictly; a different status code for an existing condition; a changed default; a different scope on an existing endpoint; changing how pages or ids work; and changing how paths are written back to you (their case, their Unicode form, the leading `/`).

How long things last:

- A major version is supported for at least 24 months after the next one is generally available.
- A deprecated endpoint keeps working for at least 12 months before its sunset date.
- On your own Foldr server, removal also waits for a major Foldr release, and the release notes of the two releases before it say what's going, because servers upgrade on their owners' schedules.
- You'll know an endpoint is deprecated from a `Deprecation` header (RFC 9745) on every response from it, a `Sunset` header (RFC 8594) once its date is set, a `Link` header with `rel="deprecation"`, and `deprecated: true` in the OpenAPI document.
- An endpoint marked `x-foldr-stability: beta` may change on 30 days' notice and stays off unless the organisation opts in. There are none in v1 today.

One exception, which we'd rather say plainly: a change that closes a security hole may break a client without notice. We'll say what changed straight afterwards.

1.5 What's in this guide

Page	What it covers
Getting access	Turning connected apps on, registering your app, the OAuth 2.1 sign-in flow step by step, scopes, token lifetimes, refresh and revocation
Conventions	Ids, paths, ETags and conditional requests, idempotency keys, pagination, timestamps, rate limits, HEAD, 405 and 404
Errors	The problem document and every problem type, with what to do about each
Reading	Shares, listings, item metadata, downloads, search, custom fields, versions
Writing	Folders, uploads (streamed and direct), rename, overwrite, move, copy, delete, restore a version, field values, share requests
Operations and approval	Long-running operations (`202` and polling), cancelling, and changes held for approval
Recoverability and audit	What "can it be undone?" means on each kind of storage, and what administrators can see of an app
A worked example	A short end-to-end client in Python
Trying it in Postman	A ready-made Postman collection with a request for every operation, and how to use it
Troubleshooting and a checklist	Common questions, and what a well-behaved client does

1.6 A first look

Once you have an access token (see Getting access), this is the whole of a first session:

```
export FOLDR=https://files.example.com
export TOKEN=...   # an access token; never paste a real one into a script you commit

# Who am I?
curl -s -H "Authorization: Bearer $TOKEN" $FOLDR/api/v1/me

# Which shares can I read?
```

```
curl -s -H "Authorization: Bearer $TOKEN" $FOLDR/api/v1/shares

# What's in the top of one of them?
curl -s -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/shares/01a11dce-14f0-7f23-b857-c621356097a0/children"
```

The reference for every operation is the OpenAPI document. The rest of this guide is about the things the document can't say well: how to get a token, what a person's organisation can decide about your app, and how to behave when things go wrong.

2.0 Getting access

An app gets access to a Foldr site in three steps: an administrator turns connected apps on, your app is registered with the site, and a person signs in and agrees to what the app asks for. This page covers all three, then the tokens you end up with.

Foldr uses OAuth 2.1: the authorisation code flow with PKCE, and refresh tokens. There are no API keys, no app passwords and no passwords handled by your app; a person always signs in to Foldr itself, with whatever the organisation uses (a Foldr password, Microsoft Entra, Google, SAML, two-factor, a passkey), and your app only ever sees the result.

2.1 What an administrator does

If you're building an app for a customer, this is what you'll ask their Foldr administrator to do. If you run Foldr yourself, it's your own setup.

1. **Check the plan.** Connected apps are part of some Foldr plans. A site whose plan lacks them shows "Connected apps are not included in your plan" under **Settings > Connected apps**.
2. **Turn connected apps on.** In **Settings > Connected apps**, switch on **Allow connected apps**. It's off by default. While it's off, every address under `/api/v1` answers a bare `404` with `{"error":"not_found"}`, on purpose, so the API looks as though it isn't there. If you get that on a site that should have the API, this is the first thing to check.
3. **Say who may connect apps.** Under **Who may connect apps**, give the permission to the people or groups who should have it. Nobody has it until it's given. Taking it away from someone pauses every app they've connected (their tokens are refused) and giving it back resumes them.
4. **Register your app** (below), either by adding it by hand or by letting it register itself.
5. **Decide what the app may change.** A newly registered app can read and search, and change nothing. To let it make changes, the administrator opens **Policy** for the app and ticks what it may do (see Scopes).

Two ways to register an app

Added by an administrator. On the **Apps** tab, **Add an app**: a name, one or more redirect URLs, and whether the app can keep a secret.

- Redirect URLs must be exact: `https://`, or `http://` only for the same computer (`localhost`, `127.0.0.1`, `[::1]`). The URL your app sends in the sign-in request has to match one of them character for character. (For a loopback URL, as an app on someone's own computer uses, the port may differ.)
- The name and the redirect URLs can be changed afterwards: **Edit** on the **Apps** tab. Plenty of tools can't tell you their callback until they've been given a client ID, so registering first and filling the URL in after is a perfectly normal order to work in. Nothing else about the app changes, and nobody already connected to it is disconnected.
- Tick **This app can keep a secret** for an app that runs on a server, where the secret stays hidden. It makes the app *confidential*: it gets a client ID and a client secret, shown once, and must present the secret at the token endpoint. Leave it clear for an app on someone's computer or phone, which can't keep a secret; that app is *public* and gets only a client ID.
- Either way PKCE is mandatory. A secret is an extra proof, never a substitute for it.

- Make a note of the client ID and, for a confidential app, the secret as soon as they're shown. The secret isn't shown again.

Registered by the app itself. AI assistants and other tools that have to work on any site register themselves the first time someone connects them, using dynamic client registration (RFC 7591). If your app needs to do that:

```
curl -s -X POST https://files.example.com/oauth/register \
  -H "Content-Type: application/json" \
  -d '{
    "client_name": "Acme Reports",
    "redirect_uris": ["https://reports.acme.example/foldr/callback"],
    "token_endpoint_auth_method": "none",
    "grant_types": ["authorization_code", "refresh_token"],
    "response_types": ["code"],
    "scope": "foldr.read offline_access profile"
  }'
```

```
{
  "client_id": "e3a846ddd53b7e70896e7d63d023ffd5",
  "client_name": "Acme Reports",
  "redirect_uris": ["https://reports.acme.example/foldr/callback"],
  "grant_types": ["authorization_code", "refresh_token"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none",
  "scope": "foldr.read offline_access",
  "client_id_issued_at": 1791501531
}
```

Things to know about self-registration:

- The answer is `201`. Registration is anonymous, so it's rate limited: expect `429` if a lot of registrations come from one address or for one site.
- The app is public (`token_endpoint_auth_method` of `none`) unless it asks for `client_secret_post` or `client_secret_basic`, in which case it's confidential and the response carries a `client_secret` once.
- Foldr substitutes rather than refuses where it can: scopes, grant types and response types it doesn't know are dropped, and the response says what was registered. It refuses bad or missing redirect URIs (`invalid_redirect_uri`, at most 10, `https` or loopback only), and a missing, unusable or impersonating name, or any other authentication method (`invalid_client_metadata`). A name can be 1 to 100 characters and can't contain "Foldr" or the site's own title.
- **Self-registration never grants the right to change anything.** Write scopes are never registrable by an app; only an administrator can give them, in the app's policy.
- An administrator can require approval of self-registered apps. Then your app can register, but nobody can use it until an administrator approves it, and the person trying to connect it is told it's waiting. An app nobody approves within 30 days of registering is removed; if it registers again, it waits again.
- The consent page shows the name you registered with, marked as unverified. Choose a name people will recognise.

2.2 Scopes

A scope is one kind of thing your app may do. You ask for scopes in the sign-in request; the person agrees to them on a consent page that describes each in plain words.

Scope	What it allows	Granted by default?
`foldr.read`	Read: shares, listings, metadata, downloads, search, custom fields, versions.	Yes, to any app.
`offline_access`	Receive a refresh token, so the app stays connected without the person signing in again.	Yes, to any app.
`profile`	`GET /me`: the person's name, email address and sign-in name.	Yes, to any app.
`files.write`	Create folders, upload new files, rename, copy. Never replaces, moves away or deletes.	No: an administrator allows it per app.
`fields.write`	Set a custom field value on a file.	No: per app.
`sharing`	Ask to share a file or folder. Every request waits for an approver; the app never gets a link.	No: per app.
`files.overwrite`	Replace a file's contents, and put an earlier version back over it.	No: per app, and a separate decision.
`files.move`	Move files and folders to another folder, in the same share or another.	No: per app, and a separate decision.
`files.delete`	Delete files and folders.	No: per app, and a separate decision.

The last three are *changes that remove or replace*. They are never implied by `files.write`, and a person must agree to each when they connect the app.

What decides whether a change is allowed

For any change, **all** of these have to allow it:

1. **The token's scope.** The call needs the scope for that operation (the table below).
2. **The person.** The app acts as them, so they must be allowed to make that change in Foldr themselves.
3. **The app's policy**, set by an administrator under **Settings > Connected apps > Policy**:
   - which of the three write scopes (**files**, **custom fields**, **sharing**) the app may use;
   - **where it may make changes**: any share the person can write to, or only the shares named;
   - a **daily limit** on write calls for each person (counted in UTC days; a change refused before it is attempted (no permission, a name that is taken, a share outside the policy) doesn't count; one that is attempted and then fails in the storage does, and so does a call refused for being over the limit);
   - for removing and replacing, the three scopes separately, **only in shares named** (there's no "any share" for these), and a **daily limit counted in items** (a folder counts everything in it; the default is 100 items a day for each person).
4. **The share's own switch** for changes that can't be undone: **Allow unrecoverable changes from connected apps**, in the share's settings. It's on by default. If an administrator switches it off, apps can't replace, restore over or delete anything in that share if its storage couldn't bring it back (moves are still allowed, because moving something back undoes it). People using Foldr itself aren't affected. See Recoverability.
5. **Foldr's own protections**, which apply to every app whatever it's allowed: a file someone else has checked out, an item under a legal hold, a published file (never moved or deleted by an app) and the

document of a signing still out are refused.

An administrator can also require **approval** for an app's deletes, moves and version restores. See Operations and approval.

A narrowed policy bites immediately, not when the token expires. If an administrator removes a scope from your app, your next call that needs it is refused (`403 write-not-allowed`), and a refresh hands out a token without it.

Which scope each operation needs

Operation	Scope
`getMe`	`profile`
`listShares`, `getShare`, `listChildren`, `getItem`, `getContent`, `getDownloadUrl`, `search`, `listFields`, `getFieldValues`, `listVersions`	`foldr.read`
`createFolder`, `uploadFile`, `renameItem`, `copyItem`, `createUploadSession`, `completeUploadSession`, `cancelUploadSession`	`files.write`
`setFieldValue`	`fields.write`
`createShareRequest`, `getShareRequest`, `listShareRequests`	`sharing`
`overwriteFile`, `restoreVersion`	`files.overwrite`
`moveItem`	`files.move`
`deleteItem`	`files.delete`
`getOperation`, `listOperations`, `cancelOperation`	the scope of the change that started the operation (`files.write` for a copy, `files.move` for a move, `files.delete` for a held delete, `files.overwrite` for a held restore); the OpenAPI document lists all four as alternatives

A token that lacks the scope gets `403 insufficient-scope`, and its `WWW-Authenticate` header names the scope it needed:

```
WWW-Authenticate: Bearer error="insufficient_scope", scope="sharing"
```

Asking for scopes

Ask only for what the app needs, and ask for `offline_access` if it should stay connected. Foldr grants **less than you ask for, rather than refusing**: a scope the app isn't allowed (an unknown one, a write scope its policy doesn't give it) is dropped, and the consent page shows what's actually granted. If you ask for a change scope, `foldr.read` comes with it. If nothing you asked for is available, the app gets its default read scopes.

So never assume you got what you asked for. **Read the `scope` in the token response**, and treat a later `403 insufficient-scope` or `403 write-not-allowed` as a normal answer.

2.3 The sign-in flow, step by step

1. Discover the endpoints

The site describes itself (RFC 8414):

```
curl -s https://files.example.com/.well-known/oauth-authorization-server
```

```
{
  "issuer": "https://files.example.com",
  "authorization_endpoint": "https://files.example.com/oauth/authorize",
  "token_endpoint": "https://files.example.com/oauth/token",
  "registration_endpoint": "https://files.example.com/oauth/register",
  "revocation_endpoint": "https://files.example.com/oauth/revoke",
  "response_types_supported": ["code"],
  "grant_types_supported": ["authorization_code", "refresh_token"],
  "code_challenge_methods_supported": ["S256"],
  "token_endpoint_auth_methods_supported": ["none", "client_secret_basic", "client_secret_post"],
  "revocation_endpoint_auth_methods_supported": ["none", "client_secret_basic",
"client_secret_post"],
  "scopes_supported": ["foldr.read", "offline_access", "profile"],
  "authorization_response_iss_parameter_supported": true
}
```

`scopes_supported` lists only the scopes any app can have. The change scopes are never advertised, because an app has them only when an administrator has given them to it.

The API is also described as a protected resource (RFC 9728) at `/.well-known/oauth-protected-resource/api/v1`, and every `401` from the API points to it in its `WWW-Authenticate` header. The resource is `https://files.example.com/api/v1`.

2. Make a PKCE verifier and challenge

PKCE is required, with the `S256` method only. Make a random verifier of 43 to 128 characters from `A-Z a-z 0-9 - . _ ~`, and the challenge is the unpadded base64url of its SHA-256:

```
VERIFIER=$(openssl rand -base64 48 | tr '+/' '-_' | tr -d '=\n')
CHALLENGE=$(printf %s "$VERIFIER" | openssl dgst -sha256 -binary | openssl base64 -A | tr '+/' '-_'
| tr -d '=')
```

Keep the verifier with the sign-in attempt; you'll need it in step 4.

3. Send the person to Foldr

Redirect the person's browser (or open the system browser, for an app on their computer) to the authorisation endpoint:

```
https://files.example.com/oauth/authorize
  ?response_type=code
  &client_id=e3a846ddd53b7e70896e7d63d023ffd5
  &redirect_uri=https%3A%2F%2Freports.acme.example%2Ffoldr%2Fcallback
  &scope=foldr.read%20offline_access%20profile%20files.write
  &state=7c1f0e5a9b3d4e28
  &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
  &code_challenge_method=S256
  &resource=https%3A%2F%2Ffiles.example.com%2Fapi%2Fv1
```

- `state` is required. Make it unguessable, keep it, and check it when the person comes back.
- `redirect_uri` must match a registered URL exactly.
- `resource` is optional but recommended: it ties the token to the API, so it can't be used anywhere else (RFC 8707). It must be your site's own `/api/v1` address. Leave it out and the token works on every API route; send it and the token works only under `/api/v1`.
- Only `response_type=code` exists.

The person signs in if they aren't already, sees the app's name and what it asks for in plain words, and chooses **Allow** or **Deny**. Then Foldr sends their browser to your `redirect_uri`:

```
https://reports.acme.example/foldr/callback?
code=fdr_ac_...&state=7c1f0e5a9b3d4e28&iss=https%3A%2F%2Ffiles.example.com
```

Check that `state` is the one you sent, and that `iss` is the site you meant to talk to. The code lives for 60 seconds and works once.

If something's wrong with the request, an app an administrator registered gets `?error=...&state=...&iss=...` on its redirect URL instead (`invalid_request`, `unsupported_response_type`, `invalid_scope`, `invalid_target`, `access_denied`). For a self-registered app the person sees an error page rather than being sent back, because anyone can register a redirect URL, unless this person has already connected that app and not disconnected it: then Foldr trusts the redirect URL and sends errors back as it does for an administrator's app. A client or redirect URL that is not valid is an error page for every app. Either way, an error here tells you something in the request is wrong; fix it rather than retrying.

If the person isn't allowed to connect apps, or the app is waiting for approval, the consent page says so to them. Your app only sees that no code arrives.

4. Exchange the code for tokens

From your server (or, for a public app, from the app itself), `POST` form-encoded to the token endpoint. A public client sends its `client_id`; a confidential client authenticates with HTTP Basic (`client_secret_basic`) or sends `client_id` and `client_secret` in the form (`client_secret_post`), not both:

```
curl -s -X POST https://files.example.com/oauth/token \
  -u "$CLIENT_ID:$CLIENT_SECRET" \
  -d grant_type=authorization_code \
  -d code="$CODE" \
  -d redirect_uri=https://reports.acme.example/foldr/callback \
  -d code_verifier="$VERIFIER"
```

(For a public app, leave out `-u` and add `-d client_id="$CLIENT_ID"`.)

```
{
  "access_token": "fdr_at_...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "scope": "foldr.read offline_access profile files.write",
  "refresh_token": "fdr_rt_..."
}
```

- The body must be `application/x-www-form-urlencoded`. JSON is refused.
- `scope` is what was granted.
- `refresh_token` appears only if `offline_access` was granted.
- Tokens begin `fdr_at_` and `fdr_rt_` so that secret scanners recognise them. Store them as you'd store a password.
- If you send `resource`, it has to match the one in step 3.

5. Call the API

```
curl -s -H "Authorization: Bearer $ACCESS_TOKEN" https://files.example.com/api/v1/me
```


```
{"name":"Alex Morgan","email":"alex@example.com","username":"alex@example.com"}
```


2.4 Token lifetimes

Token	Lifetime
Authorisation code	60 seconds, single use.
Access token	1 hour (`expires_in` says so). Opaque: don't try to read it.
Refresh token	Used once and replaced. Each connection lasts up to 90 days from the person agreeing (an administrator can set 1 to 365 days), and ends sooner after 30 days unused (also the administrator's setting). The lifetime is counted from the first sign-in and rotation doesn't extend it.

When the connection's lifetime or idle limit is reached, the refresh fails with `invalid_grant` and the person has to connect the app again.

2.5 Refreshing

Before the access token runs out (or when a call answers `401 invalid-token`), exchange the refresh token:

```
curl -s -X POST https://files.example.com/oauth/token \
  -u "$CLIENT_ID:$CLIENT_SECRET" \
  -d grant_type=refresh_token \
  -d refresh_token="$REFRESH_TOKEN"
```

The answer has the same shape as step 4, and it contains a **new refresh token**. Refresh tokens rotate:

- **Save the new refresh token and discard the old one**, before you use the new access token.
- A refresh token that's been used already is a sign it leaked. If an already-used one is presented again, the whole chain it belongs to is revoked (the access tokens that came from the same sign-in too), and the person has to connect again. The exception is a deliberate allowance for clients that refresh from two places at once: within 30 seconds of the first use a second attempt is refused with `invalid_grant` but revokes nothing. Even so, serialise your refreshes: one at a time for each connection.

- You may send `scope` with fewer scopes than the connection holds, to get a narrower access token. Asking for a scope the connection never held is `invalid_scope`. The new refresh token keeps the connection's full scope.
- If an administrator has since narrowed your app's policy, the new access token carries only what the app may still do.

2.6 Errors from the token endpoint

The token and revocation endpoints answer in OAuth's own format, not problem documents: a JSON body with an `error` such as `invalid_request`, `invalid_client` (401), `invalid_grant`, `invalid_scope`, `unsupported_grant_type`, or `server_error` (500, a temporary fault: try again). There's deliberately no description. Which check failed is never said, so `invalid_grant` covers an expired code, a wrong verifier, a used code, a revoked connection, a person who no longer exists and a person who's lost the right to connect apps. When you get `invalid_grant` from a refresh, stop refreshing and send the person through the sign-in again.

2.7 Revoking a token

When a person disconnects your app inside your app, tell Foldr (RFC 7009):

```
curl -s -X POST https://files.example.com/oauth/revoke \
  -u "$CLIENT_ID:$CLIENT_SECRET" \
  -d token="$REFRESH_TOKEN" \
  -d token_type_hint=refresh_token
```

- The client authenticates just as at the token endpoint.
- Revoking an **access token** ends that token only.
- Revoking a **refresh token** ends the whole connection: every token the app holds for that person, and the refresh chain. The person has to agree again to reconnect.
- The answer is always `200` with an empty body, including for a token that doesn't exist or isn't yours, so the endpoint can't be used to test whether a token is real.

2.8 What a person's Disconnect does

People see the apps they've connected in Foldr, in the **Connected apps** section of the Security tab of their profile, and can disconnect any of them. Administrators can disconnect any person's connection, or revoke an app for everyone, in **Settings > Connected apps**. Disconnecting:

- ends the connection at once: the next call with any of its tokens is `401 invalid-token`, and the refresh token no longer works;
- withdraws any of the app's changes still held for approval, so nothing waiting is made later (see Operations and approval);
- can be undone only by the person connecting the app again, through the sign-in flow.

A person is also disconnected when their account is deleted, disabled in the directory, or removed by directory sync, with the same result on the next call. And a person who loses the right to connect apps is paused rather than disconnected: their tokens are refused while the right is missing, and work again if it comes back.

2.9 Things that stop an app seeing a share

A token carries no password. A share that opens only with the person's own password (an SMB or WebDAV share set up that way, for instance) can't be mounted on a token's behalf, so it doesn't appear in what the app sees. This is why an app can show fewer shares than the person sees in Foldr's own web app.

Some shares live in the person's own Google Drive, OneDrive, Dropbox or Box account. Foldr reaches those with a separate sign-in to that service, which the person gives once, in Foldr's web app: by opening the share there (Foldr asks them to sign in to the service) or under **Me › Services**. Until they have, opening such a share through the API answers `409 storage-account-not-connected`. Your app can't make that connection for them, and the sign-in to Foldr above doesn't make it either.

3.0 Conventions

The rules that hold across the whole API. Most of the surprises people meet are on this page.

3.1 Requests and responses

- **Send the token** on every call: `Authorization: Bearer <access token>`. There's no other way to authenticate. A browser's Foldr cookie is not accepted and gets the same `401` as no token.
- **JSON bodies** are sent as `Content-Type: application/json` and are limited to 1 MB. File contents are the exception: an upload or overwrite body is the file's raw bytes (`application/octet-stream`), whatever its size.
- **JSON answers** come as `application/json`. Every error is `application/problem+json` (see Errors).
- **Shared caches never store a response.** An answer carrying an `ETag` is `Cache-Control: private, no-cache`: the client's own cache may keep it but must ask again with `If-None-Match` before using it. Every other answer is `no-store`. The OpenAPI document and the problem-type pages are the exception: public, and cached for a short while.
- **Ignore what you don't know.** New fields and new header values can appear within v1. Don't fail on them.

3.2 Ids

- **Shares, custom fields, share requests and operations have UUIDs** (for example `01a11dce-14f0-7f23-b857-c621356097a0`). They're always UUIDs, on Foldr Cloud and on a Foldr server alike. Treat them as opaque strings anyway; compare them as text, in lower case.
- **Files and folders have no id.** An item is its share's id plus its path inside the share. If a file is renamed or moved, its path changes, and the response tells you the new one. (If Foldr ever adds stable item ids, an `id` field will be added to items; that's an additive change.)
- **People** are identified by `id` in a few responses (the person who has a file checked out, the person who last set a field value). That's their directory id, an opaque string.
- **Version ids and upload session ids** are opaque strings. Never build one.

3.3 Paths

Files and folders are addressed by a share id and a path from the top of the share.

Where the path goes. The path **never goes in the URL path**. It's a query parameter, `?path=`, on reads, on the raw-bytes upload (`uploadFile`), on `overwriteFile` and on `deleteItem`, and a JSON `"path"` field on everything else. The reason is that proxies and frameworks quietly rewrite `%2F`, `//` and `..` in URL paths, and file names can legally contain `#`, `?` and `%`.

Encoding. Percent-encode the value of `path` as you would any query value, with your HTTP library's own encoder: a name such as `/Invoices/Q3 & Q4 #1 100%.txt` has to travel as `%2FInvoices%2FQ3%20%26%20Q4%20%231%20100%25.txt` (a `/` may also be left as it is). With curl, let it do the work:

```
curl -s -G -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/shares/$SHARE/item" \
```

```
  --data-urlencode "path=/Invoices/Q3 & Q4 #1 100%.txt"
```

Give a parameter only once. `?path=/a&path=/b` is refused with `400 invalid-request`, as is a list.

The forms you may send.

- The leading `/` is optional in a request: `Reports/Q3.xlsx` and `/Reports/Q3.xlsx` mean the same.
- `/`, an empty `path` or no `path` at all is the top of the share, where the operation takes a folder. A path made only of spaces is refused (`400 invalid-path`); it's never quietly read as the top.
- Nothing is trimmed. A name that begins or ends with a space is addressed, and returned, with it.
- At most 4,096 bytes.
- Refused with `400 invalid-path` before anything is looked up: a `..` segment, a control character, a backslash, an absolute-URL or drive form (`smb://x`, `C:\x`, `//server/x`), a path of only spaces. A new name (a folder, a rename, a copy's or upload's name) is a single name: no `/`, no control characters, and no space at either end.

The form you get back. Every response gives the item's **canonical path**: exactly as the storage holds it (its own upper and lower case), in Unicode NFC, always beginning with `/`. `/` alone is the top of the share. Keep that form and use it in later calls, in `If-Match` bookkeeping and in your own records. Two storage details:

- *Case.* Storage that ignores case (file servers, for one) finds an item whatever case you send and answers with the stored case. Storage that is case sensitive (S3, for one) matches exactly.
- *Unicode.* macOS tends to produce names in decomposed form (NFD). You can send either; Foldr converts the request to NFC. If a folder holds two names that differ only in how an accented letter is encoded, a path that could mean either is `409 ambiguous-path`.

Names may be tidied. A name you supply for something new is made safe for the storage: characters the storage forbids are replaced, runs of spaces are collapsed, trailing dots and dashes are removed and a very long name is shortened. Reserved device names and format characters are refused. So always use the `path` in the answer, never the name you sent, for every create, copy, move and rename. A name that already exists is addressed as stored, even if a new name like it would have been changed.

After a change. A rename or move answers with the item at its *new* canonical path. The old path names nothing afterwards, except where the storage ignores case and the old spelling is just another way to write the new one (a rename that only changes case). The case in a canonical path comes from cached folder listings, so after a case-only change made outside Foldr it can be a minute out of date.

3.4 Conditional requests: ETags

Every item has an `etag`, and the same value comes in an `ETag` header when you read an item on its own, download it, or change it.

- **An ETag is an opaque string.** Send it back exactly as you received it, including any `W/` and the quotation marks. It may be a weak tag (a hash of size and modified time) or the storage's own tag; which depends on the storage. A weak tag is a best-effort change detector: two writes of the same size within the same second can share one, and a folder's tag follows the folder's own timestamp, not its contents. Where you need to be sure, read the item on its own with `getItem` first and send the tag it gives. A listing may carry one form and a read of the item on its own another: either is accepted when you send it back as `If-Match`.

- **Reading with `If-None-Match`.** Send an item's ETag as `If-None-Match` to `getItem`, `getContent` or `getFieldValues`. If nothing has changed, you get `304` with no body, and the call costs 1 on your rate budget whatever the read would have cost. (`If-None-Match` compares weakly, as HTTP says it should. A list of tags, or `*`, is accepted.)
- **Changing with `If-Match`.** A call that changes something that already exists says *which version it means* with `If-Match: <etag>`. The operations that need it are `renameItem`, `overwriteFile`, `setFieldValue`, `moveItem`, `deleteItem` and `restoreVersion`.
  - Without it: `428 precondition-required`.
  - With a tag that no longer matches: `412 precondition-failed`. Someone changed the item since you read it. Read it again, decide again, and don't just retry.
  - For overwrite, move, delete and restore, `If-Match: *` is refused (`400 invalid-request`): name the version you mean. (`setFieldValue` accepts `*`, meaning "whatever the values are now".)
  - A list of tags matches if any member does.
- **Creating with `If-None-Match: *`.** `uploadFile` creates a file and never replaces one, and says so in HTTP's own words: `If-None-Match: *` is required (`428` without it). If the name is taken: `412 precondition-failed`.
- A write's answer carries the item's new ETag. Use it for the next change instead of reading again.
- A search hit comes from the search index, so its `etag` and `checked_out` are `null`. `getItem` gives the live ones.
- The ETag of a file's *field values* is a different tag, over the whole set of values; it's used with `getFieldValues` and `setFieldValue` only.

3.5 Idempotency keys

Networks drop answers. If you create something and never see the reply, you can't tell whether it happened. An `Idempotency-Key` lets you send the same request again safely.

Which calls take one. All of these **require** it (without one: `400 invalid-request`, and nothing runs): `createFolder`, `uploadFile`, `copyItem`, `moveItem`, `deleteItem`, `restoreVersion` and `createShareRequest`. The others don't take one, because `If-Match` already protects them (`renameItem`, `overwriteFile`, `setFieldValue`) or they're safe to repeat.

The key. A value of your own, 1 to 255 visible ASCII characters (a UUID is ideal), new for every new request. Keys belong to your app and the person it acts for, so nobody else's keys can collide with yours.

What the server does.

You send	You get
A key it hasn't seen	The request runs, once. A successful (`2xx`) answer is kept for **24 hours**.
The same key and the same request again, within 24 hours	The first answer again, with the header `Idempotent-Replayed: true`. Nothing is done a second time, and a repeat doesn't use up your daily limit.
The same key with a *different* request (another path, another query, other bytes, another body)	`422 idempotency-key-reused`.

You send	You get
The same key while the first request is still being handled	`409 conflict`. Wait a moment and send it again.
The same key after the first request was cut off part way (a server restarting, say)	The key, and any name it was writing, are free about two minutes after the request last made progress; send the same request again and it runs.
A key whose first request was *refused* or failed (any answer that isn't `2xx`)	It runs again: a refused request changed nothing, so the key is free.

The one refusal that is kept like a success is `409 source-remains` for a move to another share, because that move did change things (the copy was made); sending it again tells you the same and doesn't copy a second time.

An upload is matched on its bytes too: Foldr reads the repeated body to be sure it's the same file. Other bytes of the same length under the same key are `422`.

After 24 hours. The kept answer is forgotten.

- For a request that can become an operation (`copyItem`, `moveItem`, `deleteItem` and `restoreVersion`) the key still finds the operation: one still under way answers `202` again, and one that has finished answers what the request would have answered if it had made the change at once (`201` with the copy, `200` with the item or the deletion) or the problem it ended with, carrying the operation.
- For `createFolder`, `uploadFile` and `createShareRequest` the key is simply forgotten and the request runs as a new one. Nothing is ever replaced, so the usual result is `409 conflict` or `412 precondition-failed` because it already exists; but with `on_conflict=rename` you would get a second copy under a new name. So retry soon, within the day, and use the default `on_conflict` (`fail`) when you rely on a key.

3.6 Pagination

Collections come as a page:

```
{ "items": [ ... ], "next_cursor": "opaque-string-or-null", "truncated": false }
```

- Ask for the next page by sending `next_cursor` back as `?cursor=`. When `next_cursor` is `null` there are no more.
- **A cursor is opaque.** Never build one or change one, and don't keep one for later; its form can change within v1. Send it back with the same other parameters (the same `path`, the same search).
- `limit` sets the page size, from 1. A `limit` over the largest allowed is read as the largest; `0`, or something that isn't a whole number, is refused (`400 invalid-request`).

Collection	Default `limit`	Largest
`listShares`	100	200
`listChildren`	100	200
`search`	20	50

Collection	Default `limit`	Largest
`listOperations`	50	200
`listShareRequests`	50	200

- `listVersions` puts all of a file's versions on one page. `listFields` and `getFieldValues` return a plain array with no paging.
- A cursor that isn't one Foldr issued for that collection is `400 invalid-cursor`. A listing cursor whose snapshot has gone is `410 cursor-expired`: start again from the first page.
- **Folder listings are a snapshot.** The first page takes a snapshot of the folder and keeps it for a minute while you page through it. A folder that changes in the meantime shows the snapshot, so don't use a listing to decide whether something exists now; use `getItem`.
- **Very big folders.** One listing reads at most 25,000 entries. A folder with more is paged as its first 25,000, and **every** page of it says `"truncated": true`. A `null` `next_cursor` on a truncated listing is therefore not "the whole folder". `truncated` is always present and `false` otherwise.
- **Search pages** walk the person's search indexes one after another, so a page can hold fewer hits than `limit` and still not be the last. `total_estimate` is only an estimate.

3.7 Timestamps

RFC 3339, in UTC, to the second: `2026-10-08T23:17:11Z`. A timestamp can be `null` where the storage doesn't report one (`modified` on some storage). Daily limits reset at 00:00 UTC.

3.8 Rate limits

Every call takes some tokens out of a bucket that refills steadily. There are two buckets, and a call is charged to both or to neither:

- your app's budget for **each person**, by default 120 tokens refilling at 2 a second;
- your app's budget **across the organisation**, by default 1,200 tokens refilling at 20 a second.

So you can burst, and then sustain a rate. These numbers are defaults, and a Foldr server's administrator can change them.

What a call costs. Each operation's `x-foldr-rate-cost` in the OpenAPI document says:

Cost	Operations
1	`getMe`, `listShares`, `getShare`, `listChildren`, `getItem`, `listFields`, `getFieldValues`, `listOperations`, `getOperation`, `listShareRequests`, `getShareRequest`
2	`getDownloadUrl`, `listVersions`, `createFolder`, `renameItem`, `copyItem`, `setFieldValue`, `createShareRequest`, `cancelOperation`, `createUploadSession`, `completeUploadSession`, `cancelUploadSession`
3	`search`
5	`getContent`, `uploadFile`, `overwriteFile`, `moveItem`, `deleteItem`, `restoreVersion`

A `HEAD` costs 1 whatever its `GET` costs, and so does a `304`.

Headers. Every answer, a success or a refusal, carries `X-RateLimit-Limit` (the size of the bucket that would run out first for you) and `X-RateLimit-Remaining` (what's left in it). Use them to slow down background work before you're refused.

When you're over budget: `429` with a problem document of type `rate-limited` and a `Retry-After` header, in seconds.

```
HTTP/2 429
Retry-After: 3
X-RateLimit-Limit: 120
X-RateLimit-Remaining: 0
Content-Type: application/problem+json
```

Wait at least that long. If several requests are waiting, don't send them all at once afterwards; space them out, and back off further on repeated `429`s. Treat `429` as flow control, not as a failure to show a person.

Other limits that answer 429:

- By default at most **4 uploads in flight** at once for each person through your app (`uploadFile` streams the file through a server process for as long as it takes): `rate-limited`, with `Retry-After: 10`.
- At most **20 copies waiting** for each person through your app, with two running at a time: `rate-limited`. Two copies also run at a time across the whole site, so a copy can wait behind other people's.
- The **daily limits** an administrator sets for your app: `daily-write-limit` for writes, `destructive-budget` for removing and replacing. `Retry-After` says how many seconds until the limit resets at midnight UTC.
- A very heavy flood from one network address is stopped before it reaches the API, with an HTML `429` and no `Retry-After`. A client that parses every `429` as JSON should check the content type first.

3.9 HEAD

Every read (`GET`) also answers `HEAD`: the same status and headers as the `GET`, with no body, for a cost of 1. `HEAD` on `getContent` answers from the file's details and doesn't read the file. Writes don't answer `HEAD`.

3.10 405 and 404

- **An address that exists but doesn't answer that method** is `405 method-not-allowed` with an `Allow` header naming the methods it does answer. For example, `PATCH` on a file's content answers `Allow: GET, HEAD, PUT`. `OPTIONS` is answered `405` everywhere: the API sends no CORS headers, because it's meant to be called from servers.
- **An address that isn't the API's** is `404 not-found`.
- **Without a valid token**, both of these are the `401` first. You find out nothing about the API's shape before you've signed in.
- **`404 not-found` is deliberately blunt.** It answers a share or path that doesn't exist, one that exists but isn't the person's, one that a share's path rules hide, and a share outside the shares your app's policy names. They look the same on purpose, so that an app can't use the API to find out what it isn't allowed to see. Don't build logic that tries to tell them apart.

- The exceptions are two true answers about the person's *own* shares: `503 share-unreachable` (a file server that's down) and `409 storage-account-not-connected` (a cloud storage account of theirs that they haven't yet connected in Foldr's web app).

4.0 Errors

Every error from `/api/v1` is a problem document (RFC 9457), with `Content-Type: application/problem+json`:

```
{
  "type": "https://files.example.com/api/v1/problems/precondition-failed",
  "title": "Precondition failed",
  "status": 412,
  "code": "PRECONDITION_FAILED",
  "detail": "It has changed since you read it.",
  "instance": "#a1b2c3d4"
}
```

Member	
`type`	What went wrong, as a URL on the site: `https://<site>/api/v1/problems/<slug>`. Always present.
`title`	A short summary of the type. Always present.
`status`	The HTTP status. Always present.
`code`	The same condition as an upper-case code (`PRECONDITION_FAILED`), in the style Foldr's other APIs use. Always present.
`detail`	More about this occurrence, where there's something safe to say. May be absent or `null`.
`instance`	A fault code (`#` and 8 hexadecimal digits) for this occurrence, where there is one. Quote it to support if you ask for help.
`operation`	Present only on a problem that is how an operation ended: that operation as it stands.

Branch on `type` (or `code`) and the HTTP status. Never branch on `title` or `detail`, which are for people and can be reworded. New problem types can appear within v1, so keep a sensible default for an `type` you don't know: handle it by its status class (a `4xx` you can't fix by retrying, a `503` you can).

The problem-type pages. Each `type` URL resolves, with no token, to a short page saying what the problem means and what to do about it. It's HTML for a browser, and JSON if you ask with `Accept: application/json` (it adds a `description` member). An unknown slug is `404 not-found`.

The API answers in problem documents everywhere, including a `401` for a missing token and a `429` from a rate limiter. The two exceptions are the OAuth endpoints (`/oauth/token`, `/oauth/revoke`, `/oauth/register`), which answer in OAuth's own `{"error": "..."}` form (see Getting access), and the bare `{"error":"not_found"}` that a site with connected apps switched off gives for every address.

Headers that go with errors. `Retry-After` (seconds) accompanies every `429` and every `503`, and most operations' `202`. `WWW-Authenticate` accompanies `401 invalid-token` and `403 insufficient-scope`. `Allow` accompanies `405`.

4.1 Every problem type

The "what to do" column is advice for a client. "Nothing was changed" means what it says: where a refusal leaves a change undone, a client may treat the request as not made.

400: the request is wrong

Sending it again unchanged will fail the same way.

Type	Meaning	What to do
`invalid-request`	The request couldn't be read: malformed JSON, a body over 1 MB, a parameter given twice or as a list, a value of the wrong kind (a `limit` that isn't a whole number, or `0`), a missing or invalid `Idempotency-Key`, `If-Match: *` where a version has to be named, or `If-Match` on `uploadFile`.	Fix the request. The `detail` usually says which part.
`invalid-path`	The path isn't a path inside the share (see Conventions), or a new name isn't a single valid name. Nothing was looked up.	Fix the path or name.
`invalid-cursor`	The cursor isn't one Foldr issued for this collection and query.	Use `next_cursor` exactly as returned, with the same other parameters.
`copy-into-itself`	A folder can't be copied into itself or into a folder inside it.	Choose a destination outside it. Nothing was copied.
`move-into-itself`	A folder can't be moved into itself or into a folder inside it.	Choose a destination outside it. Nothing was moved.

401: no valid token

Type	Meaning	What to do
`invalid-token`	The access token is missing, expired, revoked, or not for this API. Every token failure is this one answer, with no reason, on purpose.	Refresh the token and try once more. If refreshing fails, send the person through sign-in again. The `WWW-Authenticate` header says where the authorisation server is described.

403: not allowed

Type	Meaning	What to do
`insufficient-scope`	The token is valid but wasn't granted the scope this operation needs. `WWW-Authenticate` names it.	Ask for the scope next time the person signs in (if the app's policy allows it). Don't loop.
`write-not-allowed`	The change isn't allowed: the person may not make it there (no write permission, a file published as an approved version they may only read, a field they may not set, a kind of sharing they may not ask for), or the organisation doesn't allow this app to make this kind of change.	Tell your user the change isn't available to this app here. Retrying won't help.
`download-blocked`	A label on the file stops it being downloaded (or, for a copy or move into another share, stops it leaving its share).	Tell your user. Retrying won't help.

Type	Meaning	What to do
`legal-hold`	The item, a folder it's in, or something inside the folder is under a legal hold. Nothing under a hold can be replaced, moved, renamed, restored over or deleted, by anyone, until the hold is released in Foldr.	Tell your user. Nothing was changed.
`needs-approval-not-available`	The organisation requires approval for this app's changes, and replacing a file's contents can't be held for approval, so the app can't replace files.	Don't offer overwrite to this app's user; use another route (the person can replace the file in Foldr). Deletes, moves and version restores are offered, each waiting for an approver.
`change-declined`	A change that was held for approval was declined by an approver. Nothing was changed.	Show it as declined. Ask again only if the person wants to. Seen when you poll the operation.
`ai-content-blocked`	Reserved: a label keeps the file away from AI tools. No v1 operation raises it today, because the API calls no AI model.	Handle it like `download-blocked` if you ever see it.

404 and 405

Type	Meaning	What to do
`not-found`	There's nothing here the person can see: no such share, nothing at that path, no such field or operation. It's the same answer whether the thing doesn't exist or isn't theirs.	Don't try to tell the cases apart.
`method-not-allowed`	The address exists but doesn't answer that method. `Allow` lists the methods it does.	Fix the method.

409: the thing is in a state that doesn't allow this

Type	Meaning	What to do
`conflict`	Something with that name is already there, or the item already has that name. Nothing is ever replaced. Also: the name is being written right now, or a request with the same `Idempotency-Key` is still being handled.	Choose another name, or send `on_conflict: "rename"`. For an in-progress request, wait and send it again.
`not-a-folder`	The path names a file where a folder was expected.	Fix the path.
`not-a-file`	The path names a folder where a file was expected.	Fix the path.
`ambiguous-path`	Two items in the folder have names that differ only in how their accented characters are encoded, so the path could mean either.	Take the exact path from a listing.

Type	Meaning	What to do
`not-downloadable`	A Google document type that can't be downloaded (a Google Form). Its item's `export` is `null`.	Open it in Google.
`no-direct-download`	The share doesn't offer direct downloads.	Use `getContent`.
`no-direct-upload`	The share doesn't take uploads straight to its storage.	Use `uploadFile`.
`upload-incomplete`	You completed an upload session but the storage doesn't hold the declared file (nothing there, or not that many bytes). The session stays open.	Finish sending the file, then complete again; or cancel.
`storage-account-not-connected`	The share is in the person's own cloud storage account (Google Drive, OneDrive, Dropbox, Box) and they haven't yet given Foldr their sign-in to that service.	Ask the person to open the share in Foldr's web app, or go to **Me › Services**, and connect the account. Then try again.
`approved-version-unavailable`	The file is published as an approved version, which is what this person is given, and that version can't be read from storage. Asking again won't help.	Someone who manages the share has to publish the file again.
`no-approver`	A share request, or a change that needs approval, would need someone other than the person to approve it, and there is nobody. Nothing was saved or changed.	An administrator has to name approvers for the share.
`already-shared`	The person already shares that item, or a request to share it is waiting. An app only starts sharing.	Leave it; changes to an existing share are made in Foldr.
`copy-incomplete`	A copy finished but some of the folder's items couldn't be copied. What was copied is left in place.	The operation's `failures` list names what wasn't copied. Copy those yourself, or remove the partial copy and ask again.
`operation-cancelled`	The operation was cancelled before it finished. Whatever a cancelled copy or move had made was removed, and a cancelled move's original is where it was. A change held for approval that was withdrawn changed nothing.	Show it as cancelled.
`operation-finished`	You tried to cancel an operation that has already finished.	Read the operation for how it ended.
`published`	The file is published as an approved version (or the folder holds one). An app can't move or delete a published file.	Someone who manages the share has to unpublish it in Foldr first.

Type	Meaning	What to do
`envelope-source`	The file is the document of a signing request still out for signature (or the folder holds one). An app can't replace, move or delete it.	Wait until the signing is finished or cancelled.
`folder-too-large`	A folder has more than 1,000 files and folders, more than an app may move or delete in one request (or, for a move to another share, is nested more than 64 folders deep).	Move or delete the folders inside it separately. Nothing was changed.
`not-recoverable-here`	The share's storage can't undo this change and its administrator doesn't let apps make changes there that can't be undone.	Tell your user the change isn't available to apps on this share. People can still make it in Foldr. Nothing was changed.
`source-remains`	A move to another share copied everything, and the copy is whole, but the original (or some of a folder's items) couldn't then be removed. It is now in both places. Nothing is lost.	Check the copy, then delete what's left at the original place yourself. Moving it again would make a second copy.
`change-expired`	A change held for approval wasn't decided in time (14 days unless the organisation says otherwise). Nothing was changed.	Ask again if it's still wanted.
`requester-not-connected`	A change was approved, but the share couldn't be opened as the person who asked (their own sign-in to the storage isn't available now, or they can no longer reach the share). Nothing was changed.	Ask for the change again.

410, 412, 413, 422, 423, 428

Type	Status	Meaning	What to do
`cursor-expired`	410	The listing behind this cursor is no longer held.	Start again from the first page.
`precondition-failed`	412	`If-Match` didn't name the item (or its field values) as it is now; or `If-None-Match: *` on an upload where the name is taken.	Read the item again and decide again. For an upload, choose another name or use `on_conflict=rename`.
`changed-since-requested`	412	A change held for approval was approved, but the item changed after you asked: the `If-Match` you sent no longer names it. Nothing was changed.	Read the item again and ask again if the change is still wanted.
`too-large`	413	There isn't enough storage left in the share for that (a declared size over quota, or over a storage limit such as a Dropbox direct upload's 150 MB).	Free space, or send less.

Type	Status	Meaning	What to do
`copy-too-large`	413	A folder holds more files and folders than one copy takes (10,000) or is nested more than 64 folders deep.	Copy the folders inside it separately. Nothing was copied.
`validation-failed`	422	A value can't be used as given: a field value that doesn't suit the field, a recipient who can't be found, a sharing term that isn't allowed. Nothing was changed.	Fix the value.
`idempotency-key-reused`	422	This `Idempotency-Key` was used within 24 hours for a different request.	Use a new key for each new request; repeat a key only with the same request.
`checked-out`	423	Someone else has the item (or a file inside the folder) checked out in Foldr.	Try again later; it can be changed once they check it in.
`precondition-required`	428	The change needs an `If-Match` header (or, for `uploadFile`, `If-None-Match: *`).	Read the item, then send the change with its ETag.

429: slow down or wait

Type	Meaning	What to do
`rate-limited`	Too many calls, too many uploads in flight (4 for each person through your app), or too many copies waiting (20).	Wait `Retry-After` seconds.
`daily-write-limit`	The app has reached its daily limit of changes for this person. `Retry-After` is the time until it resets at midnight UTC.	Stop making changes for that person until then.
`destructive-budget`	The app has used its daily allowance of replacements, moves and deletions for this person (counted in items; a folder counts everything in it), or this change alone would go past it. Nothing was changed.	Wait for `Retry-After`, or do less.

5xx: not your fault

Type	Status	Meaning	What to do
`internal-error`	500	Something went wrong in Foldr.	Try again later. If it keeps happening, quote `instance` to support.
`share-unreachable`	503	The share's file server couldn't be reached just now.	Try again after `Retry-After`.
`share-unavailable`	503	The share couldn't be opened just now.	Try again after `Retry-After`.
`write-failed`	503	The storage didn't complete the change just now.	**Read the item again before trying again**: the change may or may not have been made. With an `Idempotency-Key`, sending the same request again is also safe.

Type	Status	Meaning	What to do
`copy-interrupted`	503	A copy or move was stopped by the server more often than it's restarted, or waited too long to start (a day).	Read the destination before asking again: part of it may have been copied. A move stopped while copying hasn't touched the original.

4.2 Which statuses can an operation give?

The OpenAPI document lists, for each operation, every status it can answer and the problem types behind each. Any operation can also answer `400 invalid-request`, `401 invalid-token`, `403 insufficient-scope`, `429 rate-limited` and `500 internal-error`; the document includes those for you.

5.0 Reading

Everything on this page needs the `foldr.read` scope (and `profile` for `getMe`), and works on exactly what the person can see in Foldr.

5.1 Who am I: `getMe`

```
curl -s -H "Authorization: Bearer $TOKEN" $FOLDR/api/v1/me
```

```
{"name":"Alex Morgan","email":"alex@example.com","username":"alex@example.com"}
```

`name` is the display name (or the sign-in name where there's none), `email` is `null` where the directory has none, and `username` is the sign-in name. Needs the `profile` scope.

5.2 Shares: `listShares`, `getShare`

A **share** is a storage location the person can read: a folder on a file server, a bucket, a SharePoint library, a Google Drive, and so on.

```
curl -s -H "Authorization: Bearer $TOKEN" "$FOLDR/api/v1/shares?limit=50"
```

```
{
  "items": [
    {
      "id": "01a11dce-14f0-7f23-b857-c621356097a0",
      "name": "Finance",
      "type": "smb",
      "indexed": true,
      "read_only": false,
      "reachable": true
    }
  ],
  "next_cursor": null,
  "truncated": false
}
```

Field	
`id`	The share's id, always a UUID. Use it in every other call.
`name`	The name people see in Foldr.
`type`	The kind of storage: `smb`, `local`, `hosted`, `s3`, `b2`, `azure_blob`, `sftp`, `ftp`, `webdav`, `edge`, `google_drive`, `google_shared_drive`, `group_drive`, `onedrive`, `sharepoint`, `teams`, `dropbox`, `box` or `other`. The list is open: **treat a value you don't know as `other`.**
`indexed`	Whether the share can be searched with `search`.
`read_only`	`true` when the person can't change anything in the share.

Field	
`reachable`	`false` when the share's file server couldn't be reached just now. It still exists and is the person's; try again in a few minutes. Unreachable shares are listed after the rest.

- `getShare` (`GET /shares/{share_id}`) returns one share. A share that doesn't exist and a share that isn't the person's get exactly the same `404`.
- A share that opens only with the person's own password isn't listed, because a token carries no password. See Getting access.
- A share the person can read isn't necessarily one your app may change: check `read_only`, and remember that an app's policy decides where it can make changes.

5.3 Listing a folder: `listChildren`

```
curl -s -G -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/shares/$SHARE/children" \
  --data-urlencode "path=/Reports" --data-urlencode "limit=100"
```


```
{
  "items": [
    {
      "share_id": "01a11dce-14f0-7f23-b857-c621356097a0",
      "path": "/Reports/2026",
      "name": "2026",
      "kind": "folder",
      "size": null,
      "modified": "2026-10-08T23:16:47Z",
      "mime_type": null,
      "export": null,
      "etag": "W/\"d-1791501407\"",
      "read_only": false,
      "checked_out": null
    },
    {
      "share_id": "01a11dce-14f0-7f23-b857-c621356097a0",
      "path": "/Reports/Q3.xlsx",
      "name": "Q3.xlsx",
      "kind": "file",
      "size": 48213,
      "modified": "2026-10-07T09:02:11Z",
      "mime_type": "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
      "export": null,
      "etag": "\"3a5f9c1d7e\"",
      "read_only": false,
      "checked_out": null
    }
  ],
  "next_cursor": null,
  "truncated": false
}
```


Folders come first, then everything by name. `path` omitted, empty or `/` is the top of the share. A file instead of a folder is `409 not-a-folder`.

An item has these fields, all always present:

Field	
`share_id`, `path`, `name`	Where it is. `path` is the canonical path (see Conventions); `name` is its last segment.
`kind`	`file` or `folder`.
`size`	Bytes. `null` for a folder, and for an item with no fixed size (a native Google document).
`modified`	When it was last modified, where the storage says; otherwise `null`.
`mime_type`	The storage's media type, or one guessed from the name. `null` for a folder.
`export`	For a native Google document that can be downloaded: `{ "mime_type": ..., "extension": ... }`, what `getContent` will deliver (a Google Doc comes down as a `docx`). `null` for everything else.
`etag`	The item's entity tag. See Conventions.
`read_only`	`true` when the person can't change things in this share.
`checked_out`	Who has the item checked out, as `{ "by": { "id", "name" }, "since" }`, or `null`.

Paging, the one-minute snapshot and the 25,000-entry limit are described under Pagination. Two things to remember in particular: a listing is a snapshot, so use `getItem` for the truth about one item; and a `truncated: true` listing is not the whole folder.

5.4 One item: `getItem`

```
curl -s -i -G -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/shares/$SHARE/item" --data-urlencode "path=/Reports/Q3.xlsx"
```

The answer is the item above, with its ETag also in the `ETag` header. This is the call to make before you change something: it gives the live ETag. Send that ETag back as `If-None-Match` on a later call to get `304` while it hasn't changed.

5.5 Downloading: `getContent`, `getDownloadUrl`

```
curl -s -H "Authorization: Bearer $TOKEN" -o Q3.xlsx -G \
  "$FOLDR/api/v1/shares/$SHARE/content" --data-urlencode "path=/Reports/Q3.xlsx"
```

The body is the file's bytes, with the file's own `Content-Type`, `Content-Disposition: attachment; filename*=...` (UTF-8) and its `ETag`.

- **Ranges.** Send `Range: bytes=0-1023` and, where the storage can seek, you get `206` with `Content-Range`. Where it can't, you get `200` with the whole file. Use ranges to resume an interrupted download. A range that runs past the end is cut short, as HTTP says.
- **Conditional.** `If-None-Match` with the file's ETag answers `304` while it's unchanged.
- **A folder** is `409 not-a-file`.
- **A label that blocks download** is `403 download-blocked`. This is checked before any redirect.

Redirects to direct download

On a share set up for direct downloads (a setting on the share, which Foldr Cloud turns on for storage that supports it), `getContent` answers `302` with a `Location` pointing straight at the storage, so the bytes don't pass through Foldr. That URL is a short-lived credential of its own.

- **Follow the redirect without your Foldr token.** Your HTTP client must not send `Authorization` to the storage host; most clients drop it for a redirect to a different host, but check yours (curl's `-L` does; `--location-trusted` doesn't).
- **Don't log the URL or hand it to anyone else.**
- `?direct=false` keeps the bytes coming through Foldr instead.
- A redirect counts as the download.

If your platform can't follow a redirect to another host (some connector tools can't), ask for the URL yourself with `getDownloadUrl` and fetch it as a separate step:

```
curl -s -G -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/shares/$SHARE/download-url" --data-urlencode "path=/Reports/Q3.xlsx"
```

```
{ "url": "https://storage.example.net/...", "expires_at": "2026-10-08T23:32:11Z" }
```

`expires_at` is a floor: the URL works until at least then, and some storage keeps it working longer. A share without direct downloads answers `409 no-direct-download`; use `getContent` there. The same download rules apply (`download-blocked`, and so on).

Native Google documents

A Google Doc, Sheet, Slides or Drawing has no bytes of its own. `getContent` exports it: the answer's `Content-Type` is `export.mime_type` and `Content-Disposition` names it with `export.extension` (`Minutes.docx`). Because the length isn't known until it has been sent, there's **no `Content-Length` and no range support** (`Accept-Ranges: none`; a `Range` is answered `200`, whole), and it always comes through Foldr, never as a redirect. A Google type with no export, a Form, is `409 not-downloadable`. Its `name` and `path` stay as stored, without the extension.

5.6 Search: `search`

```
curl -s -G -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/search" \
  --data-urlencode "q=quarterly forecast" --data-urlencode "limit=20"
```

```
{
  "items": [
    {
      "item": { "share_id": "...", "path": "/Reports/Q3.xlsx", "name": "Q3.xlsx", "kind": "file",
                "size": 48213, "modified": "2026-10-07T09:02:11Z",
                "mime_type": "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
                "export": null, "etag": null, "read_only": false, "checked_out": null },
      "score": 7.42,
      "snippet": "... the quarterly forecast for Q3 assumes ..."
    }
  ],
  "next_cursor": "...",
```

```
  "total_estimate": 134,
  "not_searched": [
    { "share_id": "...", "name": "Archive", "reason": "unreachable" }
  ]
}
```

- `q` is required: the words to search for, at most 500 characters. `share_id` limits the search to one share. `limit` is 20 by default and at most 50.
- It's a keyword search over the shares the person can see, and respects their permissions. Only **indexed** shares are searched (`indexed` on the share). It does not call an AI model.
- A hit's `item` comes from the search index, so its `etag` and `checked_out` are `null`; call `getItem` for the live values. `score` is only comparable within one response. `snippet` is a short passage around the match, where there is one and it may be shown.
- **Pages.** Pass `next_cursor` back with the same `q` and `share_id`. Where the person's shares sit in more than one search index, pages walk the indexes one after another, so a page can have fewer hits than `limit` and still not be the last. `total_estimate` is only an estimate.
- `not_searched` names shares the search couldn't cover and why: `unreachable` (its file server is down), `not-indexed` (it isn't indexed) or `index-unavailable` (its search index didn't answer). The list of reasons is open. If a search comes back thin, check this before concluding that nothing matches.
- `search` costs 3 on your rate budget.

5.7 Custom fields: `listFields`, `getFieldValues`

Foldr lets organisations record extra information (custom fields) on files. `listFields` (`GET /fields`) returns, as a plain array, the fields the person can see:

```
[
  { "id": "5b3f...", "name": "Status", "type": "select", "options": ["Draft", "Approved"],
    "multiple": false, "read_only": false }
]
```

`type` is one of `string`, `text`, `number`, `bool`, `date`, `location`, `select`, `signature`, `delegate`; the list is open. `options` is the list of allowed values where the field has one, otherwise `null`. `read_only` is `true` when the person can't set the field on any of their shares.

`getFieldValues` returns the values on one file or folder:

```
curl -s -i -G -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/shares/$SHARE/field-values" --data-urlencode "path=/Reports/Q3.xlsx"
```

```
[
  { "field_id": "5b3f...", "value": "Approved",
    "updated_at": "2026-10-07T09:10:02Z",
    "updated_by": { "id": "8f2c1d4e-...", "name": "Alex Morgan" } }
]
```

A field that holds several values appears once for each value. `value` is a string, a number or a boolean according to the field's type; a `location` is an object. `updated_by` is `null` where it isn't known. Only the fields the person is allowed to see on that share are included. The `ETag` header covers the whole set:

send it as `If-None-Match` to get `304`, and as `If-Match` when you set a value.

5.8 Versions: `listVersions`

```
curl -s -G -H "Authorization: Bearer $TOKEN" \
  "$FOLDR/api/v1/shares/$SHARE/versions" --data-urlencode "path=/Reports/Q3.xlsx"
```

```
{
  "items": [
    { "id": "3HL4kqtJvjVBH_QUKNd.Fb3NUi", "modified": "2026-10-07T09:02:11Z", "size": 48213,
"current": true },
    { "id": "mBLNn8a.dZS0QpWuxkmW3J_FT.", "modified": "2026-10-05T16:40:30Z", "size": 47902,
"current": false }
  ],
  "next_cursor": null,
  "truncated": false
}
```

The versions of a file that its storage keeps, newest first, all on one page. They are earlier saves (Foldr's own hosted storage, S3 and Azure where versioning is on, SharePoint) or an SMB server's shadow copies ("Previous Versions"). Storage that keeps none answers an empty list, and so does a person who lacks the share's **List versions** permission, so an empty list doesn't prove there are no versions. The `id` is what `restoreVersion` takes; it's opaque. Someone who may only read the approved version of a published file is shown that version alone. `listVersions` costs 2.

Listing versions is a read (`foldr.read`); putting one back is a change that needs `files.overwrite`.

6.0 Writing

Making changes is where the API asks the most of a client. Read this page once before you write anything.

6.1 Before you write

What has to allow a change. Every change needs its scope, a person who may make it, an app whose administrator has allowed it, and a share where it's allowed. See what decides whether a change is allowed. An app that hasn't been given the right gets `403 write-not-allowed`; an app asking about a share outside its policy gets the same `404 not-found` as a share that doesn't exist.

Nothing is ever replaced silently. A create, a copy or a move that finds the name taken doesn't overwrite: the default is to refuse (`on_conflict: "fail"`, answering `409 conflict`, or `412` for an upload), and `on_conflict: "rename"` picks the first free name ("Q3 copy.xlsx", "2027 copy"), so **always read the `path` in the answer** to see what name the item got.

Read, then change. Changes to something that exists need the ETag you last read (`If-Match`). Read the item first (`getItem`, or the answer to your last change), and send its ETag with the change. If it's stale you get `412`: read again and decide again, rather than forcing it. See ETags.

Retry safely. The calls that create or remove things take an `Idempotency-Key`, so a retry after a timeout can't do it twice. See Idempotency keys.

Things Foldr refuses whatever the app is allowed. An item someone else has **checked out** is `423 checked-out`. An item under a **legal hold**, or a folder holding one, is `403 legal-hold` for rename, overwrite, move, restore and delete. A **published** file (one published as an approved version) is `409 published` and is never moved or deleted by an app. The document of a **signing still out** is `409 envelope-source`. Show these to your user as "not now" or "not here" answers, not as faults.

The parent folder must exist. A create or a move into a folder that isn't there is `404 not-found`. Create the folders first.

What you want	Operation	Scope	Needs	Answers
Make a folder	`createFolder`	`files.write`	`Idempotency-Key`	`201`
Upload a new file	`uploadFile`	`files.write`	`If-None-Match: *`, `Idempotency-Key`, `Content-Length`	`201`
Upload straight to the storage	`createUploadSession`, `completeUploadSession`	`files.write`	none	`201`, `201`
Rename	`renameItem`	`files.write`	`If-Match`	`200`
Copy	`copyItem`	`files.write`	`Idempotency-Key`	`201`, or `202` with an operation
Replace a file's contents	`overwriteFile`	`files.overwrite`	`If-Match`, `Content-Length`	`200`

What you want	Operation	Scope	Needs	Answers
Put an earlier version back	`restoreVersion`	`files.overwrite`	`If-Match`, `Idempotency-Key`	`200` (or `202`, held)
Move (within a share or to another)	`moveItem`	`files.move`	`If-Match`, `Idempotency-Key`	`200`, or `202` with an operation
Delete	`deleteItem`	`files.delete`	`If-Match`, `Idempotency-Key`	`200` (or `202`, held)
Set a custom field	`setFieldValue`	`fields.write`	`If-Match`	`200`
Ask to share	`createShareRequest`	`sharing`	`Idempotency-Key`	`202`

Every one of these can also answer the errors common to all calls (`401`, `403 insufficient-scope`, `403 write-not-allowed`, `429`, `500`, `503`) and, when your app has a daily limit, `429 daily-write-limit` (or `destructive-budget`). The sections below list the ones particular to each.

6.2 Create a folder: `createFolder`

```
curl -s -X POST "$FOLDR/api/v1/shares/$SHARE/folders" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d '{"path": "/Reports/2027"}'
```

`201 Created`, with the folder as an item (its canonical `path`), its `ETag`, and a `Location` header that reads it back.

- `path` is the new folder's full path, its name last. The folder it goes in must exist (`404` if not). `on_conflict` is `fail` (the default, `409 conflict`) or `rename`.
- Also: `409 not-a-folder` if the parent is a file; `403 write-not-allowed`; `413 too-large`.

6.3 Upload a file: `uploadFile`

An upload is one `PUT` whose body is the file, at any size. The file streams to the share as it arrives.

```
curl -s -X PUT "$FOLDR/api/v1/shares/$SHARE/files?path=%2FReports%2F2027%2FQ1.xlsx" \
  -H "Authorization: Bearer $TOKEN" \
  -H "If-None-Match: *" \
  -H "Idempotency-Key: $(uuidgen)" \
  -H "Content-Type: application/octet-stream" \
  --data-binary @Q1.xlsx
```

```
{
  "share_id": "01a11dce-14f0-7f23-b857-c621356097a0",
  "path": "/Reports/2027/Q1.xlsx",
  "name": "Q1.xlsx",
  "kind": "file",
  "size": 48213,
  "modified": "2027-04-02T10:15:09Z",
```

```
  "mime_type": "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
  "export": null,
  "etag": "W/\"48213-1806660909\"",
  "read_only": false,
  "checked_out": null
}
```

- **`If-None-Match: *` is required.** It's HTTP's way of saying "create this, never replace anything". Without it: `428 precondition-required`. If something already has the name: `412 precondition-failed`. With `on_conflict=rename` Foldr takes the next free name instead, so read the answer's `path` (and `Location`).
- **`Content-Length` is required**, so that a share without room refuses (`413 too-large`) before the first byte. A chunked request without a length is `400 invalid-request`. Most HTTP libraries set it when you give them a file or a byte string; if yours streams from a generator, give it the length.
- **`If-Match` is refused** (`400`): this operation only creates. To replace a file use `overwriteFile`.
- A person can have only a few uploads running at once through your app (4). More than that: `429 rate-limited` with `Retry-After: 10`. Queue your uploads.
- The `Idempotency-Key` also matches the *bytes*: the same key with other bytes is `422 idempotency-key-reused`.
- Foldr doesn't limit the size of an upload. Quotas and the storage's own limits do.
- **If an upload is cut off** (your connection dropped, or the server restarted), check with `getItem` whether anything is at the name before you retry. On some storage a partial file can be left behind, and a retry with the default `on_conflict` then answers `412`. Where the same `Idempotency-Key` is still held, the retry answers `409 conflict` until it frees (about two minutes after the last progress); then it runs.

Errors particular to it: `404` (the folder isn't there), `409 not-a-folder`, `409 conflict` (a request with this key, or the same name, is being handled right now), `412`, `413`, `422`, `428`.

6.4 Upload straight to the storage: upload sessions

On a share whose storage supports it (Amazon S3 and compatible storage, Microsoft 365, and Dropbox for files up to 150 MB), the bytes can go from your app straight to the storage, which spares the double trip through Foldr. Use it for large files and for slow or flaky connections where the storage can resume. Where the share doesn't offer it, `createUploadSession` answers `409 no-direct-upload`: use `uploadFile` instead.

1. Open a session.

```
curl -s -X POST "$FOLDR/api/v1/uploads" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"share_id": "01a11dce-...", "path": "/Reports/2027/Big.mov", "size": 734003200,
       "content_type": "video/quicktime"}'
```

`size` is the file's exact size in bytes; completing the session checks it. As with every create, nothing is replaced: `on_conflict` is `fail` or `rename`.

```
{
  "id": "u_8f2c...",
  "mode": "direct",
```

```
  "share_id": "01a11dce-...",
  "path": "/Reports/2027/Big.mov",
  "size": 734003200,
  "url": "https://storage.example.net/...",
  "method": "PUT",
  "headers": { "Content-Type": "video/quicktime" },
  "chunk_size": null,
  "expires_at": "2027-04-02T10:30:09Z"
}
```

2. Send the file to the storage, to `url`, with the `method` and exactly the `headers` the session gave you. The `url` is a credential for this one upload: **send it no Foldr token, and don't log or share it.**

- `chunk_size` `null`: send the whole file in one request.
- `chunk_size` set: send the file in pieces of exactly that many bytes (the last may be shorter), each with a `Content-Range` header (`Content-Range: bytes 0-3276799/734003200`). Microsoft 365 uses multiples of 320 KiB. Every piece but the last is answered `202`; the last is answered `200` or `201`.
- Do it before `expires_at`; after that the URL stops working.

3. Complete the session, so Foldr checks the file is there and announces it:

```
curl -s -X POST "$FOLDR/api/v1/uploads/$SESSION_ID/complete" -H "Authorization: Bearer $TOKEN"
```

`201`, with the new item. The storage must hold exactly `size` bytes at the session's path, or you get `409 upload-incomplete`; the session stays open, so finish sending and complete again. Completing an already completed session answers the same file again, so it's safe to retry.

Or cancel it: `POST /uploads/{id}/cancel` answers `204`. It removes the empty placeholder some storage keeps for the upload, never a file with bytes in it. A completed session can't be cancelled (`409`).

What happens to a session you forget. It's kept until about an hour after its `expires_at`; one neither completed nor cancelled by then is gone (`404`), and a file sent to the storage for it is removed and never announced. The removal is a scheduled clean-up of files Foldr can show were written for that session; it leaves a file alone if it cannot, so a leftover is possible.

Sessions aren't limited in number (no server process is held), but opening one costs 2 on your rate budget, as does completing it.

6.5 Rename: `renameItem`

```
curl -s -X POST "$FOLDR/api/v1/shares/$SHARE/rename" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -H 'If-Match: W/"48213-1806660909"' \
  -d '{"path": "/Reports/2027/Q1.xlsx", "new_name": "Q1 final.xlsx"}'
```

`200`, with the item **at its new canonical path** (the old path names nothing afterwards, except as another spelling of a case-only rename on storage that ignores case). `new_name` is one name: no `/`, no control characters, no space at either end (`400 invalid-path` if not). It stays in the same folder; to put something in another folder use `moveItem`.

Errors: `428`, `412`, `409 conflict` (the name is taken; nothing is replaced), `423 checked-out`, `403 legal-hold`. A rename needs only `files.write`.

6.6 Copy: `copyItem`

```
curl -s -X POST "$FOLDR/api/v1/shares/$SHARE/copy" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d '{"path": "/Reports/2027/Q1.xlsx",
       "to_share": "01a11dce-...", "to_path": "/Archive/Q1.xlsx", "on_conflict": "rename"}'
```

Copies a file **or a folder** with everything in it, within its share or into another of the person's shares (`to_share` is this share's id for a copy within it). The original isn't touched. `to_path` is the copy's full path, its name last, and the folder it goes in must exist. A folder copy is a new folder, never merged into one that's already there.

- **A small copy happens in the request**: `201` with the copy, its `ETag` and a `Location`.
- **A big or slow one happens afterwards**: `202` with an operation to poll. That's a folder, anything with Microsoft 365 at either end, a file over 64 MiB going between shares (or over 1 GiB within one share, deployment defaults an administrator can change), and any copy you send with `Prefer: respond-async`. If the `Prefer` header is why you got an operation, the answer carries `Preference-Applied: respond-async`.
- Limits: a folder copy takes at most 10,000 files and folders, nested at most 64 deep (`413 copy-too-large`); a person may have at most 20 copies waiting through one app (`429 rate-limited`), and two run at a time (two at a time across the whole site as well).
- A copy into **another** share is refused (`403 download-blocked`) when a label on the file stops it being downloaded.
- Both shares have to be ones your app may change.
- Errors: `409 conflict`, `409 not-a-file` / `not-a-folder`, `400 copy-into-itself`, `409 copy-incomplete` and `503 copy-interrupted` (from an operation), `422`, `413`.

6.7 Replace a file's contents: `overwriteFile`

Needs the separate scope `files.overwrite`. The body is the new contents, streamed to the share, at any size:

```
curl -s -X PUT "$FOLDR/api/v1/shares/$SHARE/content?path=%2FReports%2F2027%2FQ1.xlsx" \
  -H "Authorization: Bearer $TOKEN" \
  -H 'If-Match: W/"48213-1806660909"' \
  -H "Content-Type: application/octet-stream" \
  --data-binary @Q1-revised.xlsx
```

`200`, with the file and its **new** ETag.

- `If-Match` is required and must name the version you mean: `428` without it, `412` if it's stale, `400` for `*`.
- `Content-Length` is required. A body that ends before its `Content-Length` leaves the file exactly as it was.
- Refused for a file someone else has checked out (`423`), one under a legal hold (`403 legal-hold`), the document of a signing still out (`409 envelope-source`), and, for a person who may only read the approved version of a published file, that file (`403 write-not-allowed`).

- Where the share's storage keeps no earlier versions, the old contents can't be got back, and an administrator may have refused apps this change on that share: `409 not-recoverable-here`. See Recoverability.
- If the organisation requires approval for the app's changes, overwrite isn't offered at all: `403 needs-approval-not-available`. New contents can't be held for approval.
- Counts against the app's daily allowance for removing and replacing.

6.8 Put an earlier version back: `restoreVersion`

Also `files.overwrite`. Take a version id from `listVersions`:

```
curl -s -X POST "$FOLDR/api/v1/shares/$SHARE/versions/restore" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -H 'If-Match: W/"48213-1806660909"' -H "Idempotency-Key: $(uuidgen)" \
  -d '{"path": "/Reports/2027/Q1.xlsx", "version_id": "mBLNn8a.dZS0QpWuxkmW3J_FT."}'
```

`200`, with the file and its new ETag. On Foldr's own hosted storage the version is copied over the file inside the storage; on the other storage that keeps versions (S3, Azure, SharePoint, an SMB server's shadow copies) Foldr is given that version's bytes and proves them whole before they replace the file, so a restore that fails leaves the file as it was. Refused as replacing a file is. In approval mode it's held for an approver: `202` with an operation.

6.9 Move: `moveItem`

Needs `files.move`. Moves a file or a folder (with everything in it) to another folder, keeping its name or giving it a new one.

```
curl -s -X POST "$FOLDR/api/v1/shares/$SHARE/move" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -H 'If-Match: W/"48213-1806660909"' -H "Idempotency-Key: $(uuidgen)" \
  -d '{"path": "/Reports/2027/Q1.xlsx", "to_path": "/Archive/2027/Q1.xlsx"}'
```

`200`, with the item at its new canonical path. `to_path` is the full new path, its name last (the same name or a new one); the folder it goes in must exist. Nothing already there is replaced (`on_conflict` `fail` or `rename`).

- **Within a share**, the storage moves it in one step, always in the request.
- **To another share** (add `"to_share": "<other share id>"`; both shares must be ones the app may remove from and change), the item is **copied, the copy proven whole, and only then is the original removed**, so nothing can be lost. A small file is moved in the request (`200`). A folder, anything with Microsoft 365 at either end, a file over 64 MiB, or any such move sent with `Prefer: respond-async` is moved afterwards: `202` with an operation of kind `move`. A file whose label keeps it in its share can't be moved out: `403 download-blocked`.
- **`409 source-remains`**: the copy is whole but an original couldn't be removed. The item is in both places, nothing is lost, and the answer says where. Check the copy, then delete what's left; don't move it again. See Operations.
- A folder of more than 1,000 items is `409 folder-too-large`; a folder moved into itself is `400 move-into-itself`; a published file is `409 published`.

- Not asked of a move: whether the storage could undo it. Moves are allowed even where an administrator has refused unrecoverable changes, because moving something back undoes a move.
- A folder counts every item in it against the daily allowance.

6.10 Delete: `deleteItem`

Needs `files.delete`.

```
curl -s -X DELETE "$FOLDR/api/v1/shares/$SHARE/items?path=%2FReports%2F2027%2FQ1.xlsx" \
  -H "Authorization: Bearer $TOKEN" \
  -H 'If-Match: W/"48213-1806660909"' -H "Idempotency-Key: $(uuidgen)"
```

```
{
  "deleted": { "share_id": "01a11dce-...", "path": "/Reports/2027/Q1.xlsx", "name": "Q1.xlsx",
"kind": "file",
               "size": 48213, "modified": "2027-04-02T10:15:09Z", "mime_type": "...", "export":
null,
               "etag": "W/\"48213-1806660909\"", "read_only": false, "checked_out": null },
  "items": 1,
  "recoverable": "snapshot",
  "destination": "shadow-copies-if-enabled",
  "where": "the server's shadow copies, if it keeps them"
}
```

`200`, saying what was deleted **and where its storage keeps it now**, so you can tell your user whether it can be brought back (see Recoverability): `recoverable` is `bin`, `snapshot` or `none`; `destination` is a stable name; `where` is the same in words for a person.

- A folder goes with everything in it, up to 1,000 items (`409 folder-too-large` beyond that). `items` says how many files and folders went. The top of a share can't be deleted.
- Refused for anything someone else has checked out (`423`), under a legal hold (`403`), published (`409 published`), or the document of a signing still out (`409 envelope-source`).
- Where the storage has no recycle bin, an administrator may have refused apps deletions there: `409 not-recoverable-here`.
- Counts items against the daily allowance (`429 destructive-budget`).
- In approval mode, a delete isn't made at once: `202` with an operation held for an approver (see Operations and approval).
- Repeating a delete with the same `Idempotency-Key` returns the first answer. Repeating it with a new key after the item has gone is `404`.

6.11 Setting a custom field value: `setFieldValue`

Needs `fields.write`. The person must be allowed to set that field, a field with allowed values takes one of them, and a value must suit the field's type.

```
curl -s -X PUT "$FOLDR/api/v1/shares/$SHARE/field-values/$FIELD_ID" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -H 'If-Match: "fv-4d1c..."' \
  -d '{"path": "/Reports/2027/Q1.xlsx", "value": "Approved"}'
```

`200`, with every value on the file that the person can see (as `getFieldValues` does) and the new ETag. `If-Match` is the ETag of the file's *field values* as you last read them from `getFieldValues`: `428` without one, `412` if anything changed since. Send `*` for "whatever they are now".

`value` is a string, a number, or `true` or `false`, as the field's type takes it (a date as `2027-01-31`). `422 validation-failed` if it doesn't suit. Also `423 checked-out`, `409 not-a-file`. A field whose change the organisation holds for an approver is held as it is in Foldr; read the values again to see where it stands.

6.12 Asking to share: `createShareRequest`

Needs `sharing`. An app can't share anything by itself. It can **ask**, and every request waits for an approver in Foldr. The answer never carries a link.

```
curl -s -X POST "$FOLDR/api/v1/shares/$SHARE/share-requests" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d '{"path": "/Reports/2027/Q1.xlsx", "kind": "secure_link",
       "recipient_emails": ["buyer@customer.example"], "expiry_days": 14,
       "message": "The Q1 figures"}'
```

```
{
  "id": "6f1a...",
  "share_id": "01a11dce-...",
  "path": "/Reports/2027/Q1.xlsx",
  "kind": "secure_link",
  "status": "pending_approval",
  "created_at": "2027-04-02T10:20:00Z",
  "decided_at": null
}
```

`202`. The `kind` says what is asked for:

`kind`	Asks for	Also send
`people`	Access for people in the organisation	`people` (email addresses, usernames or exact group names, at most 20), `permission` (`read`, the default, or `write`)
`public_link`	A link anyone who has it can open	
`secure_link`	A link for outside email addresses	`recipient_emails` (at most 20)
`inbox`	An email address files can be sent to (a folder)	

`expiry_days` is the days until access ends once approved (`0` or absent: never; for a link, `-1` ends it after its first download); `message` is a note shown with the link, up to 1,000 characters.

- Poll the outcome with `getShareRequest` (`GET /share-requests/{id}`) or list your own with `listShareRequests` (`GET /share-requests`, paged as in Conventions). `status` is `pending_approval`, then `approved`, `declined` (by an approver, or left undecided until it lapsed) or `withdrawn`. Once approved, the sharing exists in Foldr, where the person finds the link; the API never gives it to the app.

- `409 no-approver` if nobody other than the person could approve it. `409 already-shared` if the person already shares the item or has a request for it waiting. `422 validation-failed` for a recipient who can't be found or a term that isn't allowed.
- You can read only the person's own requests, made through any app; someone else's is the same `404` as one that doesn't exist.

7.0 Operations and approval

Some work doesn't fit inside one request: copying a folder, moving a big file between shares, or a change that has to wait for a person to approve it. For these Foldr answers `202 Accepted` with an **operation**, and does the work afterwards. Your app polls the operation until it's finished.

7.1 When you get an operation

You called	You get an operation when
`copyItem`	The item is a folder, or anything with Microsoft 365 at either end, or a file over 64 MiB going between shares (1 GiB within one share; deployment defaults), or you sent `Prefer: respond-async`.
`moveItem`	The move is to another share and the item is a folder, or has Microsoft 365 at either end, or is over 64 MiB, or you sent `Prefer: respond-async`. Or the app is in approval mode (every move).
`deleteItem`, `restoreVersion`	Only in approval mode: the change waits for an approver.

A move within one share, and every change in the other tables, never produce an operation (except in approval mode for moves, deletes and restores).

`Prefer: respond-async` (RFC 7240) asks for an operation whatever the size, for a copy, or for a move to another share. When the header is why you got one, the `202` carries `Preference-Applied: respond-async`. Without the header a small file is done in the request, so a client that always wants one code path can always send it.

7.2 The `202`

```
HTTP/2 202
Location: https://files.example.com/api/v1/operations/2f7776a7-2d0a-4b90-b939-c111fadc9491
Retry-After: 5
Preference-Applied: respond-async
Content-Type: application/json
```

```
{
  "id": "2f7776a7-2d0a-4b90-b939-c111fadc9491",
  "kind": "copy",
  "state": "queued",
  "phase": null,
  "created_at": "2027-04-02T10:17:29Z",
  "started_at": null,
  "finished_at": null,
  "source": { "share_id": "01a11dce-...", "path": "/Invoices", "kind": "folder" },
  "target": { "share_id": "01a11dce-152f-...", "path": "/Invoices" },
  "version_id": null,
  "progress": { "items_done": 0, "items_total": 4, "bytes_done": 0, "bytes_total": 56 },
  "result": null,
  "deletion": null,
  "approval": null,
  "failures": [],
```

```
  "failures_truncated": false
}
```

`Location` is the address to poll, and `Retry-After` is how long to wait before the first poll (5 seconds; 60 for an operation waiting for approval, which takes hours).

7.3 Polling: `getOperation`

`GET` the `Location` until it answers something other than `202`. That is the whole protocol, and the one Power Automate and Logic Apps follow on their own.

```
curl -s -i -H "Authorization: Bearer $TOKEN" "$LOCATION"
```

The operation is	`getOperation` answers
queued, running or cancelling	`202` with the operation, `Location` and `Retry-After`. Ask again after that many seconds.
held for an approver (`pending_approval`)	`202`, with a longer `Retry-After`.
succeeded	`200` with the operation, whose `result` is what it made. `Content-Location` names the item (absent for a delete).
failed	**The problem it ended with, at that problem's own status**, carrying the operation in the problem's `operation` member. For example `409 copy-incomplete`, `409 source-remains`, `403 download-blocked`, `503 copy-interrupted`.
cancelled	`409 operation-cancelled`, carrying the operation.
declined, expired	`403 change-declined`, `409 change-expired` (see below).

So: a `2xx` is a state of play, and anything else is the end, with the reason in the problem. A well-behaved loop:

```
loop:
    r = GET location
    if r.status == 202: sleep(Retry-After); continue
    if r.status == 200: done, r.json.result is the item
    else:               it ended as r.json.type says
```

Poll no faster than `Retry-After`. Each poll costs 1.

Fields of an operation

Field	
`id`	UUID.
`kind`	`copy`, `move`, `delete` or `restore`. (Open list.) A delete or restore is only ever an operation held for approval.
`state`	`pending_approval`, `queued`, `running`, `cancelling`, `succeeded`, `failed`, `cancelled`, `declined` or `expired`.
`phase`	For a move: `copying` (everything is copied first; nothing has been removed, and a cancel removes the partial copy) then `deleting` (the copy is whole and proven, and the originals are being removed: too late to cancel). `null` for anything else.
`created_at`, `started_at`, `finished_at`	RFC 3339 UTC; `started_at` is `null` while queued, `finished_at` until it ends.
`source`	What's being copied, moved, deleted or restored: share id, path, `kind`.
`target`	Where it was asked to go: the path as you sent it. `result.path` is where it landed (`on_conflict: "rename"` may have given it another name). `null` for a delete or a restore.
`version_id`	For a restore, the version asked for; otherwise `null`.
`progress`	`items_done` and `bytes_done`, and `items_total` and `bytes_total`, which are `null` until Foldr has counted what there is to do. Nothing is guessed.
`result`	Once it has succeeded, what it made (the copy, the moved item, the restored file), as an item. `null` until then and for a delete. A move that ended `source-remains` also has it: the whole copy it made.
`deletion`	For a delete held for approval, once it has been made: the same answer `deleteItem` gives.
`approval`	For an operation held for an approver: how its approval stands (below).
`failures`	At most 100 items (see `failures_truncated`): for a copy that ended `copy-incomplete`, the items it couldn't copy; for a move that ended `source-remains`, the items still at the source. Each is `{ "path", "code" }` where `code` is a problem slug (`download-blocked`, `invalid-path`, `not-found`, …).
`failures_truncated`	`true` when more items failed than `failures` lists.

7.4 Finding an operation again: `listOperations`

If your app loses an operation's id (a restart, a crash), list them:

```
curl -s -H "Authorization: Bearer $TOKEN" "$FOLDR/api/v1/operations?limit=20"
```

The person's operations through your app, newest first, as a page, including finished ones for **7 days after they finish**. After that an operation is gone (`404`).

An operation is **yours alone**: the person, through your app. Another person's or another app's operation is the same `404` as one that doesn't exist. And it's read under the scope of the change that started it: a copy with `files.write`, a move with `files.move`, a held delete with `files.delete`, a held restore with `files.overwrite`. A token that has one of the four scopes but not that one sees `404`, not `403`. In practice an app that makes the change has the scope to watch it.

7.5 Cancelling: `cancelOperation`

```
curl -s -X POST -H "Authorization: Bearer $TOKEN" "$FOLDR/api/v1/operations/$ID/cancel"
```

`204` either way. What happens next depends on where it was:

- **Queued**: cancelled at once.
- **Running** (or a move still `copying`): it stops at the next item, and the copy it had made so far is **removed**. A move's original is untouched.
- **A move already `deleting`** its originals: stopping would leave the item split between two places, so it finishes. A copy whose last item is done as the cancel arrives also succeeds.
- **Held for an approver**: withdrawn at once, nothing changed. Its `approval.status` then says `withdrawn`.

Poll afterwards to see how it ended (`409 operation-cancelled`). Cancelling something that has already finished is `409 operation-finished`; read it for how it ended.

7.6 Moves and `source-remains`

A move to another share is made safe by order: every item is copied first, the copy is checked whole, and only then are the originals removed. So the worst case is that an item is in both places, never in neither.

If the copy is whole but an original couldn't be removed afterwards, the move ends `409 source-remains`:

- the item is in both places, and nothing is lost;
- the operation's `result` is the whole copy, and its `failures` list what is still at the original place;
- what to do: check the copy, then delete what's left at the original place yourself (with `deleteItem`). Moving it again would make a second copy.

A synchronous move (a small file in the request) can end the same way: the `409` answer says where the item is, and is kept for its `Idempotency-Key`, so a retry is told the same and doesn't copy again.

7.7 Idempotent retries

Send an `Idempotency-Key` with `copyItem` and `moveItem` and a retry after a lost `202` gives you **the same operation** back (with `Idempotent-Replayed: true`) and starts nothing new. After the 24 hours the key is kept, a retry still finds the operation: one still under way answers `202` again, and one that has finished answers as the original request would have if it had made the change at once, or with the problem it ended with, carrying the operation. If you lose the id altogether, `listOperations` finds it.

7.8 What can stop an operation

Everything that can refuse a request can also end an operation, because Foldr checks again when it does the work. The common ones: `409 conflict` (the name was taken in the meantime), `403 download-blocked` (a label keeps the file in its share), `409 copy-incomplete`, `503 copy-interrupted` (the server stopped the work too often, or it waited too long to start, a day), `503 write-failed`, `503 share-unreachable`. Operations that were never started, or whose worker was lost, are restarted or failed by Foldr; they're never left waiting for ever.

7.9 Changes held for approval

An organisation can require **approval** for an app's changes that remove or replace. It is set per app, in **Settings > Connected apps > Policy**, as **Approve each change first**, and needs Reviews & approvals on the site. An app in approval mode works like this.

What a client sees

A **delete**, a **move** or a **version restore** passes every check it would pass to be made at once (scope, permissions, policy, `If-Match`, holds, the daily allowance) and is *not made*. Instead it answers:

```
HTTP/2 202
Location: https://files.example.com/api/v1/operations/9d3c...
Retry-After: 60
```

with an operation in state `pending_approval`:

```
{
  "id": "9d3c...",
  "kind": "delete",
  "state": "pending_approval",
  "phase": null,
  "created_at": "2027-04-02T10:20:00Z",
  "started_at": null,
  "finished_at": null,
  "source": { "share_id": "01a11dce-...", "path": "/Reports/2027/Q1.xlsx", "kind": "file" },
  "target": null,
  "version_id": null,
  "progress": { "items_done": 0, "items_total": null, "bytes_done": 0, "bytes_total": null },
  "result": null,
  "deletion": null,
  "approval": { "status": "pending", "expires_at": "2027-04-16T10:20:00Z", "decided_at": null },
  "failures": [],
  "failures_truncated": false
}
```

(An illustration of the shape: your values will differ.) Someone **other than the person who asked** is asked in Foldr to approve it. They are shown the app, the person, what will happen and where the item is going. Who approves, and where, is the organisation's business and isn't visible to the app.

How the app learns the outcome

Poll the operation (`getOperation`), at the slow pace its `Retry-After` gives, a minute at a time, because approvals take hours or days. It ends in one of these ways:

Outcome	`getOperation` answers	`approval.status`	State
Approved, and made	`200`; for a delete the `deletion` says what was deleted and where it is now; for a move or restore `result` is the item	`approved`	`succeeded`
Declined by an approver	`403 change-declined`	`declined`	`declined`
Nobody decided in time (14 days, unless the organisation says otherwise)	`409 change-expired`	`expired`	`expired`
The app, or the person, withdrew it	`409 operation-cancelled`	`withdrawn`	`cancelled`
Approved, but the item changed after you asked	`412 changed-since-requested`	`approved`	`failed`
Approved, but the share can't be opened as the person now	`409 requester-not-connected`	`approved`	`failed`
Closed before anyone decided (for instance, the app was disconnected)	`409 operation-cancelled`	`closed` or `withdrawn`	`cancelled`

In every failing case in this table, nothing was changed. An approved change that then fails while it is being made is different: a move to another share can copy the item and then fail to remove the original, which ends as `409 source-remains` with the copy as the `result` (see Moves and source-remains).

- **Declined or expired**: show it to your user as such. Ask again only if they want to.
- **`changed-since-requested`**: an approved change is made *as things stand then*. Your `If-Match` is checked again at that moment; if the item has changed since, nothing is touched. Read the item again, and ask again if the change is still wanted.
- **`requester-not-connected`**: an approved change is only ever made as the person who asked, never with anyone else's access. If their own sign-in to the storage isn't available any more, or they can no longer reach the share, it can't be made. Ask again.
- Approval statuses are an open list: `pending`, `approved`, `declined`, `expired`, `withdrawn`, `closed`.

Other things to know

- **The daily allowance is charged when you ask**, not when it's made, and isn't given back if the change is declined, expires or is withdrawn. It bounds how much an app can ask approvers for. A request refused for another reason costs nothing.
- **No approver, no request.** If nobody other than the person could approve it (or the organisation has no approvals), the request is refused up front: `409 no-approver`. Nothing is charged.
- **Replacing a file's contents is not offered** to such an app: `403 needs-approval-not-available`. New contents can't be held for approval. The person can replace the file in Foldr.
- **Rename, copy, upload and the rest are not held.** Approval mode covers deletes, moves and version restores only, because those are the ones that remove or replace.
- **Cancelling** a held operation withdraws it at once (`cancelOperation`, `204`).
- **If the person disconnects the app**, or an administrator revokes it, everything of its still waiting is withdrawn and never made. The app's tokens stop working at the same moment, so it can't poll; if the person connects it again, `listOperations` shows those operations ended.
- Operations are kept 7 days after they end, so a late poll still finds the answer.

8.0 Recoverability and audit

8.1 Why recoverability matters to a client

Deleting and replacing are the changes people regret. Whether one can be undone depends on the storage behind the share, and Foldr can't make a file server keep what it doesn't keep. So the API does two things: it tells your app, after the fact, **where a deleted item went**, and it lets an administrator **refuse apps changes that can't be undone** on a share.

You can use both. Tell your user what happened in words they can act on, and don't promise "you can get it back" where the answer says otherwise.

8.2 What a delete tells you

`deleteItem` answers with, besides the item that was deleted:

Field	Values
`recoverable`	`bin`: the storage keeps it in a recycle bin, trash or earlier versions, so it can be brought back there. `snapshot`: only the server's shadow copies hold it, as of the last one, so anything changed since is lost. `none`: as far as Foldr can tell it's gone for good.
`destination`	A stable name for where it is: `microsoft-365-recycle-bin`, `google-drive-trash`, `box-trash`, `dropbox-deleted-files`, `foldr-deleted-files`, `bucket-versions`, `azure-soft-delete`, `shadow-copies`, `shadow-copies-if-enabled`, or `unknown` (Foldr can't tell whether this storage keeps deleted items). `null` where it's certainly gone. The list can grow.
`where`	The same in words for a person: "the Microsoft 365 recycle bin". Use `destination` in code and `where` in a message.
`items`	How many files and folders went.

Bins belong to the share's storage account, not to the person. Getting something back is usually a job for whoever manages that storage, and Foldr's API has no "undelete". Say so to your user.

8.3 By storage

Storage	After a delete	After an overwrite or restore	Notes
Microsoft 365 (OneDrive, SharePoint, Teams)	`bin`, `microsoft-365-recycle-bin`	Earlier versions where the storage keeps them (SharePoint does)	
Google Drive	`bin`, `google-drive-trash`		
Box	`bin`, `box-trash`		
Dropbox	`bin`, `dropbox-deleted-files`		
Foldr hosted storage	`bin`, `foldr-deleted-files`: Foldr keeps deleted items for 30 days	Earlier versions are kept and listed	

Storage	After a delete	After an overwrite or restore	Notes
Amazon S3 and compatible (your own bucket)	`bin`, `bucket-versions`, **only while the bucket's versioning is on**; otherwise `none`; `unknown` until Foldr has been able to ask	Earlier versions only while versioning is on	Foldr asks the bucket whether versioning is enabled
Azure Blob storage	`bin`, `azure-soft-delete`, **only while the account's soft delete is on**; otherwise `none`; `unknown` until asked	Earlier versions where the account keeps them	
SMB file servers	`snapshot`, `shadow-copies` where the server keeps shadow copies ("Previous Versions"); `snapshot`, `shadow-copies-if-enabled` where Foldr hasn't asked yet; `none` where the server has none	Shadow copies, as of the last one	Changes since the last shadow copy are lost
Edge agents	`snapshot`, `shadow-copies-if-enabled`		Foldr can't ask the customer's own server
Local disk, FTP, SFTP	`none`	`none`	
WebDAV, other	`none` with `unknown`		Some WebDAV servers have a bin, some don't; Foldr can't tell

Two things follow:

- **`recoverable: "bin"` is a promise about the storage, not about getting the bytes back by API.** Recovery happens in the storage's own tools.
- **`snapshot` counts as "not safely undoable".** A change made since the last shadow copy is gone for good, so Foldr treats `snapshot` like `none` when it decides whether an app may make a change.

8.4 What an administrator can refuse

For each share, an administrator can switch off **Allow unrecoverable changes from connected apps** (in the share's settings). It is **on by default**.

- **On**: apps with the right permission may replace, restore over and delete in the share, whatever the storage keeps. This is how people already work in Foldr, and `deleteItem` still tells you where the item went.
- **Off**: where the storage could not bring the change back (anything but `bin`), an app's overwrite, version restore or delete is refused with `409 not-recoverable-here` and nothing is changed. Where the storage does keep a bin or versions, the switch makes no difference.
- **Moves are never refused for this**, because a move loses nothing and moving the item back undoes it.
- It affects apps only. People using Foldr itself aren't stopped.

Handle `409 not-recoverable-here` by telling your user the change isn't available to apps on this share, and that they can make it in Foldr themselves. Retrying won't help.

An app also needs its own permission for each of these changes, in the shares an administrator names (Getting access), and the consent page tells the person in plain words when a scope can't be undone: "Replace the contents of files, and put back earlier versions, in the shares an administrator allows. Where the storage keeps no earlier versions, the old contents cannot be got back."

8.5 Telling the person afterwards

Foldr emails the person a daily summary of what an app removed or replaced for them ("Acme Reports deleted 3 items and replaced 2 in Finance"), without a link, so an app can't be impersonated by a mail. Your app doesn't need to send its own confirmation for that, but it should show the user what it did when it did it.

8.6 What administrators can see (audit)

Everything an app does is recorded, as the person and naming the app:

- **Every call that reaches the API is an activity entry** for the person (unless the site has switched that activity type off; a call refused for a bad or missing token never gets that far), in Foldr's activity log (**Settings > Activity**), reads included. It records *where* (the share and the path as your app gave it) and *what* (the operation, for example `getContent` or `deleteItem`). Downloads and changes are always recorded.
- **It never records what a call carried or returned**: no file contents, field values, recipients, messages or passwords.
- **Connections are listed** in **Settings > Connected apps** (the **Connections** tab): who connected which app, with what access, when, and when it was last used. An administrator can disconnect any connection or revoke an app for everyone.
- **A person sees their own** under **Connected apps** on the Security tab of their profile, and gets an email when an app is connected, or when an app they'd already connected is given a new right to make changes.
- **Administrators are alerted** (by default) when an app reaches a person's daily allowance for removing or replacing.
- Held changes are approved or declined in **Reviews & approvals**.

The practical consequence for your app: act as the person and nothing else, and keep your own log of what you did on whose behalf. An administrator who asks "what did this app do on Tuesday?" will get the answer from Foldr, and your own records should agree with it.

9.0 A worked example

A short client in Python that does the whole round trip: sign in with OAuth and PKCE, list the shares the person can write to, upload a file, copy it (asking for an operation, and waiting for it), and handle the answers a client has to expect. It uses only the `requests` library.

Before you run it, register an app as described in Getting access (an administrator adds it under **Settings > Connected apps**, with a redirect URL, and allows `files.write` in its policy) and set:

```
export FOLDR_SITE=https://files.example.com
export FOLDR_CLIENT_ID=...
export FOLDR_CLIENT_SECRET=...        # leave unset for a public app
export FOLDR_REDIRECT_URI=https://reports.acme.example/foldr/callback
export FOLDR_SHARE=...                # optional: the id of the share the app may write to
python foldr_example.py
```

The script prints a sign-in address, you open it in a browser and agree, and you paste back the address Foldr sent you to (the one that begins with your redirect URL and carries `?code=...`). A real app would run a small web server on the redirect URL instead and read the code from the request.

```
"""A small Foldr API v1 client: sign in with OAuth (PKCE), list shares,
upload a file, copy a folder and wait for the copy to finish."""

import base64
import hashlib
import os
import secrets
import time
import uuid
from urllib.parse import parse_qs, urlencode, urlparse

import requests

SITE = os.environ["FOLDR_SITE"]                          # https://files.example.com
CLIENT_ID = os.environ["FOLDR_CLIENT_ID"]
CLIENT_SECRET = os.environ.get("FOLDR_CLIENT_SECRET")    # a confidential app only
REDIRECT_URI = os.environ["FOLDR_REDIRECT_URI"]
API = SITE + "/api/v1"


# --- OAuth: authorisation code with PKCE --------------------------------

def make_pkce():
    verifier = secrets.token_urlsafe(48)                 # 64 characters of A-Za-z0-9_-
    digest = hashlib.sha256(verifier.encode("ascii")).digest()
    challenge = base64.urlsafe_b64encode(digest).rstrip(b"=").decode("ascii")
    return verifier, challenge


def authorize_url(challenge, state, scope):
    return SITE + "/oauth/authorize?" + urlencode({
        "response_type": "code",
        "client_id": CLIENT_ID,
        "redirect_uri": REDIRECT_URI,
        "scope": scope,
        "state": state,
        "code_challenge": challenge,
        "code_challenge_method": "S256",
```

```
        "resource": API,
    })


def token_request(**form):
    auth = (CLIENT_ID, CLIENT_SECRET) if CLIENT_SECRET else None
    if auth is None:
        form["client_id"] = CLIENT_ID
    r = requests.post(SITE + "/oauth/token", data=form, auth=auth, timeout=30)
    if r.status_code != 200:
        raise RuntimeError("token endpoint: " + r.text)  # e.g. {"error":"invalid_grant"}
    return r.json()


def exchange_code(code, verifier):
    return token_request(grant_type="authorization_code", code=code,
                         redirect_uri=REDIRECT_URI, code_verifier=verifier)


# --- The API client ----------------------------------------------------

class FoldrError(Exception):
    """A problem document (RFC 9457). Branch on .type, never on the text."""

    def __init__(self, response):
        try:
            problem = response.json()
        except ValueError:
            problem = {}
        self.status = response.status_code
        self.type = str(problem.get("type", "")).rsplit("/", 1)[-1]
        self.detail = problem.get("detail")
        super().__init__(f"{self.status} {self.type}")


class Foldr:
    def __init__(self, tokens, save_tokens=lambda tokens: None):
        self.tokens = tokens
        self.save_tokens = save_tokens
        self.http = requests.Session()

    def refresh(self):
        # The refresh token is used once: keep the new one before anything else.
        self.tokens = token_request(grant_type="refresh_token",
                                    refresh_token=self.tokens["refresh_token"])
        self.save_tokens(self.tokens)

    def call(self, method, path, ok=(200, 201, 202, 204), **kwargs):
        headers = kwargs.pop("headers", {})
        url = path if path.startswith("http") else API + path   # an operation's Location is a full
URL
        for attempt in range(4):
            headers["Authorization"] = "Bearer " + self.tokens["access_token"]
            r = self.http.request(method, url, headers=headers, timeout=120,
                                  allow_redirects=False, **kwargs)
            if r.status_code == 401 and attempt == 0:
                self.refresh()                          # expired: refresh once, then retry
                continue
            if r.status_code in (429, 503):             # flow control, or a share briefly away
                wait = int(r.headers.get("Retry-After", 5))
                if wait > 120:                          # a daily limit: not for sleeping through
                    break
                time.sleep(wait)
                continue
            break
```

```
        if r.status_code not in ok:
            raise FoldrError(r)
        return r

    # Reading

    def pages(self, path, params=None):
        params = dict(params or {})
        while True:
            page = self.call("GET", path, params=params).json()
            yield from page["items"]
            if not page["next_cursor"]:
                return
            params["cursor"] = page["next_cursor"]       # opaque: send it back as it came

    # Writing

    def upload(self, share_id, path, data: bytes):
        r = self.call("PUT", f"/shares/{share_id}/files", params={"path": path},
                      headers={"If-None-Match": "*", "Idempotency-Key": str(uuid.uuid4()),
                               "Content-Type": "application/octet-stream"},
                      data=data)
        return r.json()                                 # read its "path": it may have been
renamed

    def copy(self, share_id, path, to_share, to_path):
        r = self.call("POST", f"/shares/{share_id}/copy",
                      headers={"Idempotency-Key": str(uuid.uuid4()), "Prefer": "respond-async"},
                      json={"path": path, "to_share": to_share, "to_path": to_path})
        return self.wait(r) if r.status_code == 202 else r.json()

    def wait(self, response):
        """Poll an operation until it is no longer 202; return its result."""
        location = response.headers["Location"]
        delay = int(response.headers.get("Retry-After", 5))
        while True:
            time.sleep(delay)
            # Through call(): a refresh on 401 and a wait on 429/503 apply to polling too.
            r = self.call("GET", location, ok=(200, 202))
            if r.status_code == 202:
                delay = int(r.headers.get("Retry-After", 5))
                continue
            return r.json()["result"]                   # any other status raised: the problem the
operation ended with


def main():
    verifier, challenge = make_pkce()
    state = secrets.token_urlsafe(16)
    print("Open this in a browser, sign in and agree:\n")
    print(authorize_url(challenge, state, "foldr.read offline_access files.write"))
    redirected = input("\nPaste the address you were sent back to: ")
    query = parse_qs(urlparse(redirected).query)
    if query.get("state") != [state]:
        raise SystemExit("state does not match: stop")
    if query.get("iss") != [SITE]:
        raise SystemExit("iss is not the site this app means to talk to: stop")
    foldr = Foldr(exchange_code(query["code"][0], verifier))
    print("granted:", foldr.tokens["scope"])

    # read_only describes the person's own permission, not the app's policy: an
    # administrator may have limited the app to named shares, and a share outside
    # them answers 404. A real app is configured with its destination share.
    shares = [s for s in foldr.pages("/shares") if not s["read_only"] and s["reachable"]]
    wanted = os.environ.get("FOLDR_SHARE")              # the share id the administrator approved
```

```
    if wanted:
        shares = [s for s in shares if s["id"] == wanted]
    if not shares:
        raise SystemExit("no share this person can write to (and the app may use) was found")
    share = shares[0]
    print("using share", share["name"])

    item = foldr.upload(share["id"], "/hello.txt", b"Hello from the API\n")
    print("uploaded", item["path"], item["size"], "bytes")

    try:
        copy = foldr.copy(share["id"], item["path"], share["id"], "/hello copy.txt")
        print("copied to", copy["path"])
    except FoldrError as e:
        if e.type in ("legal-hold", "checked-out"):
            print("not now:", e.type)
        else:
            raise


if __name__ == "__main__":
    main()
```

9.1 What to notice

- **Everything goes through one `call` method**, so the rules that apply to every request live in one place: the bearer header, one refresh on `401`, and waiting out a short `Retry-After` on `429` and `503`. It stops retrying after a few attempts and then raises. A `Retry-After` of more than two minutes is a daily limit (`daily-write-limit` or `destructive-budget`, which run to midnight UTC), so it raises at once for the caller to tell the user rather than sleeping.
- **The refresh token is saved the moment it changes** (`save_tokens`). It is used once; losing the new one means the person has to sign in again.
- **Paging is a generator** that hands back `next_cursor` untouched.
- **The upload sends `If-None-Match: *` and a fresh `Idempotency-Key`**, takes its data as bytes (so a retry can send it again), and reads the `path` from the answer rather than assuming the name it asked for. For a big file, give `requests` an open file object instead, and don't retry from inside `call`: reopen the file and start again.
- **A copy is always asked for as an operation** (`Prefer: respond-async`), so there's one path whatever the size: a `202` is followed to its end with `wait`, which sleeps for `Retry-After` between polls and polls through `call`, so a refresh or a short throttle does not end the wait.
- **Errors are read by `type`**, not by text. `FoldrError.type` is the last part of the problem's `type` URL (`legal-hold`, `checked-out`, `not-found`, …). The `try` at the end shows the usual shape: some problems are "not now" answers to show a person, and the rest you re-raise.
- **A key per request.** The script makes a new key for every new request. If you retry a request yourself after a timeout, send the *same* key, so Foldr can recognise it.

What this example leaves out, and a real client shouldn't: storing tokens somewhere safe, checking `scope` in the token response before relying on `files.write`, a place to show a user the daily-limit problems it raises, and revoking the token (RFC 7009) when the person disconnects the app on your side.

10.0 Trying it in Postman

To try the API before you write any code, use the Postman collection that comes with this guide. It has a request for every operation, in the order you'd try them, with the sign-in set up once for the whole collection. Each request saves what the next one needs (a share id, a path, an ETag), so you can work down the folders by clicking **Send**.

Two files come with this guide:

File	What it is
`foldr-api-v1.postman_collection.json`	The collection: the requests, the OAuth settings and the scripts that carry values from one request to the next.
`foldr-site.postman_environment.json`	An environment: your site's address, the app's client ID, and the values the requests fill in as you go.

Postman's desktop app runs the requests itself, not from a web page, so the API's lack of CORS headers doesn't get in the way. The web version of Postman needs the Postman Agent for the same reason.

10.1 Before you start

You need a Foldr site with connected apps switched on, and a person on it who may connect apps. Both are an administrator's settings, under **Settings > Connected apps**: see What an administrator does.

10.2 Setting up

1. **Import both files.** In Postman, **Import**, and choose the two files. Then pick the **Foldr site** environment in the top right.
2. **Set `site`.** In the environment, set `site` to the address you sign in to Foldr at, with no trailing slash: `https://your-company.foldr.cloud`.
3. **Register Postman as an app.** Open **0. Set up** and send **Register this collection as an app**. It registers Postman with your site as a public app (the self-registration any app can do) and saves the `client_id` in the environment. You only do this once per site. If an administrator has already added Postman by hand, skip it and put the client ID they give you in `client_id` instead. Its redirect URL must be `https://oauth.pstmn.io/v1/callback`.
4. **Get a token.** Click the collection's name, open the **Authorization** tab, and choose **Get New Access Token**. Postman opens Foldr's sign-in page: sign in as usual and choose **Allow**. Back in Postman, choose **Use Token**.
5. **Check it works.** Send **0. Set up > Who am I** (`getMe`). You should see your own name.

The access token lasts an hour. When it runs out, every request answers `401 invalid-token`: choose **Refresh** beside the token, or get a new one.

10.3 Working through the folders

Folder	What it does	Scope it needs
0. Set up	Registers the app, shows the site's sign-in addresses, and `getMe`.	`profile`
1. Shares	`listShares` picks a share to work in and saves it as `share_id`.	`foldr.read`
2. Reading	Listings, an item, a download, a download link, search, custom fields and versions.	`foldr.read`
3. Making things	Makes a folder of its own, uploads a file into it, renames it, copies it, and follows the copy as an operation.	`files.write`
3a. Uploading straight to the storage	An upload session: open it, send the bytes to the storage, complete it.	`files.write`
4. Custom fields	Reads and sets a field value on the uploaded file.	`fields.write`
5. Sharing	Asks to share the uploaded file, and checks on the request.	`sharing`
6. Replacing, moving and deleting	Overwrites the file, restores an earlier version, moves the copy, and finally deletes the whole playground folder.	`files.overwrite`, `files.move`, `files.delete`

Run **List shares** before anything else. To work in a share other than the one it picks, copy that share's `id` into `share_id`. In the same way, **List a folder** lists the folder named in `folder` (`/`, the top of the share, to start with) and saves the first file it sees as `item_path` for the other reads.

Trying changes

A self-registered app can only read, so folders 3 to 6 answer `403 insufficient-scope` until the token carries the write scopes, and `403 write-not-allowed` once it does but the app's policy does not allow what the call asks. To try them:

1. An administrator opens the app (**Postman**) under **Settings > Connected apps > Policy** and allows what you want to try. Removing and replacing (folder 6) are allowed only in shares they name: name the share you'll work in.
2. Add the same scopes to `scopes` in the environment, for example `foldr.read offline_access profile files.write files.overwrite files.move files.delete`.
3. Get a new token, and agree to the new scopes on the consent page.

Folders 3 to 6 work in a folder of their own, named by `scratch_folder` (`/Postman playground` to start with), and the last request in folder 6 deletes it with everything in it. Send them in order. If you stop part way, delete the playground folder yourself afterwards, in Foldr or with **Delete the playground folder**, after **Read the playground folder** to fetch the ETag it needs.

Some of these can't be undone where the storage keeps no earlier versions and has no recycle bin: see Recoverability and audit. The collection only ever changes things inside the playground folder, but a request you edit can change anything the person and the app may change.

Sharing tells someone. The request in folder 5 waits for an approver, and the approver is told about it. Don't send it on a site where that will surprise people.

10.4 Reading the results

Each request checks its own status code, and Postman's **Test Results** tab shows whether it passed. When Foldr refuses a request, its problem type shows there too (for example `problem: precondition-failed`), so you can look it up in Errors.

Things you're likely to meet:

- **`412 precondition-failed`** on a change: the `If-Match` ETag is stale because the item has changed since it was read. Send the request that reads it again (each saves the new ETag), then the change.
- **`409 operation-finished`** on **Cancel the operation**: the copy had already finished, so there was nothing to cancel. That's normal for a small file.
- **`409 no-direct-upload`** on **Open an upload session**: the share's storage doesn't take direct uploads. Use **Upload a file** instead.
- **`404 not-found`** on a read: `item_path` or `share_id` names something the person can't see, or something that has gone. Set them again, or clear them and send **List shares** and **List a folder**.

10.5 Using a token from somewhere else

If you already have an access token (from the worked example, say), paste it into `access_token` in the environment and every request uses it. The collection's **Token** field reads that variable until you **Use Token** after **Get New Access Token**, which puts the new token in its place.

11.0 Troubleshooting and a checklist

11.1 Questions that come up

Every call answers `404` with `{"error":"not_found"}`

That isn't the API's own `404` (which is a problem document). It's what every address under `/api/v1` gives on a site where **Allow connected apps** is switched off (or the plan lacks it), on purpose, so the API looks as though it isn't there. Ask the Foldr administrator to turn it on in **Settings > Connected apps**. The OAuth endpoints answer the same way.

`401 invalid-token`, and I just got the token

Every token failure is the same answer, with no reason, so you can't tell which of these it is:

- the token has expired (they last an hour): refresh it;
- it's been revoked, or the person disconnected the app;
- the person has been disabled or removed, or has lost the right to connect apps (their tokens are refused while the right is missing);
- it's a token from another site: each site issues its own;
- it was bound to a different resource when you asked for it (a token tied to another API doesn't work under `/api/v1`);
- the header isn't `Authorization: Bearer <token>`, or the token was cut short by a copy and paste.

Refresh once and retry. If the refresh fails with `invalid_grant`, send the person through sign-in again.

`403 insufficient-scope` or `403 write-not-allowed`?

`insufficient-scope`: the **token** doesn't carry the scope this call needs (`WWW-Authenticate` names it). Check the `scope` in your token response. Either you didn't ask for it, or the app's policy didn't give it, so it was dropped at sign-in.

`write-not-allowed`: the token has the scope, but **this change isn't allowed here**: the person can't make it in Foldr, or the organisation hasn't allowed this app to make this kind of change, or the person may only read the approved version of a published file. Retrying doesn't help; tell your user.

A share I know exists answers `404`

`404` is deliberately the same for: no such share; a share that isn't this person's; a path that the share's rules hide; and a share your app's policy doesn't name (for changes). Check you're calling as the person you think you are (`getMe`), and that the share is in `listShares`. A share that opens only with the person's own password (some SMB and WebDAV setups) isn't visible to apps, and so `listShares` can be shorter than what the person sees in Foldr's web app.

`listShares` shows a share, but opening it gives `409 storage-account-not-connected` or `503`

`storage-account-not-connected`: it's in the person's own cloud account (Google Drive, OneDrive, Dropbox, Box) and they haven't yet given Foldr their sign-in to that service. They do that once in Foldr's web app, by opening the share there or under **Me › Services**; your app can't do it for them. `503 share-unreachable` or `share-unavailable`: its file server isn't answering just now; try again after `Retry-After`. Such shares are listed with `reachable: false`.

`428 precondition-required`, then `412 precondition-failed`

`428`: you left out `If-Match` (or `If-None-Match: *` on an upload). `412`: you sent it, and the item has changed since you read it. Read it again, and decide again. Don't loop on `412` by re-reading and re-sending the same change blindly: someone else changed the thing, and that may matter to your user. See ETags.

`If-Match: *` is refused

By design, for overwrite, move, delete and version restore: name the version you mean. `renameItem` and `setFieldValue` accept `*`, meaning "whatever the item is now"; send a specific tag anyway, so that you catch a change made since you read it.

A retry answers `409 conflict`

Either the name is taken, or the **same `Idempotency-Key` is still being handled**. The `detail` and timing tell which: if your first request is still running (a big upload, say), wait, then send the same request with the same key. A request that was cut off frees its key about two minutes after it last made progress.

A retry answers `422 idempotency-key-reused`

You sent the same key with a different request. Keys are for "send the same thing again". Make a new key for each new request.

`400 invalid-request` and I can't see why

Look at `detail`. The usual causes: a query parameter given twice (`?path=` twice, or a list); a `limit` of `0` or not a number; JSON that doesn't parse or is over 1 MB; no `Idempotency-Key` where one is required; an upload without `Content-Length`; `If-Match` on `uploadFile`; `If-Match: *` where a version must be named; a token request that isn't form-encoded.

The path has `#`, `&`, `%` or accents and it doesn't work

Percent-encode the value of `path` with your HTTP library's own query encoder, and don't build the query string by hand. Either Unicode form is accepted for accents. See Paths.

A download redirects somewhere and then fails

On shares with direct downloads, `getContent` answers `302` to the storage. Follow it **without** your Foldr token. A client that sends the bearer token to the storage host gets an error back from the storage. If your tool can't follow a redirect to another host, use `getDownloadUrl`.

A Google Doc downloads as `.docx` with no length

That's right: a native Google document is exported when read, so it has no fixed size, no `Content-Length` and no ranges. See Reading.

A folder listing is missing things, or shows things that were deleted

A listing is a snapshot kept for a minute, so changes made after the first page don't show in later pages. For the truth about one item, call `getItem`. And check `truncated`: a folder of more than 25,000 entries is paged as its first 25,000, and says so on every page.

Search returns nothing, or less than I expected

Check `not_searched` in the answer: shares that aren't indexed, can't be reached, or whose search index didn't answer are named there with a reason. Search covers indexed shares only, and only what the person may see.

A copy or move came back `202` and I don't know what to do

Poll the `Location` until it's no longer `202`. See Operations. If you've lost the id, `listOperations` finds it for 7 days after it ends.

An operation is `404`

It's more than 7 days since it finished, or it belongs to another person or another app, or your token has one of the operation scopes but not the one the operation was started under. These all look the same.

My change has been `202` for days

It's held for approval (`pending_approval`). Someone other than the person has to approve it in Foldr, and nobody has yet. It lapses after 14 days (unless the organisation set another period), when it ends `change-expired`. Poll slowly, at the `Retry-After` it gives. See Changes held for approval.

I'm told I have to connect again

The connection ended: the person or an administrator disconnected the app, the refresh token's lifetime or idle limit was reached, a used refresh token was presented again (which revokes the whole chain, so refresh from one place at a time), or the person left. Start the sign-in flow again.

`429` and my retry made it worse

Wait for `Retry-After`, don't retry everything at once afterwards, and back off further on repeated `429`s. Look at `X-RateLimit-Remaining` to slow down before you're stopped. `daily-write-limit` and `destructive-budget` run to midnight UTC: don't sleep through them, tell the user. See Rate limits.

`OPTIONS` is refused and a browser can't call the API

The API sends no CORS headers and answers `OPTIONS` with `405`. It's meant to be called from a server, a script or an automation platform. A web app should call it from its own back end.

Who do I tell about a problem?

Quote the `instance` code in the problem document (`#` and 8 hexadecimal digits) when you ask support, along with the time, the operation (`operationId`) and the OpenAPI document's `info.version`.

11.2 A checklist for a well-behaved client

Signing in

- [ ] Uses the authorisation code flow with PKCE (`S256`) and a random `state`, and checks `state` and `iss` on the way back.
- [ ] Sends `resource` set to the site's `/api/v1`.
- [ ] Asks only for the scopes it needs, plus `offline_access` if it should stay connected.
- [ ] Reads the `scope` in the token response, and copes with getting less than it asked for.
- [ ] Keeps tokens in a safe place, never in logs, URLs or source control.
- [ ] Saves each new refresh token before it uses the new access token, and refreshes from one place at a time.
- [ ] Refreshes on `401 invalid-token` once, and sends the person back to sign in on `invalid_grant`.
- [ ] Revokes its tokens (`/oauth/revoke`) when the person disconnects the app.

Calling

- [ ] Ignores fields it doesn't recognise, and accepts enum values it doesn't know (a share `type` it hasn't seen is `other`).
- [ ] Branches on a problem's `type` or `code`, never on `title` or `detail`.
- [ ] Keeps the canonical `path` from each answer and uses it for what follows.
- [ ] Percent-encodes `path` with a proper encoder, and never gives a parameter twice.
- [ ] Treats cursors as opaque, and handles `410 cursor-expired` by starting over.
- [ ] Checks `truncated` and `not_searched`.
- [ ] Honours `Retry-After` on every `429` and `503`, spaces out retries, and reads `X-RateLimit-Remaining`.
- [ ] Uses `If-None-Match` for repeat reads to get cheap `304`s.

Changing things

- [ ] Sends `If-Match` with the ETag it last read, and on `412` reads again instead of forcing.
- [ ] Sends a new `Idempotency-Key` for each new request, and the *same* key when it repeats one.
- [ ] Reads the `path` in a create's answer: `on_conflict: "rename"` may have chosen another name.
- [ ] Sends `Content-Length` and `If-None-Match: *` on uploads, and at most a few uploads at a time.
- [ ] Treats `202` as "started", follows `Location`, and handles every way an operation can end, including `source-remains`.
- [ ] Reads the item again after `503 write-failed` before assuming the change did or didn't happen.
- [ ] Tells its user, in words, about `legal-hold`, `checked-out`, `published`, `not-recoverable-here`, `change-declined` and the daily limits, and doesn't retry them.
- [ ] After a delete, tells the user from `recoverable` and `where` whether it can be brought back.
- [ ] Never presents a share request as sharing: it's a request, and an approver decides.

Staying stable

- [ ] Uses `operationId`s and the OpenAPI document, not hand-copied URLs, when it can.
- [ ] Watches for `Deprecation` and `Sunset` headers.