Personal API keys: publishing through REST without a password

Source file: doc/eng-us/guides/api-keys.md Last changed 2026-11-04
https://doc.exponential.earth/exphelp/viewer/guides/api-keys.md

This guide teaches the personal API keys of Exponential 6.0.15: a signed-in user makes a key on the site, a script sends it with every REST request, and the request runs as that user, limited to what the key was made for. It covers the whole life of a key (made, used, noted, rotated, revoked, expired), what an administrator switches on and sees, and how the REST layer turns a key into a user.

It is written for three readers. **Site users** who want a key for their own script read sections 1 to 4. **Administrators** who decide who may have keys and keep an eye on them read sections 5 to 8. **Developers** who build on the REST interface or extend it read sections 9 and 10. Security notes and troubleshooting (sections 11 and 12) are for everyone.

Every command and answer below was run against the demonstration server (Apache with PHP-FPM on port 443 and Velocity on 8080) on 5 October 2026 with throwaway accounts, which were removed afterwards. Keys, addresses and host names are shortened or replaced by documentation values (`example.com`, `203.0.113.0/24`).

Guides · Related: Remote services and apps (the `expservices` tokens) · ezprestapi (the REST content routes) · REST API v1 and v2 · Security and audit · Change note: doc/eng-us/bc/6.0/api-keys.md

In short

- A user with the policy `apikey/create` opens **My account > API access** (`/apikey/list`), names a key, ticks the scopes it needs and picks a lifetime. The key, `expk_<12 characters>_<40 characters>`, is shown **once**.
- A script sends it as `Authorization: Bearer expk_...`. The request runs as the key's owner, limited to the key's scopes, and never beyond the owner's own policies.
- Only an HMAC-SHA-256 hash of the secret is stored, with a salt per key. Nobody can show a key again, not even an administrator.
- Keys end when they are revoked (by the owner or an administrator), when they expire, or when the owner is disabled or removed. Revoking takes effect at the next request on every server.
- Administrators see every key at **Setup > oAuth admin > API keys** (`/oauthadmin/keys`), search and filter them, open one user's keys, and revoke with a confirmation. Every make, revoke, first use and refusal is in the audit trail as `access.apikey.*`.
- New in 6.0.15: the table `expapikey`, the module `apikey` with the policy `apikey/create`, the `[ApiKeySettings]` block of `rest.ini`. HTTP basic authentication of the REST interface now checks every password hash type.

Contents

1. What a key is, and what it is not
2. Getting a key
3. Using a key with curl: routes, permissions
4. Rotating and revoking your keys
5. Switching keys on for your users
6. Scopes
7. The administration: every key, one user, revoking
8. Auditing and limits
9. How the REST layer resolves a key
10. The data model and the upgrade
11. Security notes
12. Troubleshooting
13. Every setting

1. What a key is, and what it is not

A personal API key is	It is not
A long random string a user makes for one program of their own: an import script, a CI job, a desktop tool	A password: it cannot sign in to the site or the administration, and it never asks for one
Sent with each REST request in the `Authorization` header	A session: there is no cookie and no form token
The user, for that request, within the scopes chosen when it was made	More than the user: a key never gets a right its owner does not hold
Revocable on its own, without touching the user's password or other keys	An OAuth application: those are registered by an administrator for apps that sign *other* users in

There are three ways into Exponential from a program. Pick the one that matches the job:

You want	Use
A script of your own reads or publishes content through the REST interface (`/api/...`)	**A personal API key** (this guide)
A script calls the remote services (`/expservices/...`): bookmarks, sessions, content jobs	An `expservices` token, see Remote services and apps
An app or partner site lets *many* users sign in and act for them	An OAuth application, registered at **Setup > oAuth admin** (section 7)

> **Note**
>
> **Why not just the password?** A password opens everything the user may do, on every page, until it is changed, and changing it breaks every script at once. A key opens only the REST interface, only for its scopes, ends on its own date, and is revoked alone.

2. Getting a key

2.1 You need an account that may make keys

Keys belong to an account. If you have none, register first through the site's normal registration (`/user/register`) and activate the account. Then ask the site's administrator to allow you API keys if your account does not have that right yet: in a new installation no role has it (section 5).

Signed out, the API access page says so and offers both ways in:

Shell

```
$ curl -s https://example.com/apikey/list | grep -o 'Sign in to manage your keys\|Create an accou
nt'
Sign in to manage your keys
Create an account
```

2.2 Make the key

1. Sign in on the site and open **My account** (`/user/edit`). The list of account links has **API access** (with the number of keys you have) when you may make keys or already have some. Its address is `/apikey/list`.
2. Under **Make a new key**, give it a name that says where it is used ("Newsroom import script"). You will recognise it later by that name.
3. Tick only the **scopes** the program needs. The page offers only the scopes your own rights allow: each one names the permission it needs ("Needs your content/read permission").
4. Choose how long it stays valid: 7, 30, 90 or 365 days by default. Without an end date only where the site allows it.
5. Press **Make the key**.

The next page shows the key in a green box with **Copy key** and a ready-made `curl` line:

```
Your new key "Newsroom import script"
Copy it now. It is shown this once and is not stored anywhere: if you lose it, revoke it and make
 a new one.
expk_du8hrkekh7wy_... (here shortened: the page shows all 58 characters)
```

> **Note**
>
> **Shown once means once.** The page that shows the key is sent with `Cache-Control: no-store`, so neither the browser nor a proxy keeps it. Reloading it does not make a second key; it says the form was already sent. If the key is lost, revoke it (section 4) and make a new one.

2.3 What the list shows

Every key you have, newest working keys first, with its name, its first characters (`expk_du8hrkekh7wy_…`), its scopes, when it was made, when and from which address it was last used, and when it ends. A key shows **Active**, **Expires soon** (within seven days), **Expired** or **Revoked**.

3. Using a key with curl

Send the key in the `Authorization` header with the `Bearer` scheme. The examples read node 2 through the `ezp` content provider. The `ezprestapi` extension serves both versions: the reads answer at version 1 and version 2 (`/api/ezp/v1/...` and `/api/ezp/v2/...`) and the writes at version 2. The deprecated `ezprestapiprovider` 6.0.4 is a shim that maps to the same provider (REST API v1 and v2). The API access page shows the path of your installation in its example.

Shell

```
export EXP_API_KEY='expk_...'
curl -s -H "Authorization: Bearer $EXP_API_KEY" https://example.com/api/ezp/v2/content/node/2
```

```
HTTP/2 200
{"metadata":{"objectName":"Websites","classIdentifier":"folder","datePublished":1033917596,
 "dateModified":1791063800,"objectId":1,"nodeId":2,"fullUrl":"https:\/\/example.com\/websites"},"
links":{...}}
```

The same request through Velocity (port 8080 on the reference server) gives the same answer. The answers a script must handle:

Status	Body	Meaning
200	the content	the key is valid and has the scope the route needs
201	`{"message":"Created","objectId":...,"nodeId":...}`	a create was published
400	`{"error":"invalid_request","error_message":...}`	a field of a write is missing, or names a class or language that does not exist
403	`{"error":"access_denied","error_message":...}`	the owner may not read or write this content (content/read, create, edit, remove; 3.2)
404	`{"error":"not_found",...}`	the node or object does not exist
401	`{"error":"invalid_token"}`	no key, an unknown key, a wrong secret, a revoked key, a disabled owner, or a key sent in the address
401	`{"error":"expired_token"}`	the key passed its end date
403	`{"error":"insufficient_scope"}`	the key lacks the scope this route needs, or its owner lost the permission behind it
429	`{"error":"rate_limited",...}` with `Retry-After`	the key sent more requests this minute than the site allows

A key sent anywhere but the header is refused, even a valid one:

Shell

```
$ curl -s -D - -o /dev/null "https://example.com/api/ezp/v2/content/node/2?oauth_token=$EXP_API_K
EY" | grep -i '^HTTP\|www-auth'
HTTP/2 401
www-authenticate: OAuth realm='Exponential REST', error='invalid_token'
```

From Python, with nothing but the standard library:

Python

```python
import json, os, urllib.request

request = urllib.request.Request(
    'https://example.com/api/ezp/v2/content/node/2',
    headers={'Authorization': 'Bearer ' + os.environ['EXP_API_KEY']})
with urllib.request.urlopen(request) as answer:
    print(json.load(answer)['metadata']['objectName'])
```

3.1 Routes of the ezp provider

The `ezprestapi` extension serves the `ezp` provider. Reads answer at `v1` and `v2`; `v1` is the API that existing clients call (a mobile app reads `/api/ezp/v1/content/node/<id>/list/offset/0/limit/20?ResponseGroups=Fields`), and the links in its answers stay in `v1`. Writes answer at `v2` only. The full route table, OAuth2 and the configuration are in REST API v1 and v2: one extension.

Method	Path below `/api/ezp/v1` and `/api/ezp/v2`	What it does
GET	`/content/node/<id>` and `/content/object/<id>`	one node or object (`ResponseGroups=Metadata,Locations,Fields`)
GET	`/content/node/<id>/list(/offset/<n>)(/limit/<n>)(/sort/<key>(/asc|desc))`	the children
GET	`/content/node/<id>/childrenCount`, `.../fields`, `.../field/<identifier>`, `.../listAtom`	counts, fields, Atom (`listAtom` answers 500, a known issue)
POST	`/content/node/create` (v2)	create and publish below `parentNodeID`
DELETE or POST	`/content/node/delete/<id>` (v2)	remove the node's object with all its locations and what is below them
POST	`/content/node/<id>` (v2)	update: checks content/edit, then answers 501, updating is not implemented yet

Shell

```shell
# create a folder below node 1234 (fields: parentNodeID, classIdentifier, languageLocale, then th
e attributes)
curl -s -H "Authorization: Bearer $EXP_API_KEY" https://example.com/api/ezp/v2/content/node/creat
e \
     -d parentNodeID=1234 -d classIdentifier=folder -d languageLocale=eng-US -d name='From a scri
pt'
{"message":"Created","objectId":5678,"nodeId":4321}

# remove it again
curl -s -X DELETE -H "Authorization: Bearer $EXP_API_KEY" https://example.com/api/ezp/v2/content/
node/delete/4321
{"message":"Removed","nodeId":4321,"objectId":5678}
```

A route whose path matches but which does not take the method no longer ends the search: the router tries the routes after it, and answers `405` with an `Allow` header only when none takes the method. A refused write answers `401` or `403`, never `405` (before 6.0.15 the authentication answers of a `POST` or `DELETE` turned into `405`).

3.2 Permissions of the REST interface

Every read and write of the `ezprestapi` content controller asks the current user's policies the way the content module asks them, whatever the request was authenticated with: an OAuth token, a personal API key, HTTP basic authentication, or the anonymous user when authentication is off. One helper of the kernel does it, `expRestContentPermission` (`kernel/private/rest/classes/auth/content_permission.php`), and the key guard of section 6 asks the same helper, so a key and a token get the same answer.

Call	Policy	Checked like
read a node, an object, a list, a count	`content/read` of the node or object (a list also of its parent; the children are filtered by the fetch)	`canRead()`
create	`content/create` of the class below the parent, in the language: Class, ParentClass, Section, Node, Subtree and Language limitations	the parent object's `checkAccess( 'create', ... )`, as `content/action` does for **New**
update	`content/edit` of the node (and `user/selfedit` for one's own user), in the posted language	`canEdit()`
remove	`content/remove` of every location of the object and of everything below them	`canRemove()` of each location and `can_remove_all` of `subtreeRemovalInformation()`, as `content/removeobject`

A refusal answers `403` with `{"error":"access_denied","error_message":"You may not create 'article' below node 60 in eng-US (content/create)."}` and changes nothing. The content module has no `content/publish` check, and neither has the REST interface: a create publishes when `content/create` allows it, and the publish workflow runs as for the content module.

The REST interface's answer cache (`rest.ini [CacheSettings] ApplicationCache`) keeps one answer per user and only for `GET` and `HEAD`: before 6.0.15 one user's read was served to the next for ten minutes, past `content/read`, and a repeated create or delete was answered from the cache without running. A refusal is never cached (it follows the rights of each request), and a cache file that cannot be read back is generated again rather than answered with 500.

> **Note**
>
> **Keep the key out of code.** Put it in an environment variable, your CI's secret store or your system's keychain, never in a repository, a shared script or a URL. Every key starts with `expk_`, so a secret scanner can be taught to look for it.

4. Rotating and revoking your keys

Rotating is making the new key before the old one ends: make a second key with the same scopes, put it in the program, check that the program works (the new key's **Last used** fills in), then revoke the old one. Two keys overlap for as long as the switch takes; nothing stops in between.

Revoking ends a key at once: press **Revoke** beside it, then **Revoke the key** on the question that follows. Every request with it is refused from the next one on, on every server. A revoked

key stays in your list, marked **Revoked** with the time, so you can see what it was. It cannot be made valid again.

Revoke a key straight away when:

- it may have been seen by someone else (pasted in a chat, committed, printed in a log);
- the program it was for is retired;
- the laptop or server it lived on is lost or handed on.

An administrator may also revoke your keys; you see them as revoked in your list.

5. Switching keys on for your users

Keys are on in every installation (`rest.ini` `[ApiKeySettings]` `ApiKeys=enabled`), but nobody may make one until a role gives the policy:

Policy	Grants	Limitation
`apikey/create`	making keys on `/apikey/list`	Scope: which scopes a key may be given (for example `read` only)

Seeing and revoking one's own keys needs no policy: any signed-in user may do that, so a user whose right was taken away can still clean up. Anonymous visitors can never make keys, whatever their role says.

To let your members publish through the API:

1. **Setup > Roles and policies**, open the role your members have (Member in a standard installation) or create a role "API publishers".
2. **New policy**, module **apikey**, function **create**. Leave the limitation empty to allow every scope, or choose **Scope** and the scopes they may give a key.
3. Make sure the role also has what the scopes stand for: `content/read` for read, `content/create` for publish, `content/edit` for edit, `content/remove` for remove (section 6). A scope whose permission the user lacks is not offered.
4. Assign the role to the users or group.

The **API access** link appears on their profile at the next page view. To switch keys off for the whole site, set `ApiKeys=disabled`: no key is accepted any more and none can be made; the pages say so.

> **Note**
>
> **Members of a public site.** Self-registration plus `apikey/create` for Members means anyone who registers can make a key. That is fine for reading, and it is what a community site wants. For publishing, give `apikey/create` with the `publish` scope to a separate role that you assign by hand, and keep `content/create` limited to the subtrees those users may write into.

6. Scopes

A scope is what a key may be used for. Each one stands for a policy of the owner, named in `rest.ini`:

INI

```
[ApiKeySettings]
Scopes[read]=content/read
Scopes[publish]=content/create
Scopes[edit]=content/edit
Scopes[remove]=content/remove
ScopeNames[read]=Read content
ScopeNames[publish]=Create and publish content
```

A route of the REST interface needs one scope:

INI

```
DefaultReadScope=read
RouteScopes[ezp7xRestContentController_CreateContentNode]=publish
RouteScopes[ezp7xRestContentController_UpdateContentNode]=edit
RouteScopes[ezp7xRestContentController_DeleteContentNode]=remove
```

A `GET` or `HEAD` route that is not listed needs `DefaultReadScope`. Any other route that is not listed is refused for keys, so a new write route of an extension is closed to keys until its scope is named.

Three checks stand between a key and a route, on every request:

1. The key has the scope the route needs.
2. The owner still holds the scope's policy (`hasAccessTo()` is not `no`): a member who loses `content/create` loses every key's publish scope at the same moment.
3. For a write route listed in `RouteGuards[]`, the owner's rights on the node it writes: `create` checks `content/create` of the posted `classIdentifier` under the posted `parentNodeID` (Node, Subtree, Class and Section limitations apply), `edit` and `remove` check the route's node.

The request then runs as the owner, so every policy check of the controller applies as well.

> **Note**
>
> **Why the third check.** Before `ezprestapi` 1.2.5 its content controller created and removed through `nxc_powercontent` without asking the permission system. It now checks every call for every kind of authentication (3.2); the guard stays as a second line and asks the same helper, so the two never disagree. Only a refusal of the rights stops a key here (`403 insufficient_scope`, recorded as `access.apikey.use.failed` with the reason `permission`); a write that names no node, or an unknown one, goes on to the controller, which answers `400` or `404` as it does for a token.

7. The administration: every key, one user, revoking

Setup > oAuth admin (`/oauthadmin/list`) has two tabs.

Applications are the OAuth clients. Each card shows the client identifier, the owner, the last change, the endpoint, how many users authorized it and how many of its tokens are still valid. **New application** creates one; its identifier and secret are random (`random_bytes()`) and made when it is first stored. The secret is folded away on the application's page until **Show the secret** is pressed. Removing an application also removes its authorizations, tokens and codes. All addresses, form fields and buttons of the earlier pages are unchanged.

API keys (`/oauthadmin/keys`) lists every key of every user:

- figures: active, expiring within seven days, active but never used, expired, revoked (each a filter);
- a search over key names, prefixes, logins and e-mail addresses, and a status filter;
- per key: name and prefix, owner (the link opens that user's keys), scopes, last use and address, end date, status;
- **Revoke** on a row, or tick several and **Revoke selected**; a confirmation page names the keys before anything happens.

One user's keys are at `/oauthadmin/keys/(user)/<user id>`. The user's settings page in the administration (**Users > the user > Settings**, `/user/setting/<id>`) shows how many keys the user has and links there.

Who sees these pages: everyone with `oauthadmin/*` (the Administrator role). The figures on a user's settings page are shown only to such users.

8. Auditing and limits

Four events of the audit's `access` channel, on by default (audit):

Event	When	Records
`access.apikey.create`	a key is made	the key's id, name, prefix and owner; its scopes and end date
`access.apikey.revoke`	a key is revoked	the same; who revoked it (by: owner or administrator)
`access.apikey.use`	a key is used for the first time	the key; its scopes
`access.apikey.use.failed`	a key is refused	the reason: `malformed`, `unknown`, `secret`, `revoked`, `expired`, `owner`, `transport`, `scope`, `policy`, `permission`, `rate_limited`; the route for a scope refusal

Never recorded: the secret, its hash or its salt. An unknown key is recorded by the prefix it claims. A key over its rate limit is recorded once per minute, not once per refused request.

Shell

```
$ ./console exp:audit tail --channel=access --name='access.apikey.*' --lines=3 --allow-root-user
2026-10-06 05:34:31.613  access   access.apikey.use.failed   anonymous(10)          apikey 13 Liv
e test second     refused  r-01M47V9N5X...
2026-10-06 05:40:34.614  access   access.apikey.use          apishotsdaf3ad0b(30320) apikey 18 Th
rowaway probe key  success  r-01M47VMQNP...
2026-10-06 05:40:34.861  access   access.apikey.use.failed   anonymous(10)          -
                  refused  r-01M47VMQXD...
```

The first line is a revoked key tried again (the actor is anonymous: a refused key signs nobody in). The second is a key's first use, recorded for its owner. The third is the same key sent in the address: refused before any lookup, so it names no key (reason `transport`). Shortened: the request and event ids.

Last use. A key's last use and address are written at most once a minute (`LastUsedInterval`), so a busy key costs no database write per request. The address is the client's, through trusted proxies (`eZSys::clientIP()`, see trusted proxies).

Limits.

Setting	Default	Effect
`MaxKeysPerUser`	10	keys that still work per user; 0 = no limit
`MaxExpiryDays`	365	the longest lifetime; 0 = keys without an end date are allowed
`RateLimitPerMinute`	120	requests per key and minute; 0 = no limit. Counted in APCu, per PHP-FPM pool or per Velocity server; without APCu it is not enforced and the API keys page says so

9. How the REST layer resolves a key

For developers. The classes are in the kernel autoload array.

1. `ezpRestAuthConfiguration::filter()` (`kernel/private/rest/classes/auth/auth_configuration.php`) starts each request with `expApiKeyRest::reset()`: a persistent worker (Velocity) keeps statics, and the key of the previous request must not be.
2. The authentication style asks `expApiKeyRest::authentication()`. Both kernel styles do: `ezpRestOauthAuthenticationStyle` (the default, `AuthenticationStyle` in `rest.ini`) and `ezpRestBasicAuthStyle`. When the `Authorization` header is `Bearer expk_...` (or `OAuth expk_...`), the answer is an `ezcAuthentication` with `expApiKeyAuthFilter`. A key in `oauth_token`, `access_token` or `api_key` of the query or body is refused without a lookup. Anything else goes to the style's own filter, as before.
3. `expApiKeyAuthFilter::run()` parses the key (`expk_` + 12 `[a-z0-9]` + `_` + 40 `[A-Za-z0-9]`), loads the row by its prefix, compares the secret with `hash_equals()`, checks status, owner, failed-login lock and rate limit, notes the use and records the first one. Its statuses are those of `ezpOauthFilter`, so the REST layer's answers (401 `invalid_token`, `expired_token`) are the same as for OAuth tokens.
4. The style returns the owner; `filter()` signs the request in as that user.
5. `filter()` then calls `expApiKeyRest::authorize()` with the routing information: scope, policy, guard (section 6). A refusal is an internal redirect to `/auth/oauth/login` with `insufficient_scope`: 403.

PHP

```
// A route of your own extension, open to keys with the publish scope:
// extension/myext/settings/rest.ini.append.php
// [ApiKeySettings]
// RouteScopes[myExtRestController_doImport]=publish
```

Key values that need no database are static and take their settings as arguments, so they are easy to test: `expApiKey::newToken()`, `parseToken()`, `hashSecret()`, `secretMatches()`, `statusFor()`, `expiryFor()`, `normaliseScopes()`, `limitScopes()`, `allowedScopes()`, `routeScope()`. Templates have `fetch( 'apikey', 'can_create' )` and `fetch( 'apikey', 'counts', hash( 'user_id', <id> ) )` (the current user's own, or any user's for an `oauthadmin` administrator).

Basic authentication (`AuthenticationStyle=ezpRestBasicAuthStyle`) now checks the password with `expRestPasswordAuthFilter`: the user by login (or e-mail, when `AuthenticateMatch` allows), enabled, published, not locked, and

`eZUser::authenticateHash()` with the user's own hash type. Until 6.0.15 it compared `md5("login\npassword")` with `ezuser.password_hash` in SQL, so every user on a modern hash (bcrypt or argon2, `php_default`, the default since 6.0) was refused. A wrong password now counts as a failed login.

10. The data model and the upgrade

One table, `expapikey`, one row per key:

Column	Holds
`id`	the key's number
`user_id`	the owner (content object id of the user)
`name`	the owner's name for it (at most 100 characters)
`key_prefix`	expk_<id>, unique: the public part, shown in lists and logs
`secret_hash`, `salt`	HMAC-SHA-256 of the secret keyed with the salt (16 random bytes, hex), 64 hex characters
`scopes`	the scope ids, space separated
`created`, `created_by`	when, and by whom
`expires`	end date, 0 = none
`last_used`, `last_ip`	the last noted use (at most once per `LastUsedInterval`)
`revoked`, `revoked_by`	when, and by whom; 0 = not revoked

A new installation gets it from the schema of its engine (`share/db_schema.dba`, used by every installer, and the kernel schemas of MySQL, PostgreSQL and SQLite). An existing 6.0 installation adds it with the update file of its engine (`update/database/<engine>/6.0/dbupdate-6.0.0-6.0.15.sql`; written so that a second run changes nothing) or, on any engine including Oracle and MongoDB, with:

Shell

```
php update/common/scripts/6.0/createapikeytable.php --dry-run --allow-root-user   # says whether it is missing
php update/common/scripts/6.0/createapikeytable.php --allow-root-user
```

```
Database: sqlite, table expapikey: missing
created expapikey
```

Then regenerate the kernel autoloads (`php bin/php/ezpgenerateautoloads.php -k`), clear the INI, template and template-override caches, reload PHP-FPM and restart Velocity.

11. Security notes

- **Nothing to steal from the database.** The secret is 40 characters from 62 (about 238 bits) and only its keyed hash is stored. A copy of the table cannot be turned back into working keys.
- **Constant-time.** A presented secret is compared with `hash_equals()`, and an unknown or

malformed key spends the time of a hash too, so timing does not tell how close a guess was. The same was done for OAuth client secrets (`ezpRestClient::validateSecret()`), which used `===`.

- **Never more than the owner.** Scopes are checked against the owner's policies when the key is made and again on every request, and write routes against the owner's rights on the node.
- **Header only.** Keys in addresses end up in access logs, proxies and browser history; they are refused there.
- **CSRF.** Every form of the API access page and the administration posts the form token. The API access page needs no policy of its own (`PolicyOmitList[]=apikey/list`) and checks the sign-in and `apikey/create` itself; it never touches another user's key, even with a forged id.
- **Shown once.** The page with a new key is sent with `Cache-Control: no-store`; the form token filter keeps that stricter header instead of its own. A one-time nonce stops a reload from making a second key.
- **Not in the response cache.** The page is a module view, never stored by the HTTP cache, which keeps content views only.
- **The application secrets.** OAuth client secrets are still stored as they are (an OAuth client must be able to read them back on its page); the page folds them away until asked.

12. Troubleshooting

Symptom	Cause and fix
No **API access** link on the profile	The user has no `apikey/create` and no keys. Give the policy (section 5)
"Your account may make keys, but none of the scopes ... is open to you"	The role has `apikey/create` but not `content/read` (or the other policies), or a Scope limitation names none of them
401 `invalid_token` with a fresh key	The key was cut while copying (it is exactly 58 characters), or sent in the address, or the header is `Authorization: expk_...` without `Bearer`
401 on one server only	A proxy in front of it drops the `Authorization` header. Apache with PHP-FPM needs it passed (`SetEnvIf Authorization "(.*)" HTTP_AUTHORIZATION=$1`, or `CGIPassAuth On`)
403 `insufficient_scope`	The route needs a scope the key lacks, the owner lost the policy, or the write is outside the owner's rights. The audit's `access.apikey.use.failed` names which (`scope`, `policy`, `permission`) and the route
405 "This method is not supported" on `/content/node/create`	Before 6.0.15 a refused `POST` or `DELETE` (no or a wrong key) was redirected to an authentication route that took `GET` only. Upgrade the kernel and `ezprestapi` 1.2.5; `v1` is read only, a write there still answers 405
403 `access_denied`	The user behind the token or key may not do this to this node (3.2); the message names the policy
429	Over `RateLimitPerMinute`; wait for `Retry-After` seconds or raise the setting
A database error on `/apikey/list`	The table is missing: run `createapikeytable.php` (section 10)

Symptom	Cause and fix
Velocity answers differently from Apache	Velocity loads classes at its start; restart it after the upgrade (`./console exp:velocity restart`)

13. Every setting

`settings/rest.ini`, block `[ApiKeySettings]`; override in `settings/override/rest.ini.append.php` or an extension's `rest.ini.append.php`.

Setting	Default	Meaning
`ApiKeys`	`enabled`	`disabled`: no key is accepted or made
`ExpiryChoices[]`	7, 30, 90, 365, 0	the lifetimes offered, in days; 0 (no end) only when `MaxExpiryDays=0`
`DefaultExpiryDays`	90	preselected lifetime
`MaxExpiryDays`	365	longest lifetime; 0 = no limit
`MaxKeysPerUser`	10	working keys per user; 0 = no limit
`LastUsedInterval`	60	seconds between two writes of a key's last use
`RateLimitPerMinute`	120	requests per key and minute (APCu); 0 = no limit
`Scopes[<id>]`	read, publish, edit, remove	the policy (`module/function`) a scope stands for
`ScopeNames[<id>]`		the scope's name on the pages
`DefaultReadScope`	`read`	the scope of an unlisted GET or HEAD route; empty = none
`RouteScopes[<Controller>_<action>]`	the ezprestapi write routes	the scope a route needs
`RouteGuards[<Controller>_<action>]`	`create`, `edit`, `remove`	the owner's rights checked on the node a write route acts on
`ExamplePath`	empty	the REST path the API access page uses in its curl example

Also: `site.ini [RoleSettings] PolicyOmitList[]=apikey/list`, `module.ini ModuleList[]=apikey`, `admininterface.ini ItemsPerPage[oauthadmin/keys]=25`.

> **Become an Exponential Partner**
>
> Partners share their information, provide funding, contribute back and develop the products with us, and that is how Exponential grows. A partner website and dashboard are planned for mid-2027: please become a partner today to speed up our growth rate and land new features faster.
>
> Partners: start here | How to apply

Exponential Copyright © 1998-2026 Exponential Foundation and 7x
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".
This documentation is free documentation, licensed under the GNU Free Documentation License 1.3 or later:

https://www.gnu.org/licenses/fdl-1.3.html.