Klevio API Docs V2

2022-10-18

1. Basic concepts

The API was initially designed as a simple way to issue multiple keys at one property for people to use in their Klevio phone app. Since then it has grown to support other use cases, such as using it to implement a different user interface and use it only to actuate owned locks. This section will try to present the basic structure and the intended uses of it, which does not mean this is the only way it can be used - if you are using it in a different way, please report back, so we can add it either here or at least to our testing procedures.

1.1 Data model

The Klevio system works as a permission tree, where keys represent which permissions keyholders have on locks (also called doors). For the API, the locks are further grouped into properties, so it is easier to access all property keys at the same time.

When a Klevio device is installed, a master key is issued. This key has the permission to unlock the door and to share the access to that lock further. It is valid from the time of issue and into eternity.

To issue a key, you will need to provide a way to identify a key with a sharing permission. When issuing a key through the API, you can only issue non-shareable keys (for issuing shareable keys, use the dashboard).

On the API, the keys come with a set of objects that give further information about the key. These are the property and lock objects, the keyholder (a user object), the validity object (which specifies when the key is valid), and the meta object used to store other pieces of data.

The permissions part of the keys is immutable, so when changing key validity, a new key is issued and the existing key obsoleted.

Since a key represents keyholder’s permissions on a lock, you cannot issue a key to a third party and then use that key to unlock the door - only the keyholder can do that. You can, however, use your own key for the same lock.

1.2 Usage scenarios

In all scenarios you will need a way to match properties or locks. This is done by either storing relevant key ids in your system, or by using the propertyId and/or externalId in Klevio systems based on existing immutable information in your systems.

For each of the scenarios you will have to build an API integration, but the scenarios differ in how many user interfaces and administration tools you will have to build.

1.2.1 Scenario I: Users use the Klevio app

Additional requirements: user email passed to Klevio.

If your users will use the Klevio app to access properties, then the main methods you will use will be `grantKey`, `changeKeyValidity` and `deleteKey` (if you use keypads also `getPin`).

You will match properties or locks in your system to Klevio by using propertyId or externalId, then when needed, you will issue keys with `grantKey` (possibly call `getPin` to load and show pin if using keypads). If the user booking changes, you will use `changeKeyValidity` to change the validity of the keys, and if they cancel use `deleteKey` to remove access.

To use key methods (`getPin`, `changeKeyValidity`, `deleteKey`) you will need key ids, which you can get by calling `getKeys` and using the same source and user objects you used with `grantKey`. This means that you don't really need to store any extra information on your side as long as your admins can see that the key issue succeeded.

1.2.2 Scenario II: Users user your own app

Additional requirements: build own permissions and time limitation tools; build user interface for locking and unlocking.

When users use your own app, there is no need for the users to exist in Klevio systems. This means that you will mostly use the `useKey` method to open doors.

If you can store Klevio key information in your system, you can store keyId for each door and you can directly call `useKey` with it.

If you cannot store Klevio key information in your system, you can store your system information in Klevio systems by either using the `propertyId` or the `externalId` fields. In

this case, you will have to call `getKeys` with either the lock or the property object as the source object, and then call `useKey` with the retrieved key ids.

The downside of this approach is that you need to manage permissions and time limitations on your side and use Klevio only as a remote control for locks. You will also likely want to store your own audit trail of key usage.

1.2.3 Scenario III: Users user your own app, but with Klevio managing permissions and time limitations

Additional requirements: pass a hashed immutable user id to Klevio; build own user interface for locking and unlocking.

If you want to use Klevio to handle permissions and time limitations for users, you will use Klevio to issue keys to a virtual user, but you will then use your own keys to unlock doors, when these keys exist.

This means that you will issue keys to a generated internal email address (you should use notify=false to not send emails), such as `kleviouser+[user id hash]@example.com` that will represent your user in the Klevio system. This means that you need to base this email on an immutable property in your system.

You will then use getKeys, `getPin`, `changeKeyValidity` and `deleteKey` to manage the access for this user the same way you would in Scenario I.

When your user wants to open a door, you will however call `getKeys` to confirm whether the user has permission to unlock, but instead of using the `keyId` of the returned key, you will use the `externalId` of its lock to fetch your own key for that door with another `getKeys` call. You will then use these keys to unlock the door with `useKey`.

This gives you the benefit of being able to issue time limited pin codes for keypads and your admins a way to check, remove or modify access through the Klevio dashboard.

1.3 Enabling API access for new devices

If you want to have all newly installed devices API enabled by default, you will need to reach out to Klevio support to have this feature enabled on your account. This will enable you to call `getNewKeys` and then `enableNewKey` so that you can either share or use these keys.

New keys are keys that do not have a `propertyId` or an `externalId` set, so cannot be used via the API in any way other than with `enableNewKey` to set either or both of those properties. When at least one of these is set, you can start using that key in other methods.

2. Setting up API access

In the dashboard, set a public key (currently only EC256 is supported) for your account. When you set a public key, an `apiKeyId` will be assigned (confusingly displayed as External ID on the Token overlay in Dashboard), you will need it to access the API. To start off, we recommend you mark your first token as a testing key. This will let you use test data and some additional test endpoints. When you are done, add the public key again as a full "Public API" type token.

For each Klevio key in the dashboard you want to access, you can set two string identifiers:
- `propertyId`
- `externalId`

`propertyId` should be used as grouping for different keys that comprise a certain property. By using it, you can issue keys for all locks at the property at the same time.

`externalId` identifies a lock and should be used to provide easier issuing of keys without storing any information about Klevio objects inside your own system.

If you intend to use `propertyId` or `externalId` so that they are connected to data inside your own system, consider using a hashing function so as not expose internal details of your system inside Klevio systems. We suggest using one of the functions for storing passwords like pbkdf2, scrypt or bcrypt.

3. Connecting to the API

Root address: **https://api.klevio.com/sl/v2**

Request format

a) HTTP Method: POST

b) HTTP Headers:
```
X-KeyID: [apiKeyId]
Content-Type: application/jwt | application/x-www-form-urlencoded
```

c) Request body:
Request body depends on the `Content-Type` HTTP header:
- "`application/jwt`": [jwt]
- "`application/x-www-form-urlencoded`": jwt=[jwt]

JWT format

a) JWT Header including the kid parameter (key algorithm should be ES256):
```
jwt_header = {
      "alg": [key algorithm],
      "typ": "JWT",
      "kid": [apiKeyId]
}
```
b) JWT payload with an added rpc parameter that includes the JSON RPC 2.0 message (`jsonrpc:"2.0"` default can be omitted):
```
rpc = {
      // RPC request id; response will have the same id
      "id": [rpc id],
      // RPC method name
      "method": [method name],
      // RPC method parameters (object or list)
      "params": [method params]
}
jwt_payload = {
      // name of the issuer, characters allowed: 32 <= code <= 126
      "iss": "[request issuer]",
      // if a different audience is used => error
      "aud": "klevio-api/v2",
      // issued at time
```

```
    "iat": [issued at],
    // OPTIONAL; your id of the request token, used for debugging
    "jti": [jwt token id],
    // OPTIONAL; expiry; not more than 30s in the future
    // DEFAULT: exp = iat + 5
    "exp": [expires at],
    // OPTIONAL; not before; if server time < nbf => error
    // DEFAULT: nbf = iat
    "nbf": [not before],
    // JSON RPC message
    "rpc": rpc
}
```

Server public key

To validate the server response, use this public key:

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEq5YAUxJ8BePUBVrxEhXJkFwGNuxM
XCV2mfyFtrwlexnf3+kWPmY6dQDPWBT+G5oeVWfCIstmuGJEZGN2cSXDAw==
-----END PUBLIC KEY-----
```

4. API Objects

Objects that can be used as input parameters have a `$type` property that defines the object type and an identifier field (both marked as bold below).
When dates are used, they are in the ISO form of `YYYY-MM-DDThh:mm:ssZ`, so should be passed in UTC timezone - when passed as inputs make sure you convert them to UTC with the correct time to avoid UTC offset issues around DST changes.

userObject

User object will be returned full, but as input it can either have an id or email. If both are specified, an error will be returned.
`{"$type": "user", "id": "[user id]", "email": "[user email]", "meta": metaObject}`

propertyObject

`{"$type": "property", "id": "[propertyId]"}`

keyObject

`{"$type": "key", "id": "[key id]", "property": propertyObject, "keyholder": userObject, "lock": lockObject, "meta": metaObject, "validity": validityObject}`
Some key details will be passed back in the metaObject, such as *`keypad_active`* and *`pin`* (if set).

lockObject

`{"$type": "lock", "ex_id": "[external_id]", "name": "[lock_name]", "is_connected": [bool], "available_actions": ["...", ...], "has_keypad": [bool]}`

sourceObject

A propertyObject, keyObject or lockObject - it only needs to include the `$type` and the identifier field (`id` for key and property and `ex_id` for lock) properties to identify the required object. Used as an argument to getKeys, grantKey and getState.

validityObject

When returned it will have both fields, but as input it can have any combination of fields (if empty, key is not limited).
`{"from": "YYYY-MM-DDThh:mm:ssZ", "to": "YYYY-MM-DDThh:mm:ssZ"}`

metaObject

Meta object is a simple object with key-value pairs, returned verbatim. The object could include other fields that are not passed by the user (`keypad_active` and `pin` for example).
`{“[key]”: “[value]”,...}`

errorObject

Error object is returned in the error field, not in the result field, as per json-rpc spec.
`{"message": "[error message]", "code": [error code]}`

stateObject[1]

`{"lock": lockObject, "battery_level": {"value": [percentage], "time": "YYYY-MM-DDThh:mm:ssZ"}, "state": {"value": "[locked|unlocked]", "time": "YYYY-MM-DDThh:mm:ssZ"}, "details": metaObject}`

[1] battery_level and state support not yet added

5. RPC

RPC endpoint is `/rpc`, requests should be HTTP POST. Responses will be signed JWTs with specified `aud` (set to the value of the incoming `iss`) and `iss` (set to "`klevio-api/v2`")

Common `errorObject` codes:

- `-32602` -> parameters errors
- `400` -> bad request, sending in wrong parameters
- `403` -> permissions error, requesting something you don't have permission to
- `404` -> object not found
- `500` -> unresolved upstream error

getKeys(source=sourceObject[, user=userObject])

If the `sourceObject` is a `keyObject` it must be a master key or a shareable key and will return keys for a lock that the specified key relates to.

If `userObject` specified, return keys for that user, otherwise return keys for API user.

Response will include a list of `keyObjects` or an empty list if none is found.

Reasons for empty list:

- No keys that have api access enabled
- No keys owned by the specified user

Common errors:

- `404` -> specified key does not exist
- `400` -> specified key is not a master key (if source is a `keyObject`)
- `400` -> bad source object specified

getKey(key=keyId)

Response will be a `keyObject` or `null` if no key with the specified id is found or you do not have the rights to view that key.

grantKey(source=sourceObject, user=userObject, validity=validityObject[, meta=metaObject, issue_pin=bool])

If the `sourceObject` is a `keyObject` the key must exist and have grant permission, in other cases all keys referenced by the `sourceObject` should have grant permission.

If the user already exists and is active as a Klevio user, they will get an email notification that they received a new key. They may also get an app notification if they have enabled notifications in their app. The email may be suppressed if required (see below). If the user does not already exist, they will receive an onboarding email from Klevio to set a password for their new account.

In `validityObject` if `from` is not included it is set to now. If it is in the past, it needs to be inside the last 24 hours to be valid, but will be set to now by the server. If `to` does not exist, the key does not expire and must be manually deleted to disable access or updated with `changeKeyValidity` later.

`metaObject` can be used to store reservation data or other details. It cannot have more than 10 fields with 1kB size each.

There are special meta properties that are used in key creation and not saved in the meta object (they are however used when evaluating `metaObject` limitations):

- `notify` set to False will not send an email to the user
- `key_name` string will define the name for the issued key(s)
- `key_description` string will define the description for the issued key(s)

If you wish to set a different name and description for each key, you should issue them independently.

Response will be a **list** of **new** `keyObjects`.

Common errors:

- `404` -> specified key or property id does not exist
- `400` -> specified key has no keypad (when source object is a `keyObject`)

changeKeyValidity(key=keyId, validity=validityObject)

Response will be a `keyObject`. If the call does not result in a change, the response will be the existing `keyObject` with the same `keyId`. If there is a change, the response will be a **new** `keyObject` with a new `keyId`.

Common errors:

- `404` -> specified key does not exist
- `400` -> can't change master key validity

useKey(key=keyId[, action=actionName])

This method runs the `unlock-lock` action on the lock (duration depends on the setting on the device itself). If action is specified, it will be run instead of the default. `actionName`

corresponds to one of the values returned in `keyObject.lock.available_actions`. Response will be a `boolean` denoting whether the lock action was successful or an `errorObject`.

Common errors:

- `404` -> specified key does not exist
- `403` -> user is not a key owner
- `400` -> action not available

deleteKey(key=keyId)

Response will be `boolean` denoting the success of the action or an `errorObject`.

Common errors:

- `404` -> specified key does not exist
- `400` -> can't delete master key

getState(source=sourceObject)

Response will include a list of `stateObjects` or an empty list if none are found.

Reasons for empty list:

- No keys with specified property id
- No keys with specified key id

getPin(key=keyId)

Response will return a key pin as a string.

- `404` -> specified key does not exist or property id is not set
- `400` -> specified key does not have an active keypad

getNewKeys()

Response will return a list of new keys - keys that have neither a propertyId or an externalId set.

Reasons for empty list:

- There are no keys that have API access enabled and don't have any of the ids set up

enableNewKey(key=keyId[, propertyId][, externalId])

This will enable a new key for unlocking via the API by setting a `propertyId` or an `externalId`. You can set both values at the same time or set just one of them. As long as a value is not set, you can set it later, but you will need to know the key id.

`propertyId` and `externalId` should both be strings. We suggest using a hashing mechanism to make sure your internal data is not exposed in Klevio systems.

If you try to set either of the properties to the currently set value, the response will be a `304`.

Common errors:

- `404` -> specified key does not exist or api access not enabled
- `403` -> trying to change a set property

6. Testing the API

You can test various aspects of the API at the following endpoints. These are only enabled while in the development stage and will be disabled once integration is marked as production.

/test/jwt/parse

Will respond with an object with debug information on how the received JWT is understood/parsed by the server. Debug this if you're getting errors regarding JWT.

/test/jwt/valid

Will respond with a JWT, signed by the server including server current time. Use this to check server time and validate that you can successfully validate the server signature.

/test/jwt/invalid/[header|body|signature]

Will respond with a JWT where [header|body] is not a JSON or signature is invalid.

/test/jwt/invalid/[header|body]/[part]

Will respond with a JWT where part inside [header|body] is invalid.

/test/rpc

Will handle any request that can be made to the standard RPC endpoint, but will not use actual data, but instead a `keyId` based static storage:
- 2 keys with propertyId "`cHJvcGVydHktdGVzdC1pZA`" and keyIds `["MS1rZXktdGVzdC1pZA", "Mi1rZXktdGVzdC1pZA"]` with `userObject` with `id="MS11c2Vy"`
- grantKey will always return the same keys with keyId "`My1rZXktdGVzdC1pZA`" (no keypad) and/or “My2rZXktdGVzdC1pZA” (with a keypad) with a fixed *userObject* with *userObject* with id=”Mi11c2Vy”
- changeKeyValidity will always return the same keyId "NC1rZXktdGVzdC1pZA" or “My2rZXktdGVzdC1pZA” with a fixed *validityObject*
- deleteKey will return true, but key will still be returned on next request

7. Scenarios

Issue key to user based on a reservation

req> {"id": 1, "method": "grantKey", "params": {"source": {"$type": "property", "id": "[propertyId]"}, "user": {"$type": "user", "email": "guest@example.com", "meta": {"userId": "[userId]"}}, "validity": {"from": "20190214T14:00:00Z", "to": "20190215T10:00:00Z"}, "meta": {"reservationId": "[reservationId]"}}}

res> {"id": 1, "result": [{...keyObject...}, {...keyObject...}]}

Change keys when reservation changes

1) Get user keys
req> {"id": 2, "method": "getKeys", "params": {"source": {“$type”: “property”, “id”: "[propertyId]"}, "user": {"$type": "user", "email": "guest@example.com"}}}

res> {"id": 2, "result": [{...keyObject...}, {...keyObject...}]}

2) Change keys
req> {"id": 3, "method": "changeKeyValidity", "params": {"key": "[keyId]", "validity": {"from": "20190214T14:00:00Z", "to": "20190216T10:00:00Z"}}}

res> {"id": 3, "result": {...keyObject...}]}

Delete keys when reservation is cancelled

1) Get user keys
req> {"id": 4, "method": "getKeys", "params": {"source”: {“$type”: “property”, “id”: "[propertyId]"}, "user": {"$type": "user", "email": "guest@example.com"}}}

res> {"id": 4, "result": [{...keyObject...}, {...keyObject...}]}

2) Delete keys
req> {"id": 5, "method": "deleteKey", "params": {"key": "[keyId]"}}

res> {"id": 5, "result": {...keyObject...}]}

Open doors

req> {"id": 6, "method": "useKey", "params": {"key": "[keyId]"}}

res> {"id": 6, "result": true}

Issue key to cleaner

req> {"id": 7, "method": "grantKey", "params": {"source": {"$type": "property", "id": "[propertyId]"}, "user": {"$type": "user", "email": "cleaner@example.com", "meta": {"employeeId": "[employeeId]"}}, "validity": {"from": "20190215T11:00:00Z", "to": "20190215T13:00:00Z"}}}

res> {"id": 7, "result": [{...keyObject...}, {...keyObject...}]}

Find cleaners' keys

1) Get user keys
req> {"id": 8, "method": "getKeys", "params": {"source": {"$type": "property", "id": "[propertyId]"}, "user": {"$type": "user", "email": "cleaner@example.com"}}}

res> {"id": 8, "result": [{...keyObject...}, {...keyObject...}]}