MFG227312

Power Up Your Integration with Fusion Lifecycle Using the New v3 API

Giliar de Fries Perez
Autodesk Canada Co.

Igor Cunko
Autodesk Canada Co.

Learning Objectives

- 3-legged OAuth flow vs. 2-legged OAuth flow
- Authentication in the v3 API
- Impersonation
- Data, metadata, bulk endpoints
- Traversing the API

Description

This document covers the principles guiding the architecture of the v3 API, basic usage, and best practices. It describes how you can leverage existing, documented v3 API resources in Fusion Lifecycle to power up your integration or custom application. For yet-to-be-documented resources, this document also provides some guidance on how to figure out the capabilities that are being built in the API. It's our intention that, whenever possible, customers switch to the v3 API. We are also open to feedback on most-used capabilities, and what can we do to make your usage of the API easier.

Disclaimer: While the v3 API is under constant development, the development team won't push any breaking changes to resources documented in our Help files until versioning is in place, allowing you to point to a specific version, and rely on its capabilities.

Speakers

Giliar de Fries Perez is currently the **Sr. Product Owner** for **Fusion Lifecycle**, working closely with the development team defining the scope of work, requirements, delivering incremental business value to customers, and connecting the feedback from customers to the developers. Passionate about enabling customers to use the system and API to achieve their goals and enabling the development team to work on what matters to the customers. With more than 20 years of experience working with the web, he previously worked as a frontend developer in in Fusion Lifecycle.

Igor Cunko is currently the **Architect** for **Fusion Lifecycle**, working with bits and bytes for 30+ years. Worked with start-ups and enterprises on various backend implementations. In the last 8 years, moved to cloud-based implementations using mostly Java for the backend. Occasional gamer, and love to watch kids playing hockey.

Why a v3 API

Development of the API started a few years ago and is the way forward for not just the new UI (also known as Modern), but also for all integrations (existing or new) that leverage FLC. Some reasons for switching to the v3 API:

- **More secure**: Support for Bearer 3-legged/2-legged OAuth token-based authentication, instead of session cookies.
- **Consistency**: the structure of the payload and values is standardized across the resources.
- **Standardized query parameters for collections**: fetching multiple pages, or specific pages, of a collection of elements is consistent (when applicable).
- **Navigability/discoverability**: enables traversing the resources by following the links in the payloads.
- **Uniqueness**: each resource contains a URN, eliminating ambiguity.
- **Support for accept headers, when applicable**: the same resource can return data, metadata, or bulk data.
- **Performance**: optimized logic in the backend, especially when calculating list of values.
- **Robustness**: more validations when persisting data.

Conventions for this document

Please note the following conventions used throughout this document:

```
Requests (GET/PUT/POST/DELETE/PATCH/etc.) are colored red.
Header-related information for requests are colored purple.
Request payloads (i.e. sent to the API) are colored blue.
Response payloads (i.e. returned by the API) are colored light
gray.
```

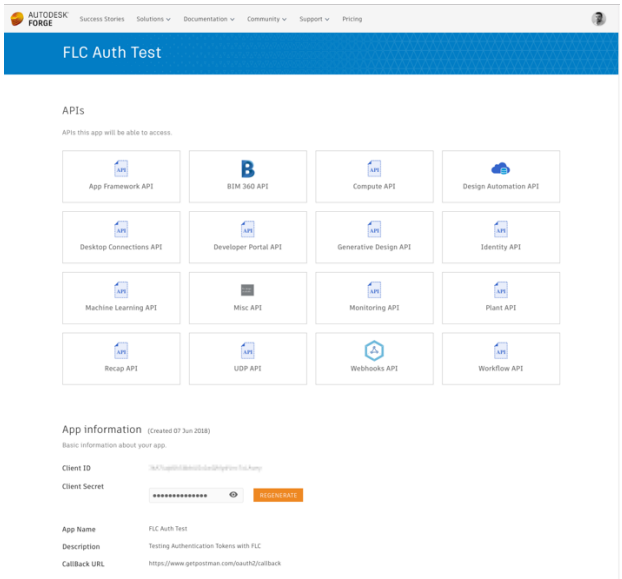
Why token-based authentication

Most integrations running against the v1, v2, and even v3 APIs currently use session-based authentication. The integration stores the username/password, logs into the API, generates a session, and stores that information in the integration. As part of the v3 API, customers are strongly encouraged to move to token-based authentication (also known as JWT – JSON Web Tokens) for the following reasons:

1. **Security**: session-based authentication requires storing user credentials in the integration, which is a huge security risk. With 2-factor authentication systems being pushed for more users, this can quickly become a non-starter.
2. **Resiliency**: sessions expire after a certain amount of time, and subject to being persisted on the server-side, or network being stable. Depending on the server load, or network conditions, the session can expire, thus requiring a new one to be generated (increasing complexity on the integration).
3. **Maintenance:** tokens enable not just impersonation (see below), but also revoking credentials more easily (e.g. blacklisting an offending client, or changing the secret), whereas sessions require the user to be deactivated (or passwords to be modified).

3-legged vs 2-legged OAuth flows

All OAuth flows use the Autodesk Forge platform to generate tokens. User authentication in Fusion Lifecycle uses the *Identify API* in the backend, which is the same available to you. A tenant user, using Autodesk Accounts, can log into the Autodesk Forge portal, and create a free app that has access to the Identity API to generate tokens – and yes, this is the same user used to log into your tenant.



Forge App settings screen.

3-legged OAuth

In the scenario, the three legs consist of the **end-user** (also called resource owner or, in our world, the tenant user), the **client** (in our case, the integration), and the **authorization server** (in our case, Forge). The authorization information consists of an **access_token** (lasts for 1h, can be used multiple times), and a **refresh_token** (lasts for 14 days, can be used once). Both these tokens contain information related to the user (in our case, the app owner), and he/she must explicitly grant the app authorization to generate the token: the user is redirected to a screen to enter his credentials (thus authorizing the app to get the auth information), and then sent back to the client (integration) with the retrieved token.

Why wouldn't I use 3-legged OAuth all the time?

Nothing stops you from doing so – however, the following should be considered:

1. If your app runs on demand, the credentials will get stale after some time (typically, after 1h), required a user to manually re-enter his/her Autodesk Accounts credentials.
2. Since the tokens contain user information, no impersonation is possible.
3. The **refresh_token** is extremely sensitive information and lasts for a long time.
4. It requires a redirect_url – i.e. your app must support viewing a web page, so that the end-user can enter his credentials and explicitly authorize the app to view sensitive information.

The tutorial for generating a 3-legged OAuth token is available in the Fusion Lifecycle Help.

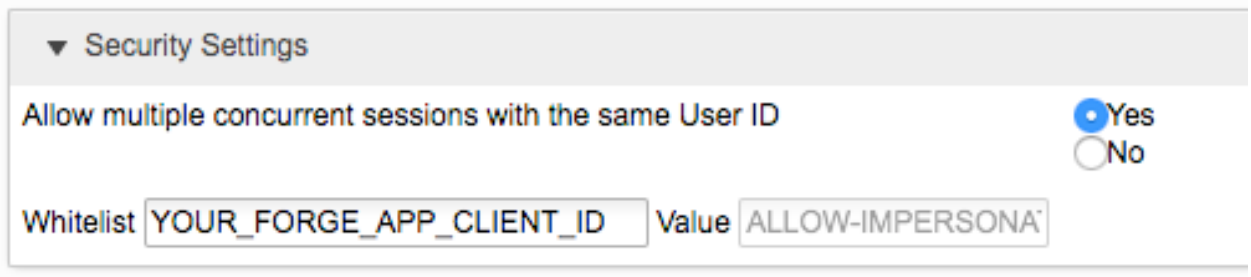
2-legged OAuth

The preferable way to run an integration is to authenticate through a **2-legged OAuth** flow. In this scenario, the **integration** (client) "talks" to the Fusion Lifecycle tenant using a token, generated by the **authorization server** (Forge), calculated using the client id and secret of the Forge app. The integration can generate tokens as needed with a simple **POST** call, whenever needed. No user intervention is needed, since the token does not contain user information (thus, he/she doesn't need to explicitly authorize it). However, in this case, Fusion Lifecycle needs to "know" that tokens generated by the **authorization server** (Forge) are allowed to get into the tenant – that's called "whitelisting the client id" (see below).

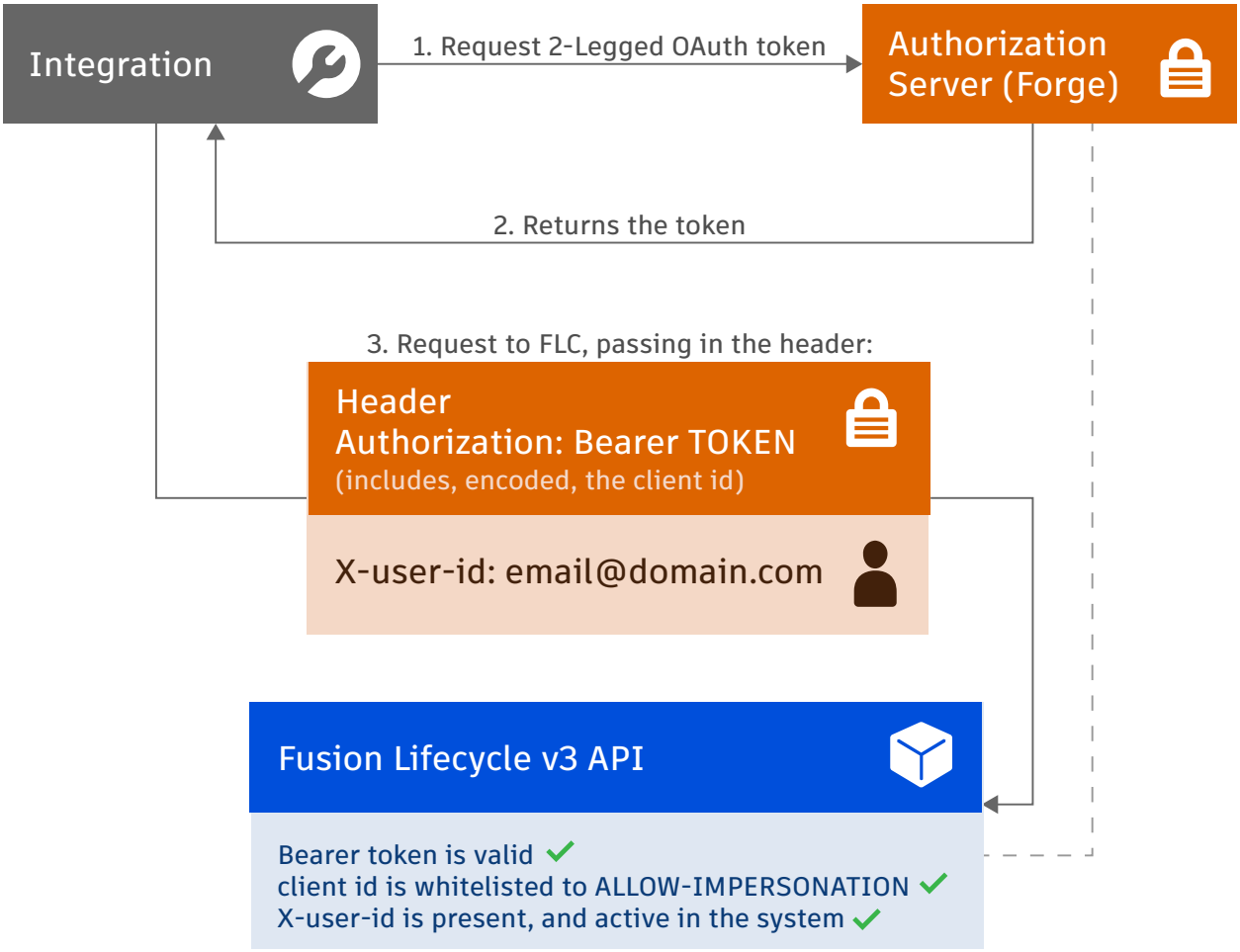
Whitelisting the client id

Every Forge app contains a client id and a secret, created under your account. When logging into Fusion Lifecycle, the 2-legged OAuth token won't have any information tied to your user, but to your app instead. Therefore, for FLC to know that tokens generated for your client id are allowed in, you need to:

1. Take note of your Forge App's client id;
2. Go to "Administration" – "General Settings" in FLC;
3. Add your client id to the form and Save.



High-level overview of the flow:



Impersonation

The preferable way to run an integration is to authenticate through a **2-legged OAuth** flow. This not only allows the client (integration) ask for tokens as needed, but also to impersonate other users in the tenant. The advantages are:

1. **Simplification:** since no user intervention is needed, the integration can generate tokens as needed with a simple POST call.

2. **Flexibility**: by impersonating another user in the system, the integration can fetch data in the context of that user's permissions (instead of having to apply filtering to the fetched content).
3. **Security**: credentials can be easily revoked from users who should not be accessing the system any longer, and those will then be unable to fetch the data.

Fetching a 2-legged OAuth token the easy way

Quick tutorial below. Full documentation is available at the Forge documentation.

Request:

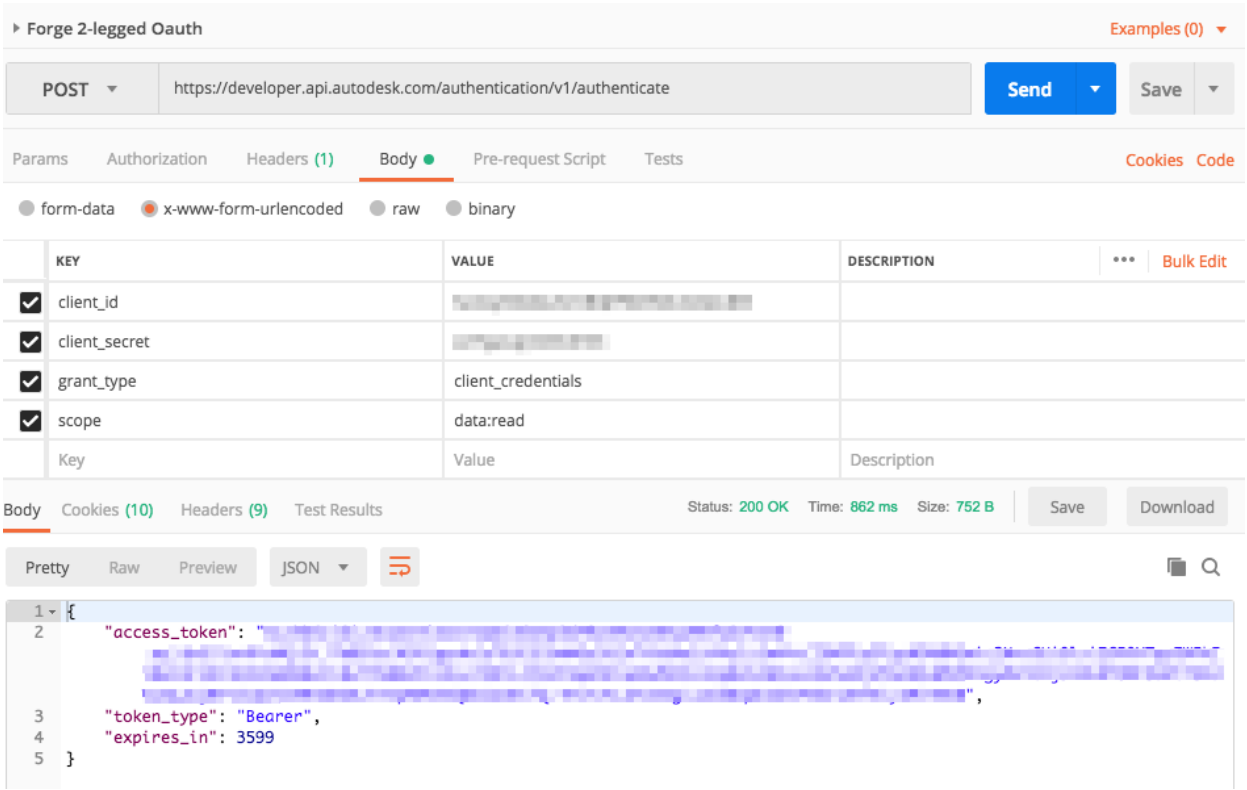
```
POST https://developer.api.autodesk.com/authentication/v1/authenticate
```

Headers:

```
Content-Type: application/x-www-form-urlencoded
```

Body:

```
client_id: YOUR_APP_CLIENT_ID
client_secret: YOUR_APP_CLIENT_SECRET
grant_type: client_credentials
scope: data:read
```



Sample request using POSTMAN.

Fetching data from the v3 API using 2-legged OAuth tokens

Now for the fun part. You need to pass a bit more information to the v3 API when performing REST operations:

Request to an endpoint containing user information:

```
GET https://mytenant.autodeskplm360.net/api/v3/users/@me
```

Headers:

```
Authorization: Bearer YOUR_2_LEGGED_OAUTH_TOKEN
Accept: application/json
X-user-id: YOUR_EMAIL
X-Tenant: YOUR_TENANT_NAME
```

Please note two key parts of this flow:
`X-user-id` is the **e-mail address** of the user who should be impersonated. It **must** be provided – remember, the 2-legged OAuth token is not tied to a particular user, so FLC "doesn't know" who is trying to get into the system, just which app (which you have whitelisted – see the high-level diagram above).

`X-Tenant` is good practice to be provided, especially in scenarios where the user participates in multiple tenants.

Data, metadata, bulk endpoints

The v3 API operates with three types of `Accept` header:

Data

This call is always going to return plain data. Most of the calls done to the API return this kind of information.
`Accept: application/json`

Example: resource containing data of views of a workspace.

```
GET https://mytenant.autodeskplm360.net/api/v3/workspaces/47/tableaus
{
  "__self__": "/api/v3/workspaces/47/tableaus",
  "tableaus": [ {
    "link": "/api/v3/workspaces/47/tableaus/235",
    "urn": "urn:adsk.plm:tenant.workspace.tableau: MYTENANT.47.235",
    "title": "My Default View",
    "deleted": false,
    "type": "DEFAULT"
  }, {
    "link": "/api/v3/workspaces/47/tableaus/418",
    "urn": "urn:adsk.plm:tenant.workspace.tableau:MYTENANT.47.418",
    "title": "A New View",
    "deleted": false
  } ]
}
```

Metadata

"Data about the data". Most of these calls can be cached, as metadata rarely changes.

```
Accept: application/vnd.autodesk.plm.meta+json
```

Example: same endpoint as above, now returning metadata.

```
GET https://mytenant.autodeskplm360.net/api/v3/workspaces/47/tableaus
[{
    "field": {
        "__self__": "/api/v3/workspaces/47/views/0/fields/DESCRIPTOR",
        "urn":
"urn:adsk.plm:tenant.workspace.view.field:MYTENANT.47.0.DESCRIPTOR",
        "title": "Item Descriptor",
        "type": {
            "link": "/api/v3/field-types/4",
            "urn": "urn:adsk.plm:tenant.field-type: MYTENANT.4",
            "title": "Alpha Numeric",
            "deleted": false
        },
        "value": null
    },
    "group": {
        "label": "ITEM_DESCRIPTOR_FIELD",
        "name": "Item Descriptor",
        "section": 15,
        "displayName": "Item Descriptor",
        "displayOrder": 0
    },
    "allowMultipleFilters": true,
    "applicableFilters": [
        {
            "link": "/api/v3/filter-types/2",
            "label": "report.filter.type.contains",
            "name": "Contains",
            "allowValue": true,
            "valueType": "STRING"
        },
        {
            "link": "/api/v3/filter-types/3",
            "label": "report.filter.type.starts_with",
            "name": "Starts With",
            "allowValue": true,
            "valueType": "STRING"
        },
        (…)
```

Bulk data

Support for adding bulk fetching of data from the database is being added as a case-by-case basis, since some requests can be very expensive. At the moment, the following areas are covered by bulk endpoints:

Bill of Materials

`Accept: application/vnd.autodesk.plm.bom.bulk+json`

The data returns the root node of the bom, and the definition or the columns of a particular view. The bulk data returns the first three levels of assemblies.

Item Details sections

`Accept: application/vnd.autodesk.plm.sections.bulk+json`

The data returns a list of sections and basic information, the bulk data returns the fields inside the sections as well.

Traversing the API

All endpoints contain enough information for the developer to traverse the API. Example: structure of a resource containing data of a workspace.

```
GET https://mytenant.autodeskplm360.net/api/v3/workspaces/47
{
    "__self__": "/api/v3/workspaces/47",
    "urn": "urn:adsk.plm:tenant.workspace:MYTENANT.47",
    "name": "Products",
    "description": "",
    "category": {
        "name": "Product Development",
        "icon": "/images/home/approval_24.png"
    },
    "type": "/api/v3/workspace-types/2",
    "fields": "/api/v3/workspaces/47/fields",
    "sections": "/api/v3/workspaces/47/sections",
    "views": "/api/v3/workspaces/47/views",
    "tableaus": "/api/v3/workspaces/47/tableaus",
    "permissions": [
        "Delete",
```

```
            "Read",
            "Create",
            "Update"
        ],
        "scripts": "/api/v3/workspaces/47/scripts",
        "deleted": false,
        "displayOrder": 1
}
```

Most resources follow the same convention:
`__self__` is the link of the resource to itself, `urn` is the unique resource name across the v3 API, and the other `keys` are links to other resources related to the current payload.

Collections

Payloads containing data structures that can be paginated are known as collections. All collections in v3 API are standardized to a common structure whenever possible, and follow the same rules for pagination.

Example: change log of an item.

```
GET https://mytenant.autodeskplm360.net/api/v3/workspaces/57/items/7825/logs
{
    "__self__": "/api/v3/workspaces/57/items/7825/logs?offset=0&limit=10",
    "offset": 0,
    "limit": 10,
    "totalCount": 23,
    "first": {
        "link": "/api/v3/workspaces/57/items/7825/logs?offset=0&limit=10",
        "title": "First",
        "deleted": false,
        "count": 10
    },
    "next": {
        "link": "/api/v3/workspaces/57/items/7825/logs?offset=10&limit=10",
        "title": "Next",
        "deleted": false,
        "count": 10
    },
    "last": {
        "link": "/api/v3/workspaces/57/items/7825/logs?offset=20&limit=10",
        "title": "Last",
        "deleted": false,
        "count": 3
    },
```

```
"items": [
(…)
```

`offset` is the number of items skipped from the beginning of the list, `limit` is for how many items are going to be returned in the payload.are used for pagination, and can be passed as query parameters. `totalCount` is the total amount of items in this collection. `first`, `next`, `last` represent pages in the collection – the most useful of these links is next, since it allows your integration to easily follow the link to the next page.

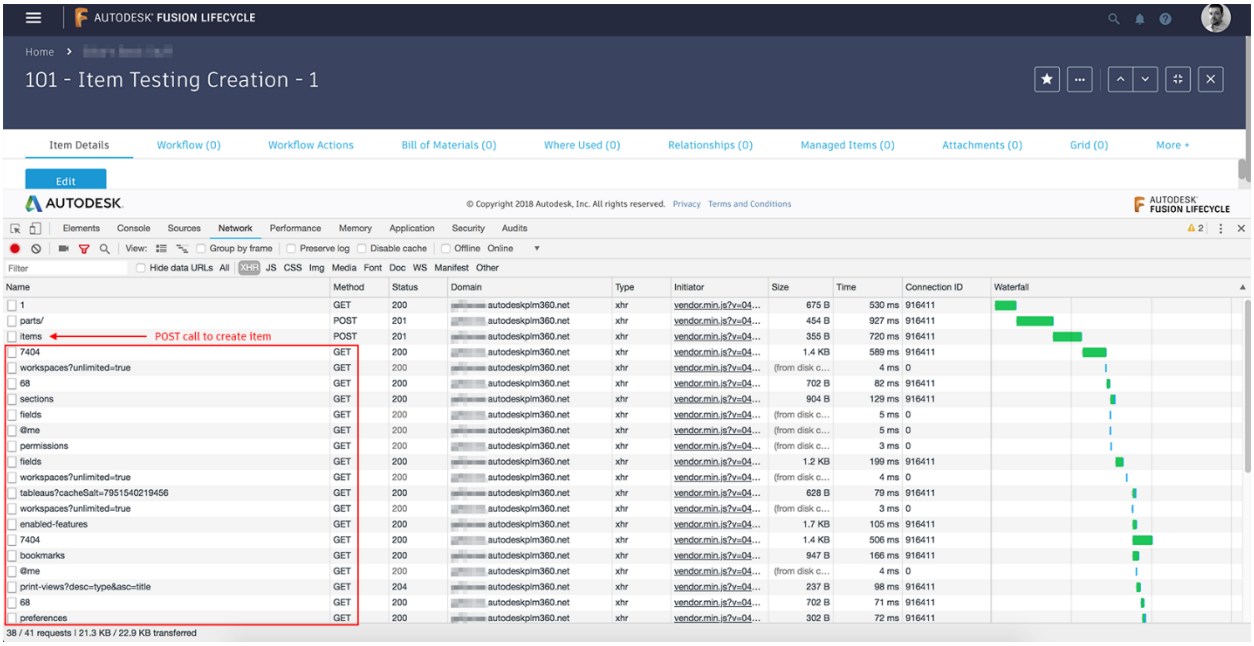
Some endpoints support query parameters `page` (instead of `offset`), and `size` (same as `limit`) for legacy purposes and will be kept consistent in this version of the v3 API, until versioning is in place.

Tips and tricks

- The v1 and v2 APIs also work with 3-legged OAuth tokens
- 2-legged OAuth tokens are only supported by the v3 API
- Remember to build resilient code checking for `403 Unathorized` response codes – that means your integration should re-fetch a 2-legged OAuth token (by default, 2-legged OAuth tokens generated by Forge expire after 1h)

- **Modern** is a thin client on top of the v3 API. Browsing the client with the Network panel open in the browser is a good way to check what are the current capabilities and what is available.

  Example: this screenshot shows the POST request performed when creating a new record, and subsequent calls to get data to render information in the user interface:



- As a **general rule**, what you `GET` is what should be `PUT` back. I.e. any payload that is returned by a `GET` is what should be `PUT` back when persisting data (in the applicable cases). E.g. Editing, then Saving Item Details.
- **Impersonation** works with the value of `X-user-id` being either an e-mail address (very easy), or the value of `externalAuthUserId` of a user.
- Some requests for collections of data coming from the UI contain the query parameter `?unlimited=true`. This is not supported across all collections, as they're potentially expensive to run, and lend themselves to abuse.
- You can use URNs to fetch data in the application. Some integrations (or developers) prefer URNs to using regular paths to resources. You can achieve this by using any URN after the `/api/v3/` part:
  `GET https://mytenant.autodeskplm360.net/api/v3/urn:adsk.plm:tenant.workspace.tableau:MYTENANT.57.212`

More reading

[1] http://adndevblog.typepad.com/cloud_and_mobile/2016/07/landing-your-forge-oauth-authentication-workflow.html
[2] https://forge.autodesk.com/blog/about-refresh-token